Using Large Language Models To Analyze Software Architecture Documentation

Bachelor's Thesis of

Robin Maximilian Schöppner

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr.-Ing. Anne Koziolek
Second reviewer:	Prof. Dr. Ralf Reussner
Advisor:	M.Sc. Jan Keim
Second advisor:	M.Sc. Dominik Fuchß

29. May 2023 – 29. September 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Abstract

Limited training data poses a challenge for Traceability Link Recovery (TLR) and Inconsistency Detection (ID). Large Language Models (LLMs) can solve this problem as they often do not require specific training. In this paper, we explore various techniques and methods for using GPT-4 for TLR and ID. Compared to state-of-the-art approaches, our approaches achieve similar performance in Unmentioned-Model-Element ID. However, in the discipline of Missing-Model-Element ID, we could not achieve their performance. For TLR, Chain-of-Thought-Prompting achieves the best results, but also performs worse than state-of-the-art. The results are promising, and it is reasonable to expect that more advanced LLMs and techniques will lead to improvements.

Zusammenfassung

Begrenzte Trainingsdaten stellen eine Herausforderung für Traceability Link Recovery (TLR) und Inconsistency Detection (ID) dar. Große Sprachmodelle (LLMs) können dieses Problem lösen, da sie oft kein spezifisches Training benötigen. In dieser Arbeit erforschen wir verschiedene Techniken und Methoden für den Einsatz von GPT-4 für TLR und ID. Im Vergleich mit State-of-the-Art-Ansätzen erzielen unsere Ansätze beim Unmentioned-Model-Element-ID ähnliche Leistung. In der Disziplin der Missing-Model-Element ID konnten wir ihre Leistung jedoch nicht erreichen. Beim TLR erzielt Chain-of-Thought-Prompting die besten Ergebnisse, schlägt jedoch auch schlechter ab als State-of-the-Art. Die Ergebnisse sind jedoch vielversprechend und es ist anzunehmen, dass fortschrittlichere LLMs und Techniken zu Verbesserungen führen.

Contents

List of Figures

List of Tables

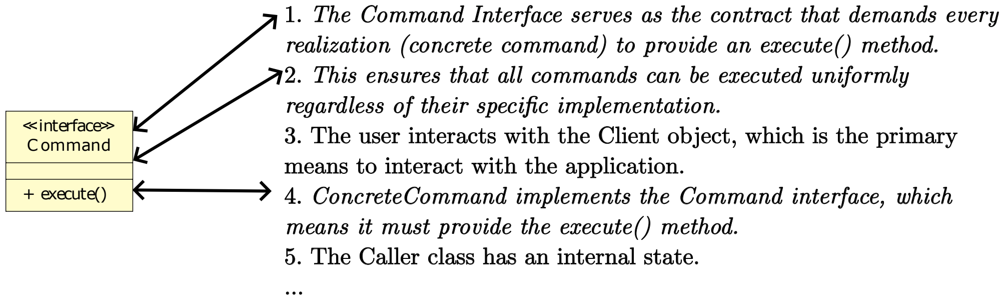
1. Introduction

In professional software engineering, engineers often take several steps from the client specifying their needs until developing and building the software[1]. In this thesis, we will focus on different artifacts produced in the process of developing software. These artifacts can include various types of models, such as use case, design, and architecture models, as well as natural language descriptions like design and requirements documents. The source code itself is also one of these artifacts[29]. During software development, some concerns about functionality or architecture decisions can be described in different sections within the same artifact or within entirely different artifacts. This means that those artifacts can have dependencies on each other[35].

To fully understand elements of the system, one must read and understand all sections relevant to these elements. For a software engineer, keeping track of these dependencies can be highly time-consuming: Xia et al. estimate that developers spend roughly 58% of their time on program comprehension[57]. Program comprehension defines the "process where developers actively acquire knowledge about a software system by exploring and searching software development artifacts and reading relevant source code and/or documentation"[57]. In addition, inconsistencies can arise across these dependencies. For instance, the same element may have contradictory specifications or be omitted in one of the artifacts.

These inconsistencies can go unnoticed for a long time until they become relevant and are discovered. In this case, they must be resolved. Thus, it is beneficial to identify inconsistencies early so that developers can react. Some inconsistencies require immediate action, while others may be tolerated, depending on their priority and relevance. However, any undetected inconsistency represents a problem, as it cannot be adequately assessed by the developers[27]. Traceability links can help address inconsistencies between different software development artifacts. They provide a formal specification of any information across various artifacts corresponding to the same element or topic. For instance, a traceability link for a component in the software architecture model points to the corresponding sentences in a design document. The design document includes various information about the class, such as its purpose and relationships with other classes. Using traceability links, developers can track and compare different specifications, which can help them identify and resolve inconsistencies more effectively. This can improve the overall quality of the software development process and reduce the risk of errors and defects. Figure 1.1 shows an example of traceability links between a class and documentation.

Having these traceability links available, developers can easily navigate between different artifacts and, thus, improve their program comprehension. Additionally, having these traceability links available makes finding and evaluating inconsistencies easier, as information about the same element can be assessed using its traceability links.



1. *The Command Interface serves as the contract that demands every realization (concrete command) to provide an execute() method.*
2. *This ensures that all commands can be executed uniformly regardless of their specific implementation.*
3. The user interacts with the Client object, which is the primary means to interact with the application.
4. *ConcreteCommand implements the Command interface, which means it must provide the execute() method.*
5. The Caller class has an internal state.

...

Figure 1.1.: An example of traceability links between an interface "Command" on the left and some documentation on the right. Arrows represent traceability links to the relevant sections in the documentation, which are marked in bold.

According to Raúl Lapeña et al., "establishing and maintaining traceability links between software development artifacts is a time-consuming, error-prone, and person-power intensive task"[30]. Therefore, we aim to find a method of automating the process of Traceability Link Recovery (TLR). So far, others have used different methods to try and achieve this goal, like utilizing neural networks and other methods in the field of Natural Language Processing (NLP) (see chapter 3).

A separate field is dedicated to Inconsistency Detection (ID) within or between software development artifacts. ID is the process of identifying inconsistencies between different or the same kind of artifacts. There are several automated approaches, and many use NLP and Information Retrieval (IR) approaches to extract information from the artifacts and compare them. ID allows developers to find inconsistencies more efficiently, which can help them improve the quality of their software.

Recently, Generative AI has been used to solve various problems, such as generating images, music, or text. Large Language Models (LLMs) are a type of Generative AI trained to generate text on the basis of a prompt[31]. In particular, an LLM is a "deep learning algorithm that can recognize, summarize, translate, predict and generate text and other content based on knowledge gained from massive datasets"[31]. Among various use cases, *ChatGPT* stands out as it has gained a large user base in a short amount of time[23]. *ChatGPT* is a chatbot capable of generating many kinds of text, including essays, articles, code, and even completing some reasoning tasks[9]. This makes LLMs an interesting candidate for automating finding traceability links and inconsistencies in software development artifacts. While there is some research[39] on using LLMs for TLR, it is still quite limited. More work needs to be done to fully understand the capabilities and limitations of LLMs for this task.

This leads to the following research question: **To what extent are LLMs capable of retrieving traceability links *from* and detecting inconsistencies *in* software development artifacts?**

The remainder of this thesis is structured as follows: First, we define and explain the foundations necessary for understanding the rest of this thesis in chapter 2. Following that, in chapter 3, we outline related work in the relevant fields of study. Chapter 4 details our approach before we evaluate it in chapter 5. Finally, we conclude our findings and give an outlook on future work in chapter 6.

2. Foundations

In this chapter, we define some relevant terminology and introduce key concepts relevant to the rest of this paper. Apart from going over the basics of traceability links and inconsistencies, we also introduce Language Models. We consider two directions within the field of Language Models that are important to our work: Embeddings and Large Language Models.

2.1. Software Artifacts

In software engineering, artifacts are created along the development process. For these, a number of different terms are relevant for a correct understanding.

We frequently differentiate between software architecture documentation (SAD) and software architecture models (SAM)[27]. An SAD usually describes a natural language description of the software. This could be a requirements- or a design document outlining the system's architecture. On the other hand, an SAM often follows a standardized modeling language, such as the *Unified Modeling Language* (UML) or the *Palladio Component Model*. These are the main types of artifacts relevant to software comprehension[57] and the ones we focus on in this paper.

2.2. Inconsistencies

Inconsistencies can appear across many artifacts. It is possible for two forms of SADs and two forms of SAMs to contradict each other. However, when comparing an SAD and an SAM directly, we can use the following definitions, which allow for a more detailed analysis. We can compare our approach more effectively because of the work done by Keim et al.[27], which also focuses on these two categories. By tracking and comparing performance in these two categories, we can make targeted adaptations to our approach to improve performance.

Unmentioned Model Elements

Detecting unmentioned model elements means finding named entities in an SAM and comparing these to an SAD. Elements that can not be found in the corresponding SAD should be labeled as Unmentioned Model Elements (UME).

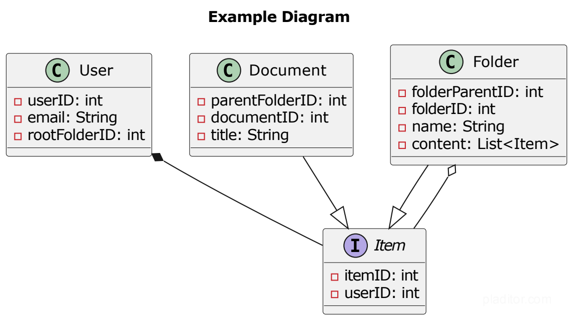
Missing Model Elements

Inversely to the UME, a Missing Model Element (MME) describes a named entity that appears in an SAD but not in the corresponding SAM.

Example

A file system during development: During the testing phase, a developer wants to check whether a class fulfills requirements and design constraints. From looking at the source code, it is not apparent where this functionality is specified in the requirements document. This is especially true when names don't exactly match.

Automated TLR could help by providing corresponding sections of available artifacts throughout the development process.



1. The Document and Folder implement the Item interface.
2. Each document has a Tag associated with it.
3. The Item interface requires an itemID and userID.

Figure 2.1.: A possible class diagram and corresponding SAD generated in the design phase.

Visible in Figure 2.1 are the following inconsistencies:

Inconsistencies

Unmentioned Model Element: The User object of the model is not mentioned in the design description
Missing Model Element: The model does not include the Tag class, which is described in the text but not modeled in the class diagram.
Missing Model Element: The collection relationship between the Folder and Item classes is missing in the textual description.

2.3. Large Language Model

LLMs can be classified as a type of generative AI that is built to generate text[31]. Many LLMs are based on the transformer model, which was initially proposed by Google[54]. Since then, two major directions have been developed: encoder-style and decoder-style transformers[38]. For generative AI, we will be mainly focusing on the decoder-style transformer.

Tokenization

For NLP, it is common to split a text into individual tokens. A token often represents a word or a substring of a word. Each token can be represented as an integer; thus, the input string becomes a sequence of integer numbers, each representing a token. This process is called tokenization[16]. GPT-3's tokenizer, for instance, would split the sentence "This is a tokenized sentence." like so:

This is a tokenized sentence.

One token roughly equates to 0.75 words in the English language[51].

Training

During training, the model is fed a large amount of textual data[19] to perform a task known as language modeling[17]. As a result, the model should be able to predict the next token in a sequence of tokens.

Language modeling can be done in a number of different ways, two of which we describe here: Masked Language Modeling and Text Generation.

Masked Language Modeling

In masked language modeling, a token is omitted from the input sequence, and the model is prompted to fill that token[17]. For instance, a model could be tasked to fill this sequence:

"Fire trucks can easily be recognized by their [MASK] color."

The model can be trained to fill in the correct token depending on the response and the original input sequence.

Text Generation

In text generation, a context is given, and a text that is supposed to be a continuation of the context is generated[17]. For instance, a model could be tasked to continue this sequence:
"In the early days, fire trucks"
The model then generates a continuation of the sequence, usually a set number of tokens, which could be:
"In the early days, fire trucks were, in fact, nothing more than water pumps on wheels."

Evaluation is more complex and is actively being researched. A popular approach is BLEURT[45], which is an evaluation mechanism based on BERT[13].

These large language models have shown great capabilities in natural language tasks such as serving as a chatbot or answering complex knowledge questions.

2.4. Embeddings

An embedding is a vector representation of a text. Given a text, the embedding of that text is a vector v representing the content's semantic information. v is a multidimensional vector, with each element usually close to 0; $v_i \in [-1.0, 1.0]$. This allows v to be compared using a common form of distance calculation, see Figure 2.2, which yields a similarity measure between two vectors.

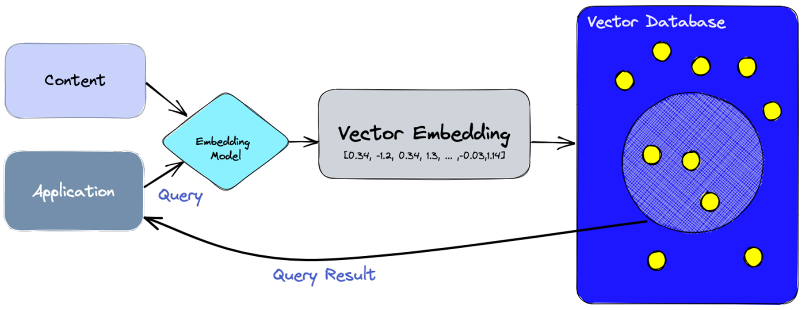


Figure 2.2.: How an application could use vector embeddings to semantically compare a piece of content to a database of other content, taken from[44].

To build a knowledge base, these embeddings will then be stored inside a vector database and efficiently searched for matching entries. For a search term, a new query vector q of that search term is generated and compared to the vectors in the database. Often, this is coupled with a threshold value that limits the distance for a match. A text v is considered a match if the distance between q and v is below the threshold. This is also visualized in Figure 2.2. The circle in the vector database represents the threshold value. All texts within the threshold radius are considered a match.

2.5. Prompting

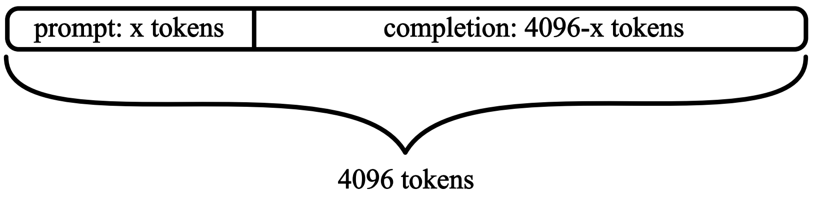


Figure 2.3.: Relationship between prompt and completion, 4096 tokens may not be exceeded when using a model with a token limit of 4096 tokens.

A prompt is a text input given to a generative LLM to produce a text output that continues or completes the prompt. A prompt, along with its completion, cannot exceed a specific token limit, which depends on the size and architecture of the LLM. This is illustrated in Figure 2.3. For instance, OpenAI's GPT-3 model has a 4097 token limit, which means that if the input prompt contains 4000 tokens, the response may not exceed 97 tokens. This token limit poses a great challenge for working with and generating large texts.

Prompting describes the practice of writing and adapting a natural text prompt, which then gets put into a transformer model to generate a response or a continuation of that prompt.

Expanding upon the concept, the field of prompt engineering has emerged, which concerns itself with strategically forming a prompt that leads to a higher quality response[48].

Prompting Methods

There is a large community around prompt engineering. Apart from OpenAI's own channels, there are other groups that specialize in describing different approaches to prompt engineering. For instance, *Learn Prompting*[43] is a group that has compiled a collection of different prompting methods:

A few approaches that could be used for our problem could be:

- Zero-Shot Prompting (Plainly asking the model for the answer, without any previous instructions on format or goal)

Example of a Zero-Shot Prompt
Multiply 2 and 3:

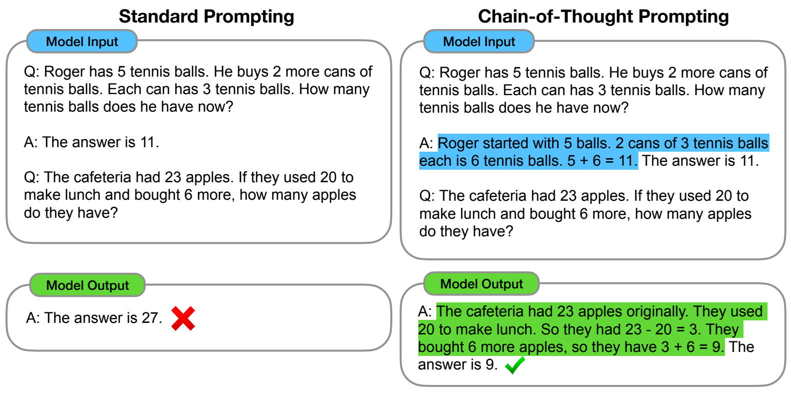
- 1-Shot / Few-Shot Prompting (Question-Answer-Question Format; Give **examples** on how a question of this kind should be answered before asking the question we are after)

Example of a 1-Shot and 2-Shot/Few-Shot Prompt
Multiply 4 and 8: 32 Multiply 2 and 3:
Multiply 4 and 8: 32 Multiply 1 and 5: 5 Multiply 2 and 3:

- Chain-of-Thought Prompting[56] (Question-answer-format with all looked at data in answer. This approach is primarily useful for multistep questions, where "human-written reasoning steps for all [...] examples"[11] are provided.), as visualized in Figure 2.4.

- Self-Consistency (Multiple results, take majority answer)[56]

  Self-Consistency prompting involves querying a model multiple times, using the same prompt, and using the majority result, as visualized in Figure 2.5.



Standard Prompting

Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model Output

A: The answer is 27. ✗

Chain-of-Thought Prompting

Model Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

Model Output

A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9. ✓

Figure 2.4.: Comparison between standard 1-Shot and Chain-of-Thought prompting methods, taken from [56].

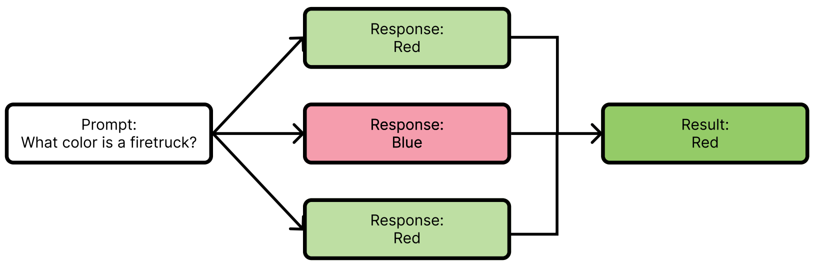


Figure 2.5.: Visualization of Self Consistency prompting, as defined by [55].

- Tree-of-Thoughts (Evaluate multiple, different solution attempts)

  Tree-of-Thoughts prompting is another prompting method that can outperform other approaches in some scenarios[58]. In the approach initially presented by Yao et al.[58], the initial prompt encourages thought decomposition. Thoughts can then be generated using one of two different methods: Sampling or proposing thoughts. Once these thoughts are collected, the LLM rates them in terms of how close they are to solving the problem. For this, they again propose two different methods: Evaluating thoughts independently or voting for "good" thoughts in comparison to others. They expand the tree using a search algorithm, such as depth-first-search (DFS) or breadth-first-search (BFS), which finds the next thought to expand upon. This process is repeated until we arrive at a solution based on the rating of the LLM for this thought.

- Prompt-Chaining (Using the output of previous prompts as input for new prompt)

  Potential for expansion: Making use of a knowledge base (Build a prompt by finding relevant information in the knowledge base)

- Prompt-Debiasing (in-prompt examples should be balanced, e.g., when using Few-Shot Prompting, the examples should be balanced.)

Example of a prompt biased to answering "Yes"

Q. Is Paris the capital of France?
A. Yes
Q. Is London the capital of the United Kingdom?
A. Yes
Q. Is Berlin the capital of Spain?

In the example above, it becomes apparent how a model might be inclined to follow the pattern of previous answers despite the factual correctness of the answer. By balancing the positive and negative answers, a prompt can be debiased.

2.6. Hallucinations

LLMs tend to 'hallucinate' information that was not provided to them in any way[24]. An example:

Prompt and Response

"Translate this sentence into English: Paul geht zur Bücherei" (Paul walks to the library)

"Paul walks happily to the library."

Here, "happily" cannot be inferred from the prompt, yet the model confidently says so.

3. Related Works

There is different related work that addresses the challenges of TLR and ID. In particular, research into TLR, ID, Large Language Models (LLMs), and Vector Space Models (VSMs) can be relevant to our research question. In the following sections, we will outline the most relevant research in these fields.

3.1. Traceability Link Recovery

TLR is an active field of research with many different approaches. IR and artificial intelligence methods are examples of these approaches and are particularly relevant for finding traceability links.

IR is a field of research that deals with finding relevant information in a large amount of data. In particular, "IR methods determine how relevant the document representation is to the query that represented user information need."[21]. In the context of TLR, IR methods can be used to identify relevant artifacts and extract information that can be used to establish traceability links[42].

Rodriguez and Carver[40] present different IR techniques to find traceability links and compare them. They make a case for several IR methods and present the method behind each. Outside the scope of their paper, others have applied machine learning[46] to retrieve information from artifacts. Also of great importance is the work done by Bouillon et al.[6] about usage scenarios for TLR based on IR. Schlutter et al. propose a TLR mechanism utilizing Semantic Relation Graphs and spreading activation[42]. They first construct a knowledge graph to represent the semantics of a sentence. Once this is complete, they use spreading activation to find nodes in the graph with higher relevance for traceability links. These approaches are relevant to our research as they show how IR can be used to find traceability links and how we can employ them to improve the performance of LLMs.

Artificial Intelligence (AI) is another field of research often used in TLR. An example of this is the approach developed by Guo et al.[20], which utilizes deep learning, a specialized area within the field of AI[26], to generate traceability links. They use word embeddings to train a Recurrent Neural Network (RNN) to recognize the sentence semantics of a requirements document. They find that their approach outperforms all other approaches that existed at the time. However, they recognize that the need for a large dataset and the difficulty of creating such a dataset is a major obstacle to their approach. Thus, they suggest looking at hybrid approaches in future work[20, p. 12]. Our research is closely related to this work: As another specialized area in the field of AI[26], LLMs could be used to improve automating TLR.

Du et al.[14] try to address this challenge of limited data availability on TLR by using an active learning approach. They hope to gain higher performance from the limited dataset available. Their work suggests that their active learning approach outperforms the more traditional IR approaches[14]. However, no comparison exists between the approaches by Guo et al.[20] and Du et al.[14]. This is relevant to our research as we deal with the same challenge of limited data availability.

Fuchß, Corallo, Keim, Speit, and Koziolek published a paper on building a benchmark dataset for evaluating algorithms at TLR[18]. The dataset contains a number of projects, each with its own documentation (i.e., SAD) and UML (i.e., SAM) files. Alongside the software artifacts, the dataset contains gold standard files for traceability links and unmentioned model elements. This dataset can be used to evaluate the performance of different algorithms and approaches to TLR and ID.

Another approach is traceability between artifacts presented by Hey et al. when examining *Fine-grained Requirements-to-Code Relations*[22]. They aim to improve recall levels of IR-based TLR by interpreting artifacts "in a more fine-grained manner"[22] to increase performance. This approach is relevant to our research as it highlights the importance of considering the granularity of artifacts when recovering traceability links and detecting inconsistencies.

Rodriguez et al.[39] have experimented with using LLMs for traceability link recovery. They aim to inspire further research into the use of LLMs for TLR. Their work is especially relevant to us as they also build prompts for LLMs to generate traceability links.

3.2. Inconsistency Detection

While TLR is primarily concerned with establishing and maintaining relationships between software artifacts, ID focuses on identifying and resolving inconsistencies within and across these artifacts.

One branch of ID research is the detection of inconsistencies between the same type of software artifact. For instance, this might involve detecting inconsistencies within multiple UML diagrams that overlap in their scope, as in the work by Egyed[15]. Similarly, Kim and Kim[28] find inconsistencies within implementation code based on knowledge they gain from trusted API documentation. In particular, their approach concerns itself with identifier consistency within implementation code. Using Java as an example, they build a "code dictionary" using a selection of API documents, like the *Java Development Kit* or *JUnit* API documents. In the next step, the approach detects inconsistent identifiers within the implementation code. In the next step, they use the previously constructed code dictionary as a means to filter out false positives. These approaches provide valuable insights into the challenges of TLR and ID and inform our work in this area.

However, ID is not limited to just the "intra-artifact" type inconsistencies. Often, ID concerns itself with inconsistencies across different kinds of artifacts.
Tan et al.[49] have examined inconsistencies between code and comments. Using the example of Java, they differentiate between comments within the body of a method and those that appear in the header to describe the method specifications, i.e., a Javadoc comment. They find that inconsistent Javadoc comments typically indicate a fault and should, therefore, be examined. They propose *@tComment*, an approach for testing "comment-code inconsistencies in Javadoc comments"[49, p. 1]. They use dynamic analysis

to achieve this goal.
In contrast, Steiner and Zhang[47] have utilized transformer models such as BERT[13] to find inconsistencies between comment and their corresponding methods. Like Tan et al., they also do not consider comments within the body of a method. Instead, they define their task as follows: "given a comment *C* and its corresponding code method *M*, determine whether *C* is inconsistent with *M*"[47, p. 3]. They examine the effectiveness of BERT and Longformer[4]. They find that BERT and Longformer outperform all of their baselines in the "post hoc" setting. Longformer beats BERT in this task. "Post hoc" describes the setting when all inconsistencies are already present within the code, and no modifications are considered, which we also exclusively work on in this paper. In the "just-in-time" setting, however, results are much more mixed. Their work illustrates the use of large language models in this field. Their work demonstrates the use of large language models in this field.

Going beyond comment and code pairs, Keim et al.[27] have examined ID between unstructured (natural language) software architecture documentation (NLSAD) and software architecture models, such as UML architecture diagrams. These artifacts consider the entire project or at least a large part of it rather than just a single method, as in the examples of Tan et al. and Steiner and Zhang. In their proposed *ArDoCo* approach, they work on named entity detection to generate traceability links between an SAD and its corresponding SAM. To do so, they perform TLR with a new approach that considers phrases as relevant information. This results in a more effective TLR performance according to the benchmark dataset by Fuchß et al.[18]. They then use the generated traceability links to perform ID, which improves UME and MME performance. They achieve this by filtering the identified traceability links for "unwanted words"[27, p. 5] and inferring inconsistencies. Their work shows how existing traceability links can be utilized to find inconsistencies between an SAD and a corresponding SAM.

3.3. LLMs

Finally, the field of Large Language Models has also seen a lot of research. Google famously published the paper introducing transformer models, *Attention is all you need*[54], upon which many of today's large language models are based. Further, OpenAI published a paper about the creation of *instructGPT*[36], a large language model purpose-built to follow instructions. Their work here could help to construct purpose-built LLMs. Taori et al. have constructed the Alpaca model, which is an LLM capable of running on even consumer-grade hardware[50]. Their work has started a trend of creating smaller, more efficient LLMs for many fields. Finally, Bubeck et al.[9] have thoroughly examined the capabilities of OpenAI's GPT-4 model. In their work, they present prompting methods and explore the limits of GPT-4's generative model in many fields, including coding, while considering societal impact.

Prompting A prompt is a natural language instruction by a user to an LLM[48]. Prompting is the process of finding a suitable input for an LLM to generate the desired output and achieve the desired quality of the output. The field of prompting is still relatively new and has seen many different prompting methods. A few of the most popular prompting methods are *Zero-Shot Prompting*, *Few-Shot Prompting* and *Chain-of-Thought Prompting*.

Zero-Shot prompting is a type of prompting where the LLM is given a task but no examples of the desired output. The hope is that the LLM, through a sort of "meta-learning"[8, p. 4] is capable of "rapidly adapt to or recognize the desired task"[8, p. 4].

Brown et al.[8] propose Few-Shot Prompting, which is a prompting method that provides the LLM with a few (≥ 1) demonstrations of the task and is expected to be able to complete another of the same task. This is also referred to as "in-context learning"[8, p. 4]. They find that, in many cases, Few-Shot prompting outperforms 1-Shot or Zero-Shot prompting. This work is important to my research as it shows that Few-Shot prompting is a viable approach to improving the performance of LLMs.

Chain-Of-Thought prompting is a prompting method that builds upon Few-Shot Prompting. According to Wei et al.[56], Chain-Of-Thought prompting "enables large language models to tackle complex arithmetic, commonsense, and symbolic reasoning tasks". The idea is to mimic human thought processes when solving a problem. Like with Few-Shot Prompting, the LLM is given demonstrations of the task. However, the task is split into intermediate steps and solutions before solving the task. These should form a "coherent series of intermediate reasoning steps that lead to the final answer for a problem"[56, p. 2]. Following these demonstrations, the LLM is tasked with solving the actual task. The promised benefit of this method uniquely positions it as a possible way to tackle TLR and ID using LLMs.

Further expanding upon the previous concept, Wang et al.[55] suggest Chain-of-Thought Self-Consistency. Their approach samples a number of results while employing Chain-Of-Thought prompting and takes the majority result to find a final solution. This method is relevant to us as it might allow us to increase the performance of LLMs when using Chain-Of-Thought prompting.

A different and iterative approach to prompting is presented by Yao et al. in their paper on the "Tree of Thoughts" method[58]. Instead of relying on a singular reasoning chain, they propose a tree of reasoning chains, which can be iteratively expanded. The idea is to let the model generate a number of different "thoughts" or "ideas". The model then evaluates the thoughts in terms of how close they are to solving the problem. Then, an algorithm decides which of the best thoughts to expand upon to generate new thoughts closer to the solution. This process is repeated until a solution – or something deemed a solution by the evaluation step – is found. Yao et al. show that this method provides some improvements over Chain-of-Thought prompting. However, it is also more computationally expensive. It is particularly successful than other prompting methods for tasks that can benefit from backtracking. Tree of thought prompting is well-suited for tasks where a human would try a combination and, if it fails, return to a previous state. They conclude that this method can benefit specific use cases with the right performance-cost tradeoff. It is a promising approach to solving problems with LLMs and could be helpful in our research.

Figure 3.1 shows an overview of different prompting methods.

All the prompting methods mentioned above are relevant to my research as they show how LLMs can be used to solve different problems. Some may be more uniquely fit for TLR and ID than others.

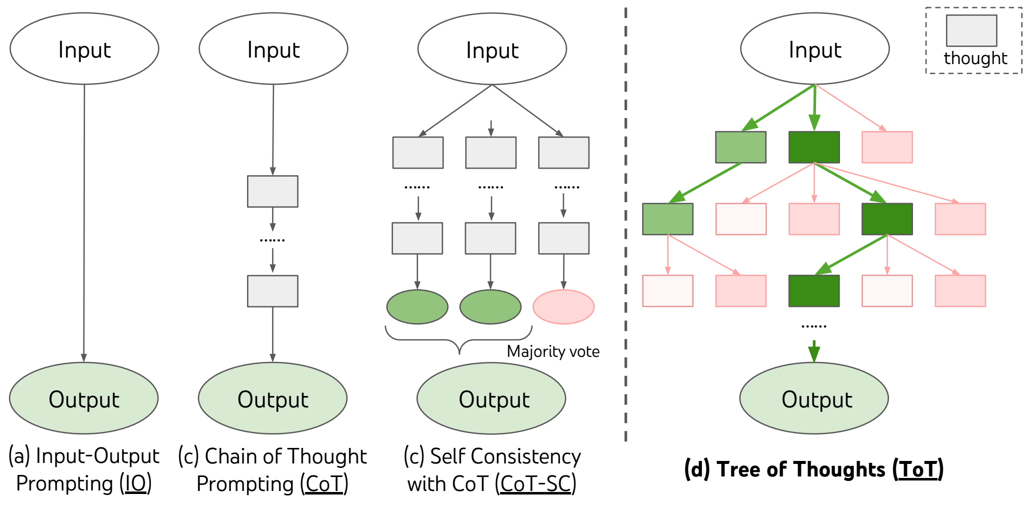

Input
Input
Input
Input
thought
Majority vote
Output
Output
Output
Output
(a) Input-Output Prompting (IO)
(c) Chain of Thought Prompting (CoT)
(c) Self Consistency with CoT (CoT-SC)
(d) Tree of Thoughts (ToT)

Figure 3.1.: Overview of prompting methods, taken from Yao et al.[58]. The rectangular boxes represent "thoughts" or "ideas" generated by the model. The color indicates the "quality" of the thought, with green indicating a "good" and red indicating a "bad" thought. The arrows indicate from which thought a new thought was generated.

3.4. Vector Space Models

Vector Space Models (VSM) represent words and sentences as vectors in a vector space. Mikolov et al.[33] propose Word2Vec, an approach often regarded as one of the most important works in the field of NLP[12, 25, 5]. Their work shows how linear relationships can emerge between vectors after training them to predict adjacent words in sentences[25]. Mikolov et al. give one example of this:

’*vector*(”King”) - *vector*(”Man”) + *vector*(”Woman”) results in a vector that is closest to the vector representation of the word Queen’[33, p. 2].

Vector Space Models have unique properties that make them useful in various tasks, which we outline here.

Turney and Pantel[53] have published a survey on vector space models (VSM). Representing documents as points in space and queries as pseudo-documents in the same space allows for a powerful search engine that returns the closest documents to the query in order. They outline how VSM models can be used in different fields and what adaptations might be necessary. When considering documents, they propose several use cases where VSMs can be helpful.[53, pp. 168-170] These include:

- Document Retrieval (Given a query, find relevant documents)
- Document Clustering (Given document similarity measure, cluster documents into groups)
- Document classification (Given training set of documents with class labels, learn to label documents)

- Essay grading (Grade student essays based on similarity to reference essay)
- Document segmentation (Partition document into segments of subtopics)
- Question answering (Given a question, find an answer within a large document corpus)
- Call routing (Ask caller questions to determine where to route call)

In our use case, **Document Segmentation** for subdividing an SAD into different subtopics and **Question Answering** for answering questions about the SAD are particularly interesting.

In other research, in their Google Cloud Blog entry, Sato and Chikanaga[41] describe how VSM also serves as the backbone of Google's search and image similarity search products. They go on to explain how VSM can be utilized beyond image and text media. They list examples of recommender systems, such as finding similar products, defective IoT devices, or even security threats.

Embeddings are vector representations of one or multiple words. This makes embeddings a type of vector space model where words are represented as points in space.

The Massive Text Embedding Benchmark (MTEB) enables comparison between different embedding models[34]. They collect datasets for different tasks and combine them into their benchmark. The results should allow users to identify the strengths and weaknesses of a model faster. This is also useful for finding a well-suited model for our specific task.

Utilizing VSM for generating traceability links has also been examined. Antoniol et al.[3] examine how vector space IR research can be applied to two case studies: They experiment with tracing C++ code to manual pages and Java Code to functional requirements. Their approach involves preparing both artifacts first and finally using a document classifier based on the VSM to calculate similarities. This process is outlined in Figure 3.2.

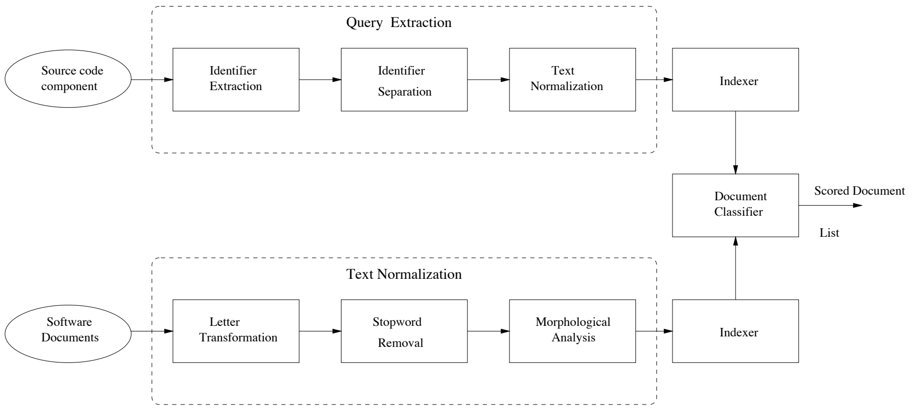


Figure 3.2.: TLR process using VSM, taken from Antoniol et al.[3].

Antoniol et al.[3] query their corpus of documents using a query generated from a source code component. The similarity value between the component query Q and document D is then calculated using the cosine angle between the respective vectors. After doing this for every component, they calculate a dynamic threshold value for every query based on the similarity value of the closest match. This threshold value is then used to filter out all Q-D pairs with a similarity value below the threshold. The remaining pairs then represent the traceability links that are returned by the algorithm. They find that their approach requires less effort in preparing queries and documents and yields better results than probabilistic IR models. This is relevant to our research as we also want to retrieve relevant documents, or sections of documents, based on a query. Liu et al.[32] similarly use a VSM to identify traceability links. They focus, in particular, on software projects with artifacts written in different languages, such as English and Chinese. They evaluate different approaches for this, including a VSM approach. These works are relevant to our research as they show how VSMs can be used to find traceability links between different artifacts.

3.5. Research Gap

Despite a lot of work in TLR and ID, we find there are relatively few publications on using LLMs for these tasks. Connecting the field of TLR and ID with the field of LLMs could be promising to improve the performance of TLR and ID. While previous studies have used either VSMs[3] or LLMs[39] for TLR, we currently, to the best of our knowledge, find no work that combines LLMs and VSMs for TLR. For all of these purposes, we want to examine how the works in prompt engineering for LLMs could aid in improving the performance of LLMs for TLR and ID.

4. Approach

We consider different methods of utilizing these models for the purpose of TLR and ID. This includes prompt engineering and using embeddings to allow models with a limited context length to analyze large architecture documents and models. Products by OpenAI, in particular, are often regarded as the market-leading models in this field. Their GPT-4 model seems uniquely fit for performing TLR and ID, as it exhibits state-of-the-art performance when it comes to reasoning tasks[9]. This section presents our approach to using LLMs for solving TLR and ID tasks. Our approach involves building a prompt and having an LLM provide a response to that prompt. For our research, the prompt needs to be written in a specific way to get the desired response from the model. It must also be in the correct format to be parsed and evaluated automatically. To optimize TLR and ID performance using an LLM, we employ the methods outlined in section 2.5, in addition to experimentation.

4.1. Use Cases

This section describes three specific use cases for utilizing LLMs to solve TLR and ID tasks. We visualize the relationships between these use cases in Figure 4.1.

Traceability Link Recovery This use case describes TLR in isolation, i.e., without any supporting information other than the provided SAD and SAM. Given these artifacts, we expect the model to generate the corresponding traceability links that can be used for evaluation.

We explore several methods for prompting language models to solve TLR and ID tasks, including Zero-Shot Prompting, Few-Shot Prompting, and Chain-of-Thought Prompting.

In the case of Few-Shot Prompting, we first establish a few simple example projects with diverse, relevant scenarios for TLR. We construct a gold standard for the ideal model response for every example.

In the case of Chain-of-Thought Prompting, we use the same scenarios as in Few-Shot Prompting but construct new gold standards. These are comprised of the contents of the Few-Shot gold standards combined with a few rows of reasoning for why these traceability links should be established. An overview of the process of building a prompt for this use case before prompting the model is shown in Figure 4.2. The results are outlined in chapter 5.

End-to-End Inconsistency Detection This use case describes ID in isolation, i.e., without any supporting information other than the provided SAD and SAM. These artifacts are supplied, and the model generates a list of inconsistencies.

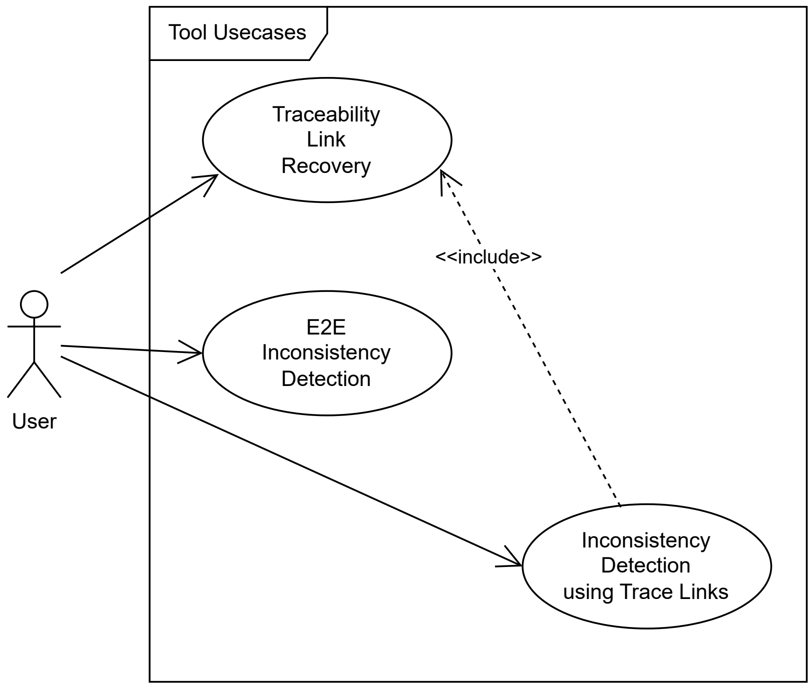

Tool Usecases
Traceability Link Recovery
<<include>>
E2E Inconsistency Detection
User
Inconsistency Detection using Trace Links

Figure 4.1.: Use Case Diagram Overview

We aim to detect Unmentioned Model Elements (UME) and Missing Model Elements (MME) (cf. section 2.2), which have been previously examined by other researchers using a different approach[27]. This research will serve as a baseline for our approach. To optimize and evaluate the performance of our approach, we treat UME and MME detection as separate use cases.

For both use cases, we create distinct system messages and prompts for each prompting method we employ. An overview of the process of building a prompt for this use case before prompting the model is shown in Figure 4.2.

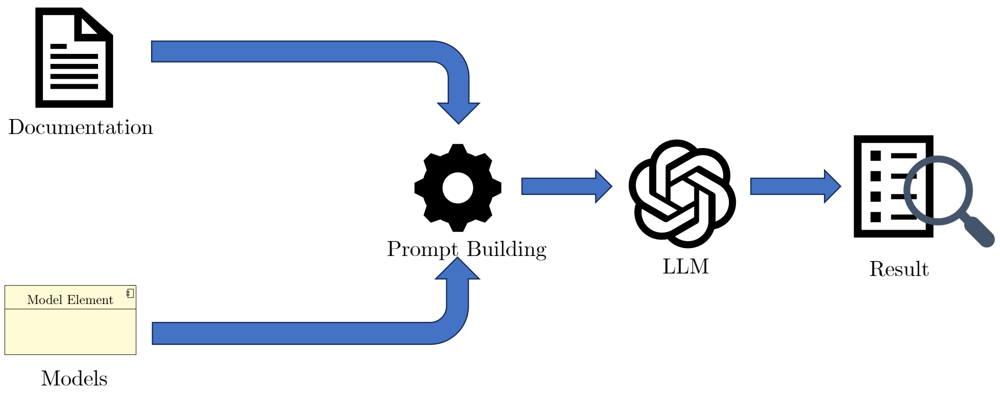


Figure 4.2.: An overview of the process of building a prompt using SAD and SAM before querying the model.

Inconsistency Detection using Traceability Links This use case describes ID when it uses traceability links as an additional source of information. Keim et al.[27] have shown that having traceability links available can help increase the performance of ID approaches. We examine the usefulness of traceability links in the prompt to see if we can observe the same effect of increased performance.

These traceability links can be generated by the model itself in the TLR use case or supplied, often from a TLR gold standard. In our work, we examine how ID can be improved by using ideal traceability links from a gold standard and using the output from the previous TLR use case as input for ID. An overview of the process of building a prompt using traceability links is shown in Figure 4.3.

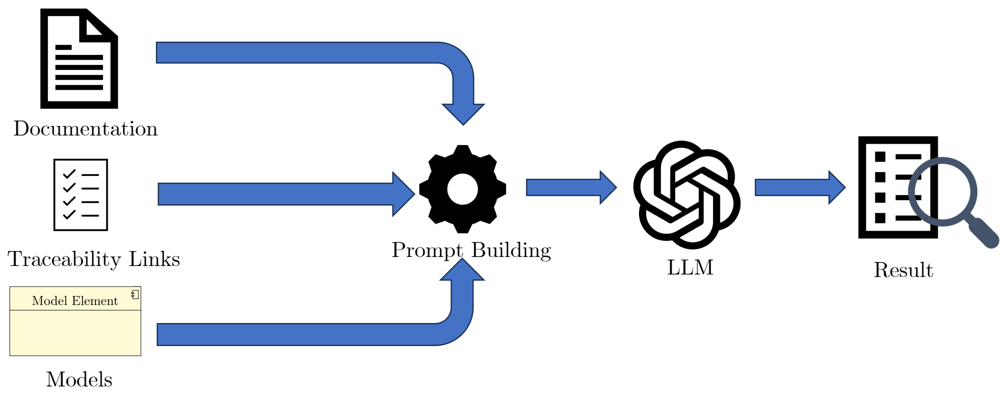


Figure 4.3.: An overview of the process of building a prompt using traceability links, as well as the SAD and SAM before querying the model.

4.2. Prompt Building

There are many ways to make large language models useful in the cases mentioned above. In this section, we outline the methods we used to build our prompts.

4.2.1. Few-Shot Examples

The idea of Few-Shot prompting is to provide a small set of Question-and-Answer examples that cover a variety of cases, including edge cases, in order to de-bias the model. Following these examples, we append the actual question for the model to answer. To achieve this goal while using as few tokens as possible, we created the following examples:

Example: "Santorini"

This example features an interface and inheritance from said interface.

Model See Figure 4.4.

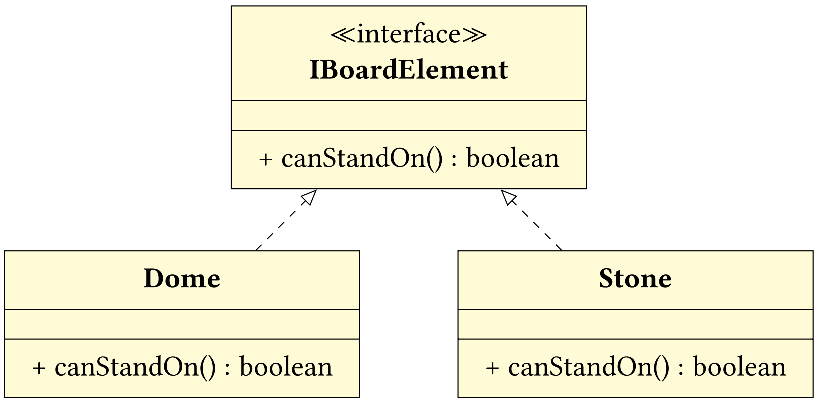


Figure 4.4.: Class Diagram of the "Santorini" example.

Documentation

SAD

The software models the gameplay of a real life game of Santorini.
In the model, a playing board is represented by a 2D array of IBoardElements.
An IBoardElement can be either a stone or a dome.
It is represented as an interface.
Every IBoardElement has the method "canStandOn" which represents whether a player can stand on that element.
The Dome class contains the attribute "height", which represents the height of this dome.
The database component gets interacted with by IBoardElements.

Example: "Utils"

This example features multiple classes with no relations between them.

Model See Figure 4.5

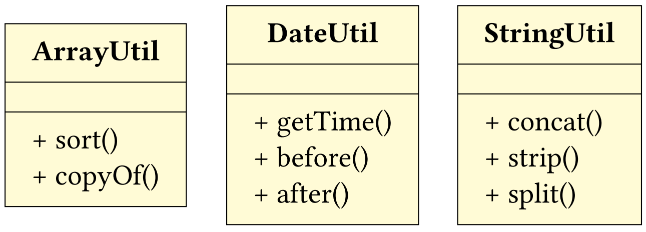


Figure 4.5.: Class Diagram of the "Utils" example.

Documentation

SAD

A number of utility classes are provided in the utils package.
The StringUtil class offers three methods for manipulating strings.
It offers the methods concat(), strip() and split().
The concat() method takes two Strings as input, the other methods only take one.
The DateUtil class offers a number of methods for manipulating dates.
The DateUtil class offers the methods getTime(), before() and after().
The third class offers a number of methods for manipulating arrays.
It offers the methods sort() and copyOf().
The class is called "ArrayUtil".

Example: "Command Pattern"

This example features the command design pattern. The relationships here are more complex, and the example is longer.

Model See Figure 4.6.

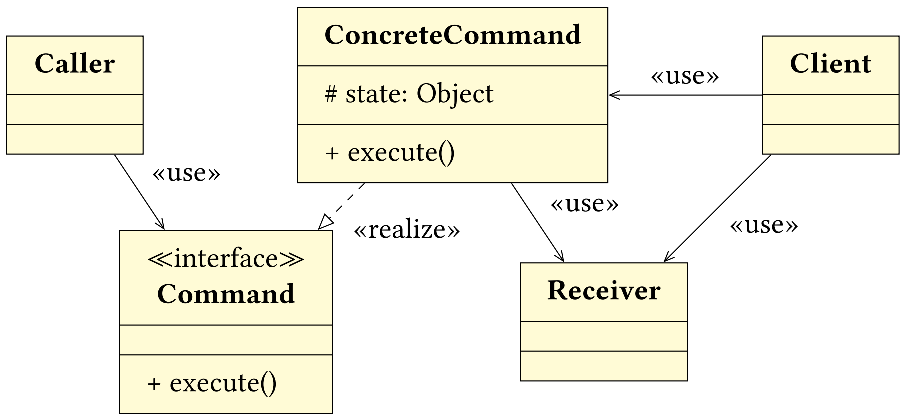


Figure 4.6.: Class Diagram of the "Command Pattern" example.

Documentation

SAD

The Command Interface serves as the contract that demands every realization (concrete command) to provide an execute() method.
This ensures that all commands can be executed uniformly regardless of their specific implementation.
The ConcreteCommand class represents a specific command in the system.
It implements the Command interface, which means it must provide the execute() method.
Additionally, the ConcreteCommand can have its own internal state, allowing it to store information related to the specific action it represents.
Its state can be used to modify the behavior or parameters of the execution process.
The Receiver class is responsible for executing the actual actions associated with commands.
It is not directly part of the design pattern, but it collaborates with the ConcreteCommand objects.
The Receiver class knows how to perform the desired action and can be configured and associated with different ConcreteCommand objects to carry out specific tasks.
The Client class acts as the entry point of the software project.
It creates and configures the necessary objects to utilize the design pattern effectively.
The Client is responsible for creating instances of ConcreteCommand and Receiver, associating them appropriately, and setting the ConcreteCommand in the Caller (Invoker).
By doing so, the Client triggers the execution of commands through the Caller, ultimately invoking the Receiver to perform the desired actions.

4.2.2. Vector Retrieval Pre-Processing

Apart from directly prompting the model with the entire SAD, as described above (cf. section 4.2), we also want to examine the effects of using only the most relevant parts of the SAD for a specific model element instead of the entire text to build the prompt. The idea is that the model can then generate traceability links for this model element more effectively. To achieve this, we use a vector retrieval system to retrieve the sentences of the SAD most relevant for this model element. We then use these parts to generate a prompt for the model. This prompt then tasks the model with finding traceability links in the supplied SAD segments. This can be achieved by first employing a Vector Store to retrieve the most relevant sections for a given model element and then prompting the model using the now filtered version of the SAD. For every sentence in a given SAD, we generate embeddings. The vectors for each sentence are stored in a vector database. An overview of the process of building a prompt before prompting the model is shown in Figure 4.7.

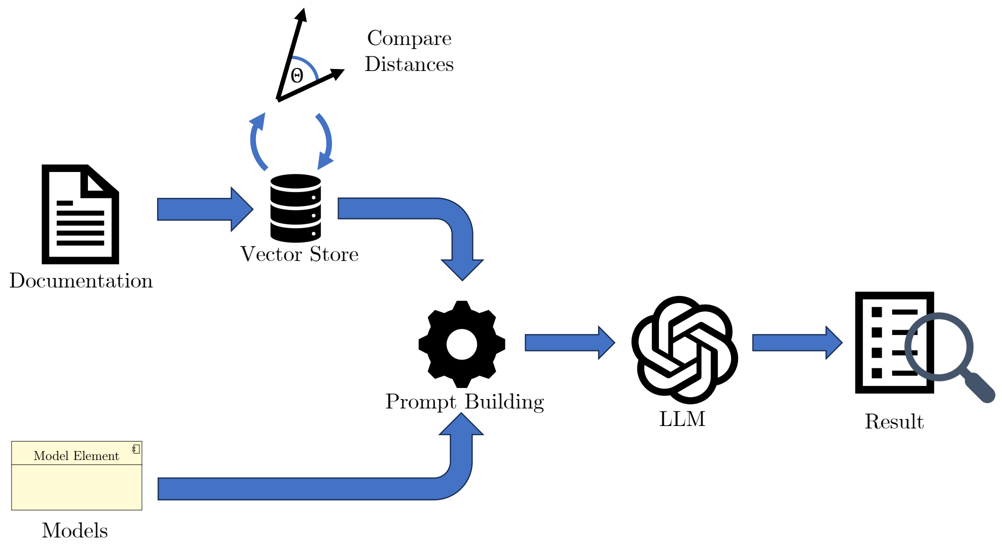


Figure 4.7.: An overview of the process of building a prompt, using a Vector Store for pre-processing of the SAD, as well as the unchanged SAM, before querying the model.

Retrieving most relevant sentences We use the names q of the different model elements as queries for the vector database, which then returns the most relevant sentences d according to a vector distance function. The function we employ is the cosine similarity, which is defined as follows:

$$\text{similarity}(d, q) = \frac{\sum_{i=1}^{n} d_i \cdot q_i}{\sqrt{\sum_{i=1}^{n} {d_i}^2} \cdot \sqrt{\sum_{i=1}^{n} q_i^2}} \tag{4.1}$$

where d_i is the i-th dimension of the document vector and q_i is the i-th dimension of the query vector. Using this method, we can retrieve sentences along with a measure of how relevant they are to the query model element.

We use a dynamic threshold value to retrieve only the most relevant sentences. To do so, we first retrieve all sentences d and calculate their similarity to the query model element q. We then sort them according to their similarity, resulting in a collection $d'_1, d'_2, \ldots, d'_n$, with d'_1 being the most similar sentence to q and d'_n being the least similar. Our threshold value t is then defined as follows:

$$t = similarity(d'_1, q) - (similarity(d'_1, q) - similarity(d'_n, q)) \cdot c \tag{4.2}$$

With c being a constant value between 0 and 1. We then retrieve all sentences d with $similarity(d, q) \geq t$. This process is further illustrated in section 2.4.

Generating traceability links Using this approach, it is possible to generate traceability links. We retrieve the most relevant sentences for a given model element according to the threshold. We denote the respective line numbers as traceability links for that model element. We then repeat this process for every model element in the SAD and combine the responses to form a complete list of traceability links.

Prompt Generation To use these most relevant documents for just one model element, we adapt the prompt to try and find traceability links for just this one model element. We do this by generating a list of the most relevant sentences that meet our threshold t. We insert the list of sentences into the prompt, along with a line number for each. The prompt describes how these sentences are ordered by descending relevance. We then ask the model to find traceability links in these sentences. To generate traceability links for the entire project, we repeat this process for every model element in the SAM and combine the responses to form a complete list of traceability links.

4.3. Prompt Engineering

Prompt engineering is another central piece of making LLMs work for our use cases.

We engineered these prompts using OpenAI's GPT-3.5 and GPT-4, when available, as these models are deemed state-of-the-art [9]. Once a good prompt was found, we examined different software architecture projects using the same prompt.

We examine the effects of various prompt engineering methods, including Zero-Shot Prompting, Few-Shot Prompting, 1-Shot Prompting, Chain-of-Thought Prompting, and Prompt-Chaining. A specific focus is placed on analyzing the impact of these methods on TLR and ID for both MMEs and UMEs, with the goal of achieving higher accuracy in the results.

Reduction of Hallucinations

Hallucination in generative language models can be divided into two categories[24]:

- *Intrinsic Hallucinations* The generated output directly contradicts information given in the prompt or during training.
- *Extrinsic Hallucinations* The generated output cannot be verified using the information given in the prompt or during training. It expands upon the information available without any proof or disproof available.

Keeping hallucinations small and infrequent is imperative for achieving good TLR and ID scores, as any hallucination would generally be immediately reflected in the performance of our model.

There are different techniques for mitigating these hallucinations, which differ depending on the use case. Ji et al. have defined a set of different task categories for hallucinations[24]. For TLR and ID, we believe the following categories to be relevant:

- TLR: Abstractive Summarization, GQA, Task-Oriented Dialogue
- ID: GQA, Data2Text, Task-Oriented Dialogue

In their[24] document, they detail mitigation methods for every category.

Other methods include lowering the *temperature* value in a GPT model, which should make the behavior more deterministic and less random, therefore also reducing the number of hallucinations. Another option is careful prompting methods, which present and overlap with section 4.3.

In our experiments, we were able to observe hallucinations mainly in the form of intrinsic hallucinations. These could often be mitigated by adapted prompt instructions. That includes concrete examples during Few-Shot prompting and explicit hints in the system message and question prompt. One issue we encountered was during traceability link generation, where the response points to line numbers that do not exist within the respective SAD. A frequency penalty could mitigate this, but we found that adding the SAD length to the prompt was a more effective method.

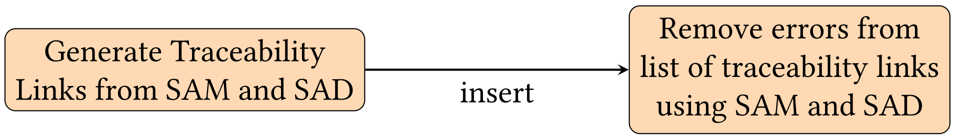
Embeddings

Embeddings are a type of vector space model (VSM), as described in section 3.4. They are used to represent the meaning of a word or several words relative to others in a vector space, which allows for semantic search: Vector databases have the unique ability to return the nearest neighbors (according to a distance function, like the cosine similarity, the Jaccard similarity, or the correlation coefficient[2]) to a search vector, which then represents the entries semantically closest to the search input. In our approach, we take every sentence/line of our SAD as a document to embed in our vector database. We then use the names of the different model elements as queries for the vector database. The returned sentences represent the most relevant sentences for the respective model element. We then explore two options:

- The returned documents represent the traceability links, which we then evaluate.
- The returned documents are used to generate a prompt for the model, which is then tasked with finding traceability links in the supplied SAD segments. The model response is then evaluated.

Prompt-Chaining

Brief experimentation with Prompt-Chaining was also made:


Generate Traceability Links from SAM and SAD
insert
Remove errors from list of traceability links using SAM and SAD

These experiments yielded no significant results. The responses generated often maintained the same accuracy as the original input. Thus, we did not pursue this method further.

5. Evaluation

In this chapter, we describe how we evaluate the performance of our approach. We first describe the benchmark dataset we use to evaluate our approach. We then look at the results in our different use cases (cf. chapter 4) and the metrics we employ for evaluating them. We examine result values and outline the most significant problems we encounter.

We use the benchmark constructed by Fuchß et al.[18] to evaluate the performance. The benchmark includes a set of 5 different projects, each with a different level of detail to their SAD and SAM. The projects are the following: MediaStore, TEAMMATES, BigBlueButton, TeaStore, and JabRef. For each project, the benchmark includes SAD files, SAM files in different formats, including UML, and gold standard files for TLR and UME-ID. Table 5.1 shows how these projects' SAM and SAD differ in size. For some projects, their large SAD size significantly affects prompt length, like in the case of TEAMMATES. A gold standard for MME performance is unnecessary because of how ArDoCo and we examine MMEs, as we explain in subsection 5.2.3.

API costs

Due to the nature of our experiments, we sometimes only conduct limited experiments, trying to save on API calls to OpenAI. Because Self-Consistency and Tree-of-Thoughts prompting require multiple queries to the API to arrive at a result, we do not consider these methods in this paper. Instead, we focus on Zero-Shot, Few-Shot, and Chain-of-Thought prompting.

Metrics

We use the same metrics to evaluate the performance of our approaches for the different use cases. We consider the same metrics for traceability links, unmentioned model elements, and missing model elements. In the following description, these items will be referred to as "solution elements". Given a gold standard G of expected solution elements, we consider the following metrics for our evaluation:

- Precision: $\frac{TP}{TP+FP}$ with TP as the set of solution elements that were correctly identified and FP as the set of solution elements that were incorrectly identified, i.e., those solution elements that were detected but do not exist within the gold standard ($TP \in G, FP \notin G$).
- Recall: $\frac{TP}{TP+FN}$ with TP as the set of solution elements that were correctly identified as present and FN as the set of solution elements were not found ($TP, FN \in G$).

- F_1: $\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision}+\text{recall}}$ is the harmonic mean of precision and recall, which takes both metrics into account and provides a single score that balances them.

Table 5.1.: SAD and SAM information for different projects

Project	SAD		SAM	
	Words	Sentences	Components	Interfaces
TeaStore	661	43	11	8
BigBlueButton	1190	85	12	12
MediaStore	572	37	14	9
JabRef	237	13	6	0
TEAMMATES	2509	198	8	8

5.1. Traceability Link Recovery

The goal for TLR in isolation is to retrieve traceability links from the provided artifacts. The result generated by our approach can be compared to the gold standard, using the benchmark[18]. For this, we want to examine three different approaches: Firstly, we try to prompt the LLM directly with the entire available SAD, without any preprocessing. Then, we look at Vector Retrieval approaches, where we directly use a Vector Store to generate traceability links. Following that, we try to take the output from the Vector Retrieval approach and use it to prompt the LLM, replacing the SAD.

5.1.1. LLM-only Approach

We evaluate performance when generating traceability links when querying the LLM with the provided SAM and SAD. There is no preprocessing step; the SAD and SAM are not filtered.

Question

Does our approach of prompting LLMs without major pre-processing outperform the ArDoCo approach[27] in terms of TLR performance?

Results

When utilizing the benchmark developed by Fuchß et al. [18], the results, as depicted in Figure 5.1, provide valuable insights. We compare the approaches of plainly prompting the model using the entire SAD and SAM with the different prompting techniques, Zero-Shot, Few-Shot, and Chain-of-Thought prompting. We observe the Zero-Shot performance of this approach to be the worst out of all three methods considered. Zero-Shot prompting yields a precision of 34%, a recall of 38%, and an F_1 score of 35%. We also find that Few-Shot prompting rarely improves performance for this kind of analytical work. However, in some cases, it can have the opposite effect and harm the model's TLR capabilities. It achieves a precision of 38%, a recall of 60%, and an F_1 score of 43%. However, Chain-of-Thought prompting yields a noticeable performance increase over the Zero-Shot approach. It achieves a precision of 56%, a recall of 56%, and an F_1 score of 52%.

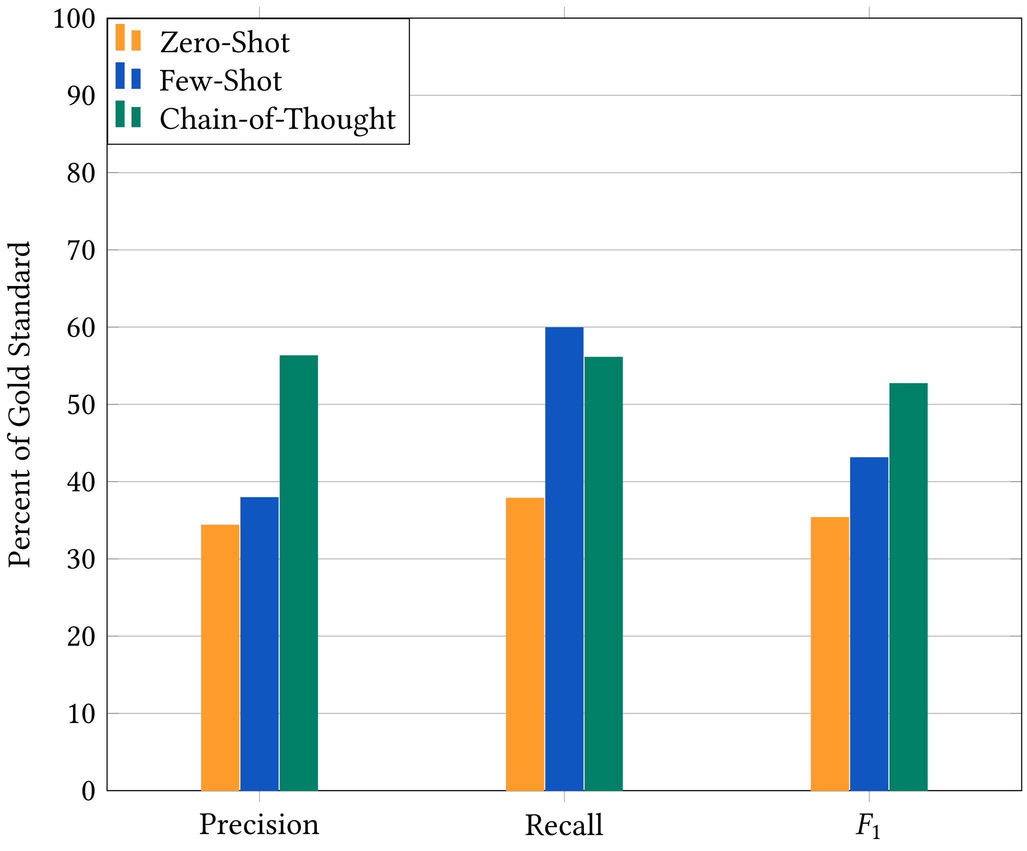


Figure 5.1.: Performance metrics weighted average on Benchmark[18] for TLR using GPT-4. In percent of gold standard.

Few-Shot prompting is a technique used to improve the performance of LLMs on various tasks[8]. However, it seems that when applying LLMs to the task of TLR, the performance is not improved by using Few-Shot prompting. In some cases, the performance even decreases. Overall, we believe it is necessary to consider the stark differences in performance between the different analyzed projects within the benchmark dataset. The performance seems to rise with projects that have a more detailed SAD available, as can be seen in Table 5.3.

Vector Retrieval Approach

We first try to use a VSM to generate traceability links before improving the result using Large Language Models, as described in paragraph 4.2.2.

Question

Does our approach utilizing VSMs for preprocessing outperform the ArDoCo approach[27] in terms of TLR performance?

Results

When using the vector store approach for TLR, we can observe a relation between the threshold coefficient and Precision and Recall. By adapting the value of our coefficient, we can find an adequate tradeoff between Precision and Recall. As shown in Figure 5.2, the higher the threshold coefficient, the higher the Precision and the lower the Recall, and vice versa. The F_1 measure peaks at a threshold coefficient value of $c = 0.75$ with a value of 51%.

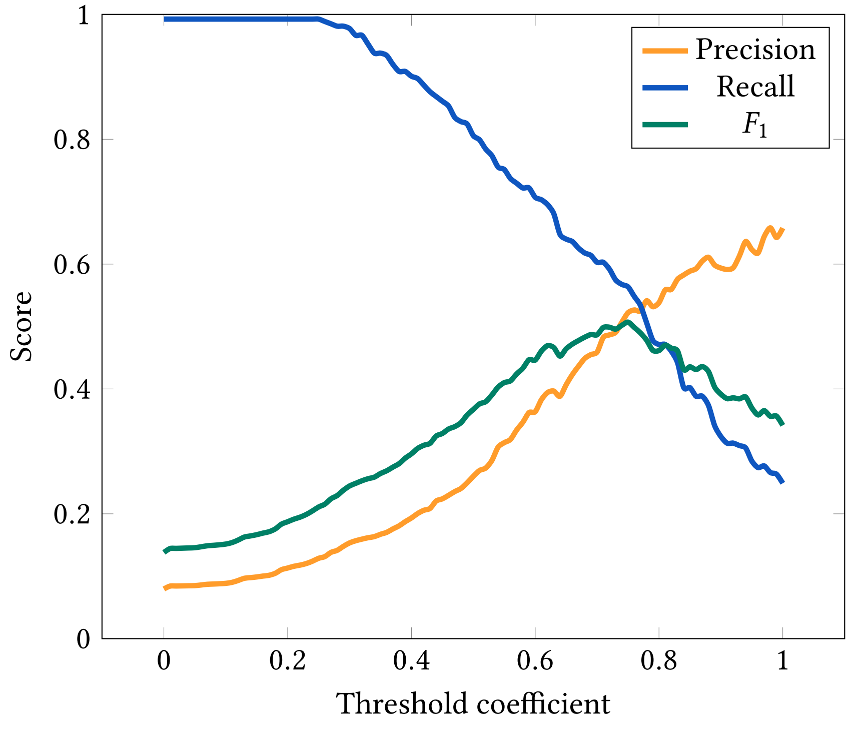


Figure 5.2.: Embedding Approach Performance when using different threshold coefficients.

When considering individual projects from the benchmark, the results look similar: The following table is optimized by F_1-performance.

5.1.2. VSM pre-processing with LLM

Instead of plain vector retrieval, we can also use the vector store approach to pre-process the SAD and then prompt the LLM with the retrieved sections of the SAD and the unchanged SAM.

Task	TS	BBB	MS	JR	TM
P	.37	.71	.34	.70	.59
R	.50	.38	.77	.74	.45
F1	.42	.49	.47	.72	.51
optimal c	.83	.75	.75	.61	.78

Table 5.2.: The Vector Retrieval Approach, with optimal threshold coefficient c values for each project.

We now want to find a new threshold coefficient c that optimizes the F_1 score. This is because we want to focus on a high Recall, as we want to cover as many relevant sections as possible. The intention is that the LLM will be able to filter out some false positives, thus increasing the precision. We conducted limited testing using the pre-processed SAD, we try to find the maximum F_1 value while minimizing the number of requests to the LLM. Using a "binary-search" approach, we found that a threshold coefficient of $c = 0.34$ provided the best results.

Question

Does our approach of using a Vector Store for pre-processing and subsequently using an LLM to generate traceability links outperform the ArDoCo approach[27] in terms of TLR performance?

Results

Figure 5.3 presents the results for different prompting methods. The reference pillar for the vector retrieval methods uses a threshold coefficient of $c = 0.75$. The values for this method form our baseline: a precision of 48%, a recall of 55%, and an F_1 score of 48%. We compare the Zero-Shot, Few-Shot, and Chain-of-Thought prompting methods. The Zero-Shot approach sees decreased precision of 44% compared to our baseline, but at the same time reaches a higher F_1 score and Recall, which improves by 30% to 85%. This is mainly due to the larger threshold coefficient, which increases the number of retrieved sections. The precision suffers a slight loss of 4% compared to the baseline vector retrieval. Zero-Shot prompting thus achieves the goal of roughly maintaining precision while allowing a higher recall value. Few-Shot prompting, however, achieves 34% precision, a loss of 14% compared to the baseline. The recall is 90%, a gain of 35% compared to the baseline. The F_1 score is 47%, a loss of 1% compared to the baseline. Few-Shot prompting thus achieves a higher recall, but at the cost of a significant loss in precision. Chain-of-Thought prompting achieves 52% precision, a gain of 4% compared to the baseline. The recall value is 90%, an increase of 35% compared to the baseline. The F_1 score is 64%, a gain of 16% compared to the baseline. This value is the highest among all considered prompting methods. For comparison between approaches, we also include the results from our best-performing LLM-only approach, which uses Chain-of-Thought prompting. This approach achieves a precision of 56%, a recall of 56%, and an F_1 score of 52%. We conclude that the VSM pre-processing approach outperforms the LLM-only approach in Recall and F_1. In terms of Precision, the LLM-only approach achieves a better result.

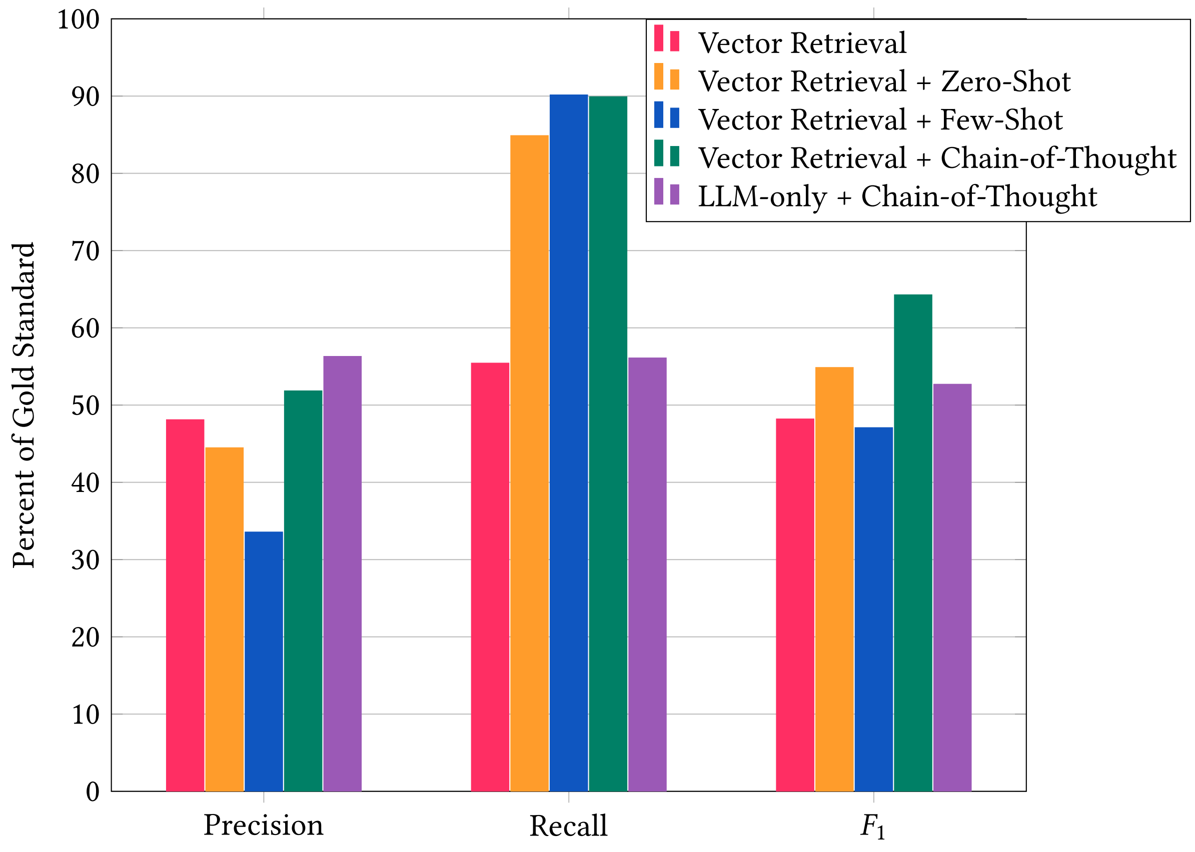


Figure 5.3.: Performance metrics weighted average on Benchmark[18] for TLR Vector store approaches compared to best LLM-only performance. In percent of gold standard.

Comparison to ArDoCo

We compare our approaches, which use the LLM, the Vector Storage approach, or a combination of both, to the ArDoCo[27] approach. The results of this comparison are presented in Table 5.3, as well as in Figure 5.4 for an overview of weighted averages of our best approaches.

Our approach can outperform the ArDoCo approach in the *Recall* metric in some projects, as shown by the results. However, it is not reliable enough to produce consistent results across the benchmark. Overall, our approach shows some success in specific scenarios.

The stark performance differences between different projects within the benchmark datasets show that a moderately good performance, like with the MediaStore project, can only be achieved if the necessary level of detail within the SAD is available. Here, Few-Shot Prompting even hurts the performance when done without encouraging Chain-of-Thought.

Task	TS			BBB			MS			JR			TM			w. Average		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
ArDoCo	**1.0**	.74	**.85**	**.88**	.83	**.85**	**1.0**	.62	**.77**	**.90**	**1.0**	**.95**	**.56**	**.90**	**.69**	**.83**	.82	**.80**
GPT-4 CoT	.50	**.92**	.65	.32	.22	.21	.80	.66	.72	.88	.88	.87	.32	.13	.18	.56	.56	.53
Vector Retrieval	.28	.56	.37	.68	.37	.48	.33	.76	.46	.65	.61	.63	.47	.48	.48	.48	.55	.48
VR + GPT-4 CoT	.49	.89	.63	.51	**.85**	.64	.30	**.90**	.45	.86	**1.0**	.92	.43	.86	.57	.52	**.90**	.64

Table 5.3.: Comparison of performance metrics for ArDoCo and our approaches: Prompting GPT-4 using Chain-of-Thought prompting, Vector Retrieval (VR) from a Vector Database, supplying Vector Retrieval results to GPT-4 and using Chain-of-Thought prompting to improve results.
Bold: Best result per metric.

Part of the performance difference between the vector-based approach using Chain-of-Thought versus the more straightforward approach of querying the LLM with the entire document and using Chain-of-Thought is the fact that the LLM cannot process the whole document at once. The context length of the used model, GPT-4, is limited to 8192 tokens. This means that the LLM cannot process the entire document at once. This is especially prevalent when using Chain-of-Thought prompting, which drastically increases prompt and response length. Particularly in the case of the Teammates project, the response is often cut off by the context length, which is shared between prompt and response. In the case of Chain-of-thought prompting, the model would provide reasoning steps for the large number of lines present in the SAD. However, the model would run out of context length before generating the evaluated part of the response, the actual traceability links. In the vector-based approach, however, we only supply the most relevant sections to the model. In the case of the Teammates project, tens of lines have to be considered rather than hundreds. The context length, in this case, is no longer a limitation.

5.2. Inconsistency Detection

The section on ID is divided into two parts: End-to-End ID and ID with supporting traceability links. We again construct different system messages and prompts for all considered prompting methods for each part.

5.2.1. End-to-End Inconsistency Detection (UME)

For this task, we prompt the model with the SAD and SAM and expect the model to find inconsistencies. This differs from the ArDoCo[27] approach, which uses traceability links to aid the model in finding inconsistencies. The LLM should handle the work required for finding traceability links and naming inconsistencies in a single step, i.e., a single prompt is used that will produce only one model response. This differs from prompt chaining.

Question

Does our approach utilizing LLMs without traceability links outperform the ArDoCo approach[27] in terms of UME performance?

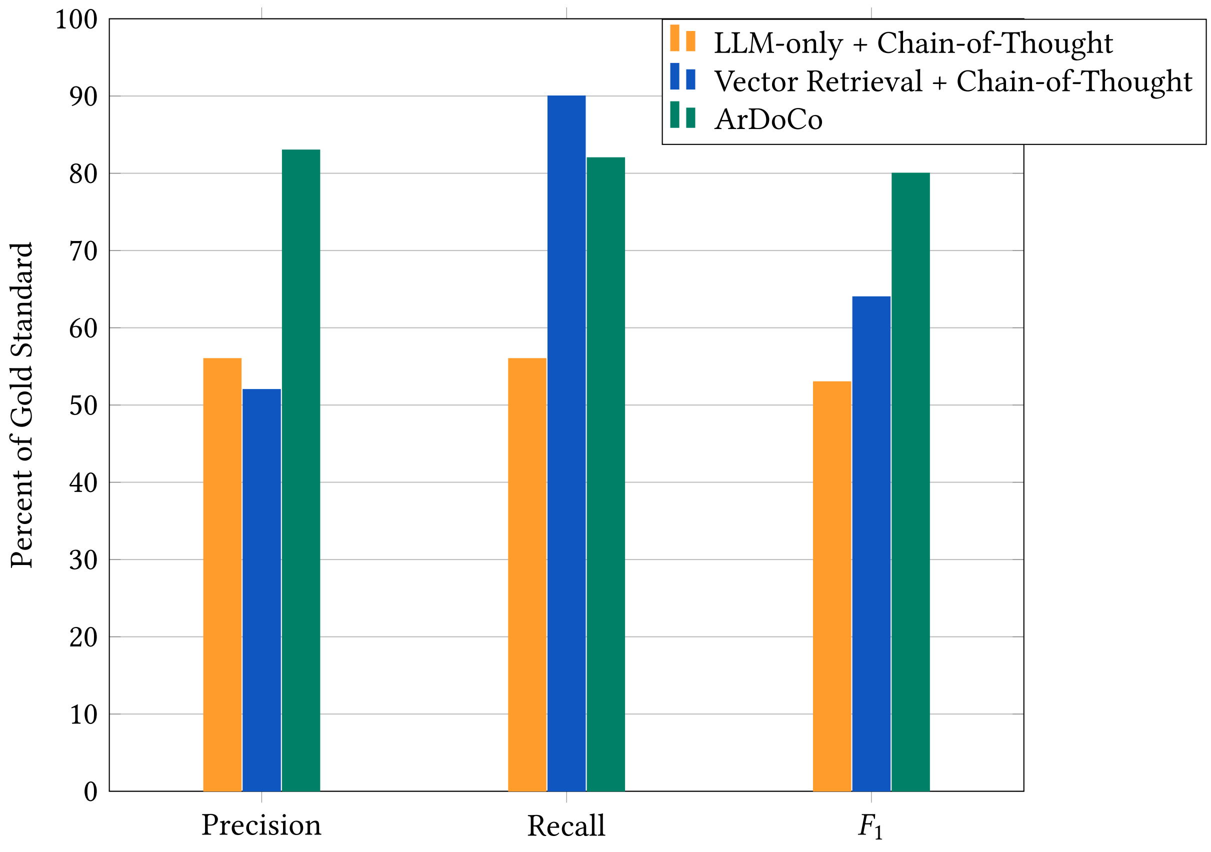


Figure 5.4.: Performance metrics weighted average on Benchmark[18] for our TLR approaches compared to ArDoCo's TLR performance. In percent of gold standard.

Results

The results for UME-ID using the benchmark by Fuchß et al.[18] are presented in Figure 5.5. These results show a different trend than the TLR results. For this round of evaluation, do not consider the TEAMMATES project, as the gold standard for this project does not contain any unmentioned model elements. While our approach, like ArDoCo[27, p. 148], correctly identifies no inconsistencies within this project, we cannot evaluate this case using the metrics we have chosen. We thus exclude this project from our evaluation. Zero-Shot prompting achieves a Precision of 77% and a Recall of 97%, resulting in an F_1 score of 80%. Few-Shot prompting achieves a Precision of 71% and a Recall of 78%, resulting in an F_1 score of 73%. Chain-of-Thought prompting achieves a Precision of 65% and a Recall of 74%, resulting in an F_1 score of 69%.

Part of the performance drop can be attributed to limitations in model size, as responses were often cut off due to the maximum length of 8192 tokens of GPT-4. This is especially prevalent when using Chain-of-Thought prompting, which drastically increases prompt and response length.

Moreover, the results again support the hypothesis that Few-Shot prompting methods are inadequate for analytical tasks. Zero-Shot prompting yields the best results.

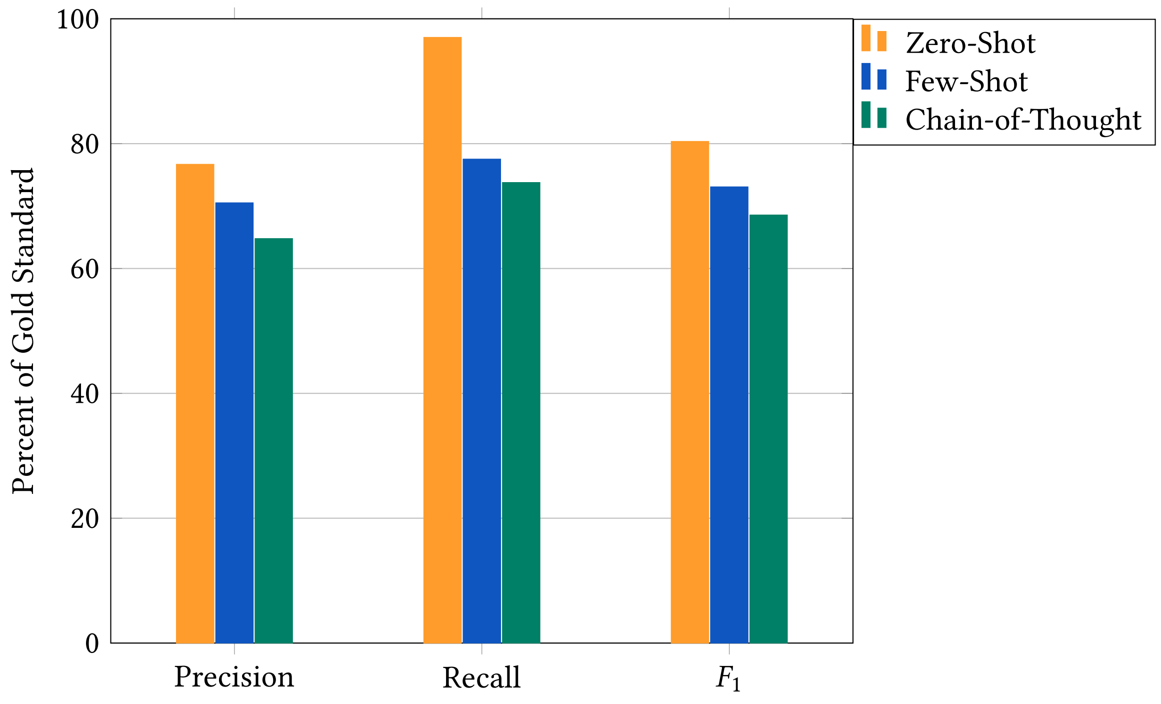


Figure 5.5.: Performance metrics weighted average on Benchmark[18] for UME-ID using GPT-4. In percent of gold standard.

5.2.2. Inconsistency Detection using Traceability Links (UME)

Similar to the ArDoCo approach, we evaluate whether supplying traceability links to the model aids performance for UME ID.

Question

Does our approach utilizing LLMs with the support of traceability links outperform the ArDoCo approach[27] in terms of UME performance?

Results

When using Zero-Shot prompting, the model achieves a precision of 91% and a recall of 98%. The F_1 score is 93%. The model performs a perfectly correct prediction across our benchmark when using Few-Shot prompting. The precision, recall, and F_1 score are all 100%. When using Chain-of-Thought prompting, the model achieves a precision of 89% and a recall of 96%. The F_1 score is 91%. This goes against the behavior in the previous method, where we did not use traceability links: Where the previous method showed a clear trend of Zero-Shot prompting outperforming Few-Shot prompting, the results here are almost identical across all prompting methods and strongest in the case of Few-Shot prompting. However, when using imperfect traceability links generated using the Vector Retrieval approach with Chain-of-Thought prompting, the performance drops below the

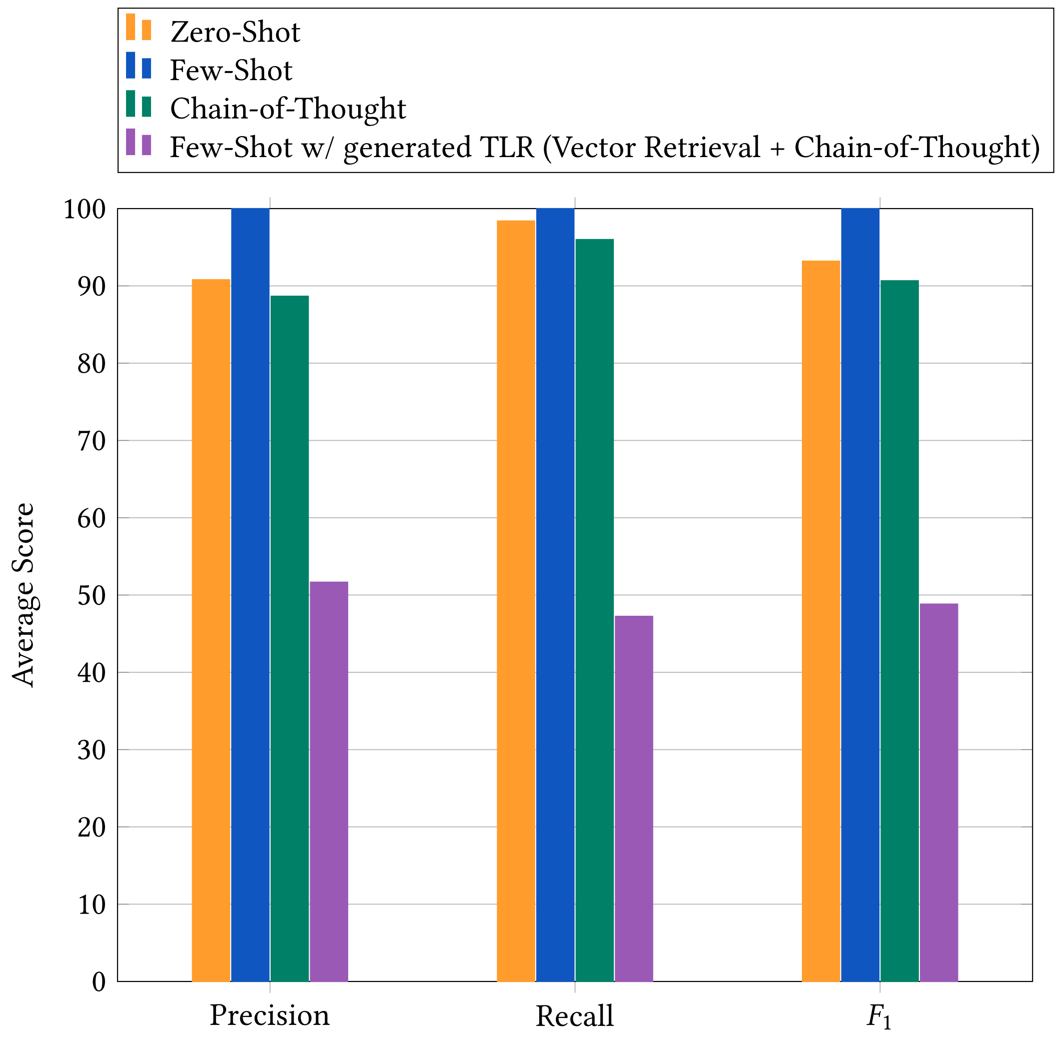


Figure 5.6.: Performance metrics weighted average on Benchmark[18] for UME-ID using GPT-4 and ideal/generated traceability links. In percent of gold standard.

performance of the non-TLR method. We achieve a precision of 52%, a recall of 47%, and an F_1 score of 49%. The task of UME-ID becomes trivial for humans when using traceability links and an SAM. In this case, GPT-4 is also able to achieve perfect performance. However, when using imperfect traceability links, the performance drops below the performance of the non-TLR approach.

Comparison to ArDoCo (UME)

Our performance is compared to the ArDoCo approach, and the results are presented in Table 5.4, as well as in Figure 5.7 for an overview of weighted averages of our best approaches.

While our results are similar to the ArDoCo performance, a notable gap exists between our approach and theirs. Once again, we see a wide variety of results, with some projects achieving perfect Precision results while others perform poorly. Moreover, we observe

Task	TS			BBB			MS			JR			w. Average		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
ArDoCo	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	.67	**1.0**	.80	**1.0**	**1.0**	**1.0**	**.88**	**1.0**	**.93**
GPT-4 Zero-Shot	**1.0**	.88	.93	.28	**1.0**	.40	**.79**	**1.0**	**.88**	**1.0**	**1.0**	**1.0**	.77	.97	.80
GPT-4 Few-Shot w/ ideal TLR	***1.0***	***1.0***	***1.0***	***1.0***	***1.0***	***1.0***	***1.0***	***1.0***	***1.0***	***1.0***	***1.0***	***1.0***	***1.0***	***1.0***	***1.0***
GPT-4 Few-Shot w/ VECTOR-CoT	.67	.64	.64	0.0	0.0	0.0	.60	.45	.51	.80	.80	.80	.52	.47	.49

Table 5.4.: Comparison of performance metrics for ArDoCo and our approaches: The first approach uses GPT-4 with Zero-Shot prompting, the second approach uses GPT-4 with Few-Shot prompting and optimal traceability links (taken from the benchmark), and the third approach uses GPT-4 with Few-Shot prompting and traceability links generated using the Vector-TLR with validation approach. **Bold:** Best result per metric.

how the ArDoCo approach manages to be more consistent while our approach struggles to produce reliable results. Using traceability links has given us some stability, especially remarkably in the case of BigBlueButton. However, the performance is still far from perfect. Further, we must note that these traceability links are ideal and taken from the gold standard.

5.2.3. End-to-End Inconsistency Detection (MME)

Like we did before (cf. subsection 5.2.1), we examine the performance when jumping right to ID (MME). Because no benchmark exists, we employ the same technique as *ArDoCo*[27] and dynamically generate a benchmark: This technique involves removing one model element at a time from the SAM. After removing each element, the model tries to identify which element was removed based on the modified SAM and the original SAD. This response is then compared to the gold standard, consisting of the exact element removed earlier. After each iteration, the removed element is added back to the SAM, and the next element is removed. This process continues until each element in the SAM has been removed once. By forming an average over all iterations of this process, we can evaluate the model's performance in identifying MMEs for this particular SAD and SAM.

1. Remove a model element from the SAM.
2. Use the modified SAM and the original SAD to prompt the model.
3. Use the name of the removed element as the gold standard.

Because we follow this same technique employed by Keim et al.[27], our results can be directly compared.

Question

Does our approach utilizing LLMs outperform the ArDoCo approach[27] in terms of MME performance?

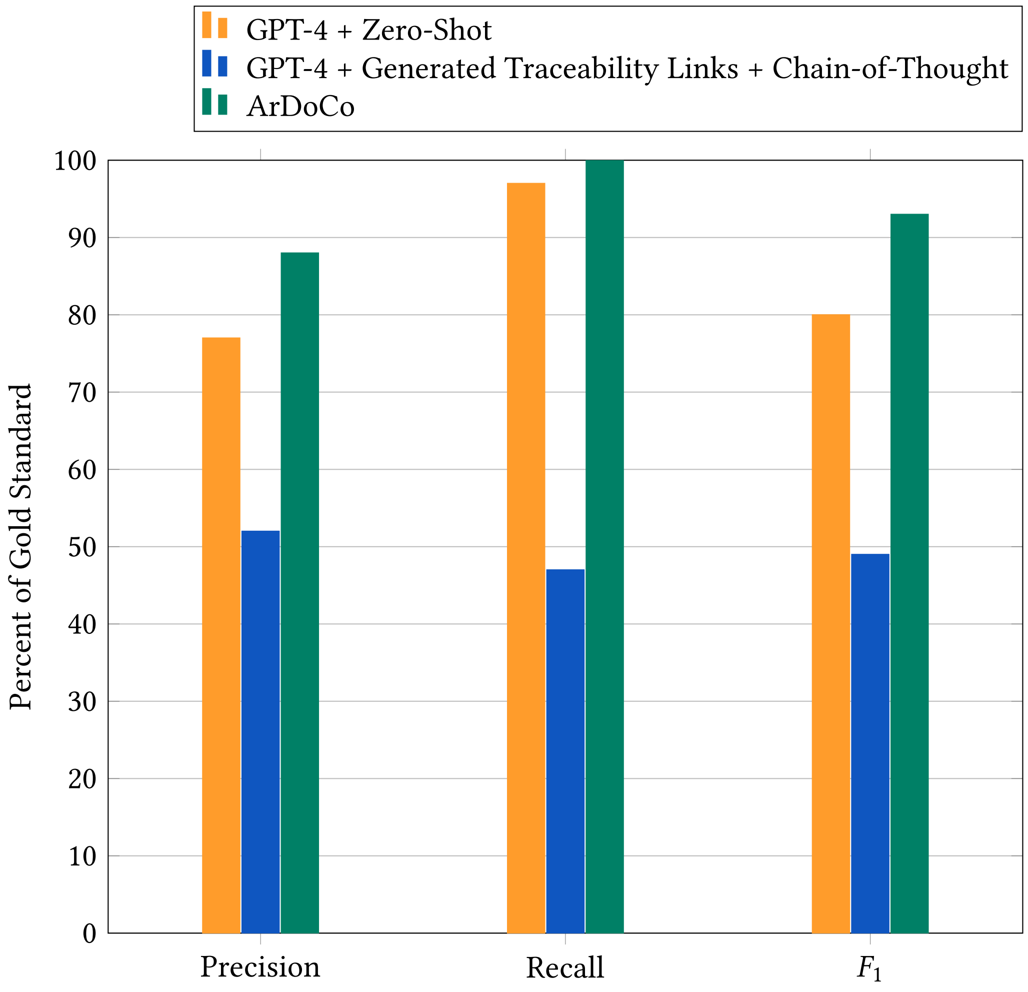


Figure 5.7.: Performance metrics weighted average on Benchmark[18] for our UME approaches compared ArDoCo's UME performance. In percent of gold standard.

Results

Chain-of-Thought prompting is most effective for Recall, and Few-Shot prompting is most effective for Precision. However, the F_1 score favors Few-Shot prompting, as section 5.2.3 shows. Zero-Shot prompting yields a Precision of 21% and a Recall of 43%, resulting in an F_1 score of 24%. Few-Shot prompting performs better and produces a Precision of 39% and a Recall of 51%, resulting in an F_1 score of 42%. Finally, Chain-of-Thought prompting achieves a Precision of 37% and a Recall of 54%, resulting in an F_1 score of 40%. This means that Chain-of-Though prompting has a higher Recall than the other methods but at the cost of lower Precision than Few-Shot prompting. Overall, Few-Shot and Chain-of-Thought performance differ only slightly.

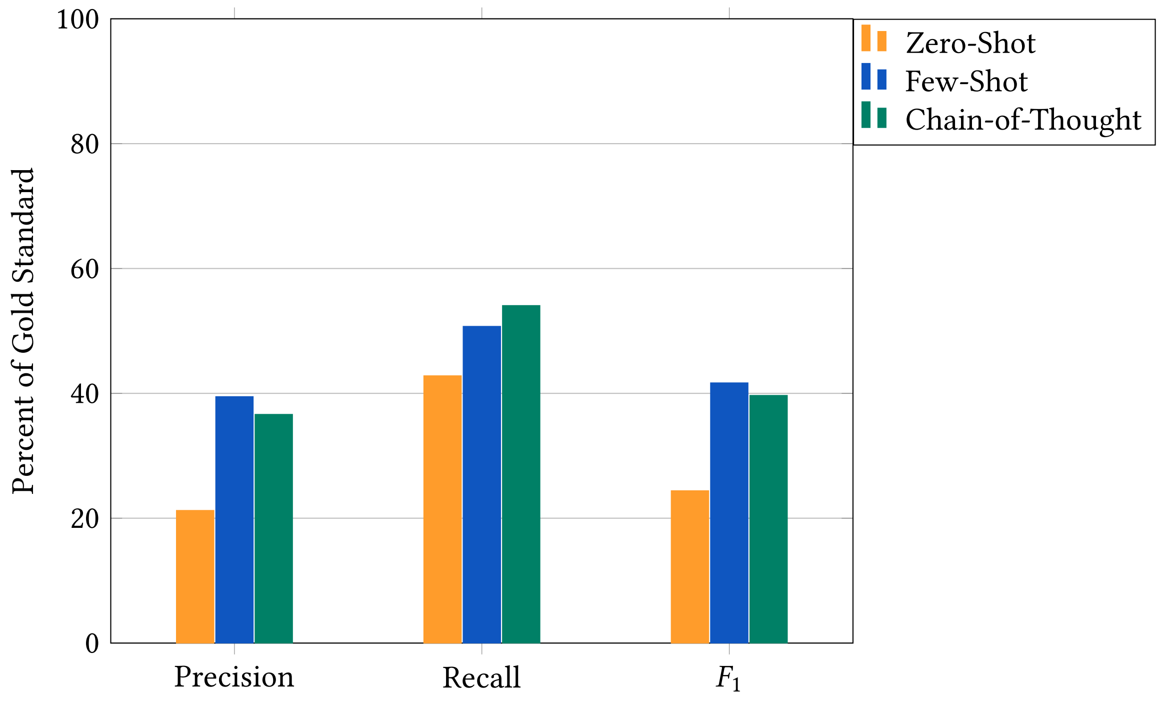


Figure 5.8.: Performance metrics weighted average on Benchmark[18] for MME-ID using GPT-4. In percent of gold standard.

5.2.4. Inconsistency Detection using Traceability Links (MME)

Like in subsection 5.2.2, we again evaluate how MME performance is impacted when using traceability links as an aid.

Question

Can our approach, which utilizes LLMs with the support of traceability links, achieve better performance than the ArDoCo approach[27] in identifying MMEs?

Results

section 5.2.4 shows how the performance changes when using traceability links. The performance drops compared to not using traceability links across all metrics and all prompting methods. Zero-Shot prompting achieves a Precision of 14% and a Recall of 38%, resulting in an F_1 score of 17%. This approach has the lowest Precision and Recall among the three methods. Few-Shot prompting achieves a higher Precision of 26% and a higher Recall of 48%, resulting in an F_1 score of 30%. This approach strikes a better balance between Precision and Recall. Chain-of-Thought prompting achieves a Precision of 14% and a Recall of 38%, resulting in an F_1 score of 17%. This approach has a similar Precision and Recall to the Zero-Shot prompting approach. Because using ideal traceability links

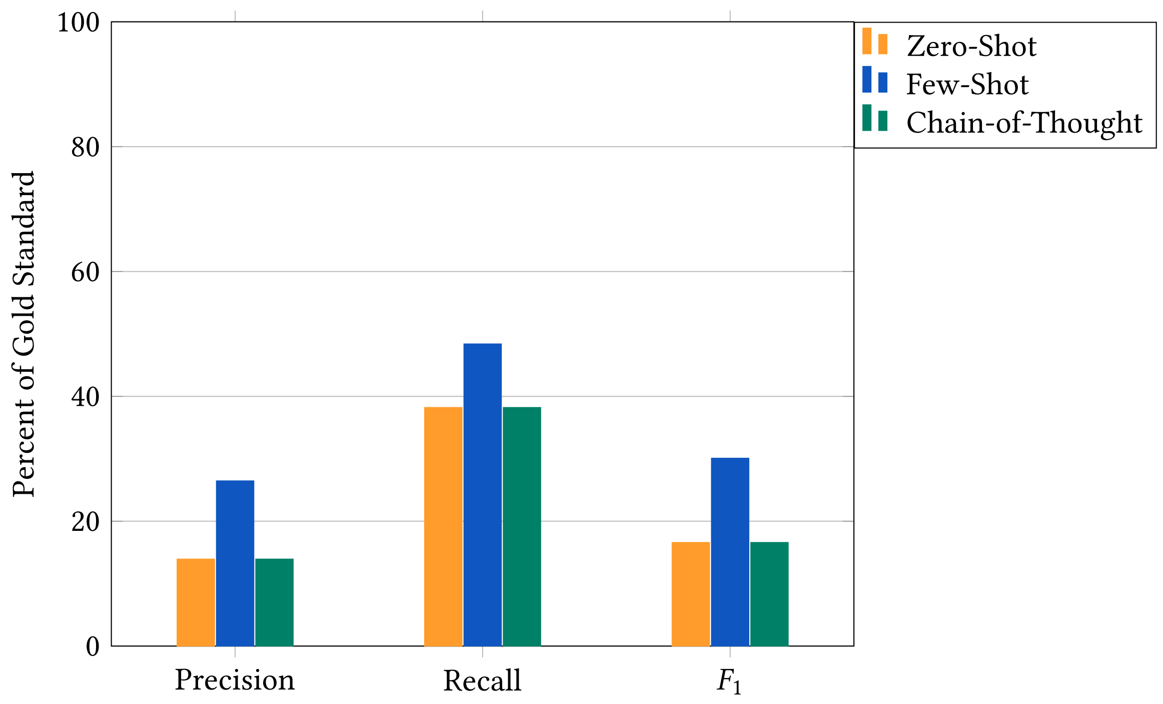


Figure 5.9.: Performance metrics weighted average on Benchmark[18] for MME-ID using GPT-4 and supporting traceability links. In percent of gold standard.

yields worse results than when using no traceability links at all, we do not consider using generated traceability links.

Comparison to ArDoCo (MME)

Our performance is compared to the ArDoCo approach, and the results are presented in Table 5.5, as well as in Figure 5.10 for an overview on weighted averages of our best approaches.

Task	TS			BBB			MS			JR			TM			w. Average		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
ArDoCo	**.96**	**.70**	**.79**	**.89**	**.46**	**.43**	**.21**	**.79**	**.33**	**1.0**	.44	.44	.18	.76	.28	**.60**	**.63**	**.43**
GPT-4 Few-Shot	.21	.21	.21	.12	.38	.17	.12	.30	.16	.71	**.83**	**.73**	**.81**	**.81**	**.81**	.39	.51	.42
GPT-4 Few-Shot w/ id. TLR	.34	.37	.35	.07	.29	.10	.04	.22	.07	.21	.67	.31	.66	.88	.68	.26	.48	.30

Table 5.5.: Comparison of performance metrics for ArDoCo and our approach when using GPT-4 and Few-Shot Prompting, or Few-Shot Prompting and additional information provided in the form of ideal traceability links.
Bold: Best result per metric.

The results show that, in most cases, the ArDoCo approach outperforms ours. However, we see that in the case of *Teammates*, our approach manages to achieve a good result. While ArDoCo's F_1 score lies at 28%, our approach manages to achieve 82%. We can

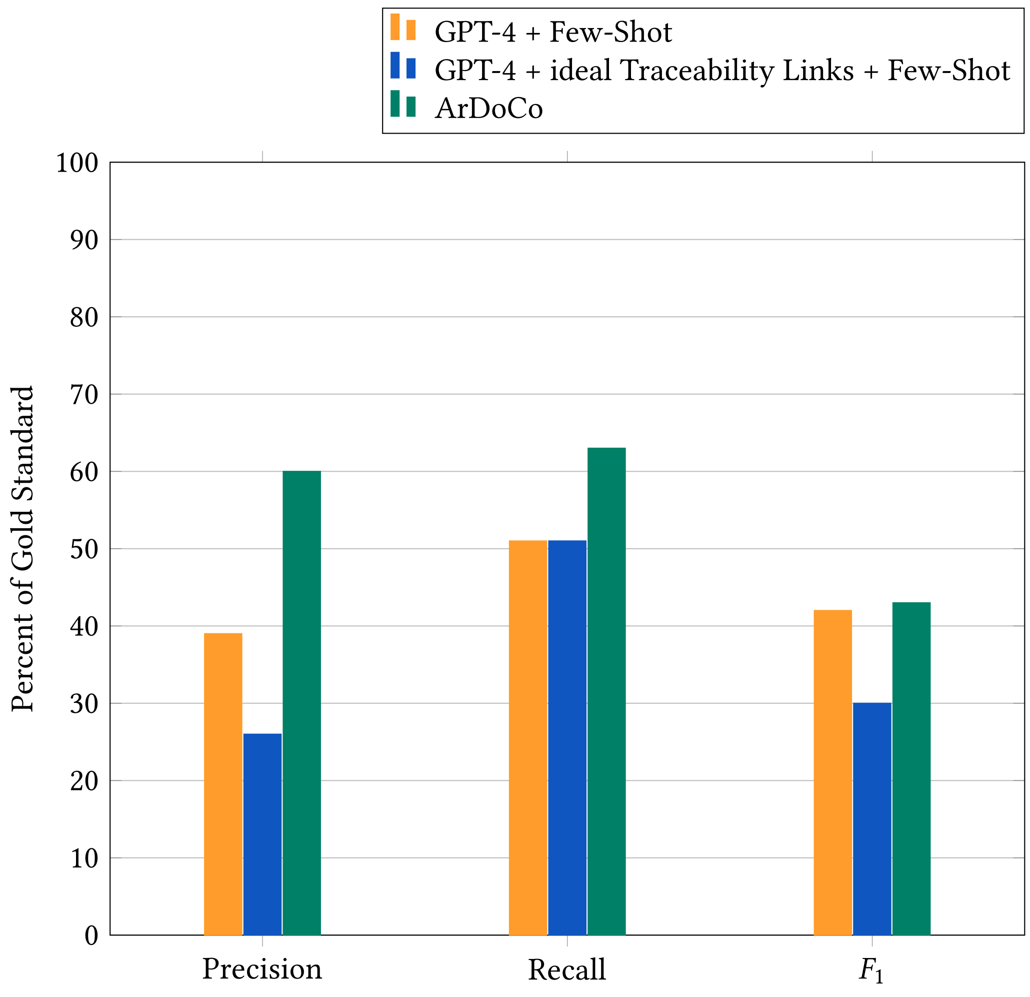


Figure 5.10.: Performance metrics weighted average on Benchmark[18] for our MME approaches compared to ArDoCo's MME performance. In percent of gold standard.

observe how additionally using traceability links worsens performance over using no traceability links at all.

5.3. Discussion

To address the research question we posed in the introduction, we can now look at the results of our experiments. Having done all of our experiments, we now see how the performance of our approaches mainly falls short of the performance of the ArDoCo approach. However, we see some success in some projects for certain use cases, particularly when looking at the MME-ID task. Here, our approach outperforms the ArDoCo approach in the case of the Teammates project. For our comparison, we look at the F_1 metric as it is the most balanced metric. For TLR, we see that our best approach, which is Vector Retrieval pre-processing combined with Chain-of-Thought prompting, lags the ArDoCo approach by 16%. In the case of UME-ID, our best approach, Few-Shot prompting without traceability links, lags the ArDoCo approach by 13%. Finally, for MME-ID, we see that our best approach, which is, again, Few-Shot prompting without traceability links, lags the ArDoCo approach by 1%. Overall, our results show that, while not capable of outperforming ArDoCo in most cases, our approach is often capable of achieving similar results. However, we also see that our approach is not as consistent as the ArDoCo approach. While the ArDoCo approach can achieve similar results across all projects, our approach has some projects where it performs well, while in others, the performance is lacking. This is, for instance, visible in the case of UME-ID, where our approach of Zero-Shot prompting can achieve perfect results in the JabRef project and good results in the MediaStore project, the F_1 score for the BigBlueButton project dips to 40%. This is in contrast to the ArDoCo approach, which never drops below 80% in any project. This can be observed across our three use cases: Our approach usually performs poorly in the case of BigBlueButton and well for JabRef. This may be due to the differences in the structure of the SAD. The SAD for BigBlueButton is divided into chapters with individual headings and contains references to graphics that can not be resolved. Meanwhile, the SAD for JabRef is shorter as well as more concise. There are no sections or chapters, and the SAD is structured as a single text. This structure might be part of the reason why the LLM achieves a higher performance for JabRef.

6. Conclusion

Our evaluation shows how LLMs show drastically different performance for the different use cases we defined in chapter 4. We examined performance for TLR – with or without Vector Retrieval pre-processing –, UME-ID and MME-ID, each with and without Traceability Links in the prompt. The results show how LLMs can perform well for some use cases, but not for others. For the different use cases we find that different prompting methods work best. For TLR, Chain-of-Thought prompting works best, while UME-ID performs best with Zero-Shot prompting and no traceability links, unless when given perfect traceability links, in which case Few-Shot prompting yields perfect performance. MME-ID performs best with Few-Shot prompting without traceability links. We come to the conclusion that when employing LLMs, even slight changes in use cases can make a stark difference in performance and that there is no one-size-fits-all solution. Our approach performs worse than the state-of-the-art ArDoCo[27]. However, we often achieve similar results or even outperform ArDoCo in certain scenarios (cf. section 5.3). We believe that future work can perform better than we did by using more sophisticated prompting methods and LLMs.

6.1. Threats to Validity

One of the main threats to the validity of our approach is the lack of established best practices for using Large Language Models (LLMs).

Using Large Language Models for enterprise applications is a relatively new field of research. As such, there is yet to be a real established best practice for using them. Until recently, many developers have been using LLMs — and predominantly OpenAI models — in a very ad-hoc manner, directly accessing the respective API, estimates Replit[10]. They find that, especially in Q2 of 2023, the LangChain wrapper for using LLMs has seen widespread adoption and currently serves as the de facto standard for developing LLM-based applications.

Prompting is also one of the main areas for research in the field of LLMs. The fact that different prompting methods work better for different use cases is a testament to that. A reason for this could be a poorly written prompt in the case of UME-ID, which caused Zero-Shot prompting to yield the best results. However, it could also mean that we found a prompt that works particularly well for UME-ID in a Zero-Shot environment. Best practices for writing prompts are still being researched. Approaches that work in some use cases might not work in others. More research is needed to fully understand the best practices for using LLMs in different contexts.

Additionally, the changing environment around LLMs is a threat to validity. Prompting methods that proved reliable for an older LLM might not work as well in a current one. One example of this we could observe in our work when using GPT-3 and GPT-4. GPT-3 often struggles with Chain-of-Thought instructions, while GPT-4 performs much better or at least shows expected behavior.

Additionally, the use of LLMs, in general, is still a relatively new field of research, and there is much that is still unknown about their capabilities and limitations. As such, there may be other factors that we have not considered that could impact the validity of our results.

6.2. Outlook

For a more flexible approach, the emergence of multimodal models is a promising future direction to consider. Multimodal models are capable of using information from a variety of sources, including images and text. This would be useful for working with architecture diagrams, which often take various forms. Often, software architecture models stray from standard UML notation. Currently, for Large Language Models, we are limited to textual form factors known to the model, such as UML represented using XML notation. Multimodal models could overcome this limitation and extrapolate relevant information from a supplied image of the model in question. Judging by the results of early experiments with multimodal models, this method would likely yield good results for notation formats never seen before[7].

For future work, we see potential in exploring other models. Future OpenAI models seem logical, but open-source models such as Falcon[37] also show promise. Fine-tuning LLMs is another direction of research that should be explored for our use cases. In the open-source community, Fine-Tuned LLMs based on Llama-2[52] models are already being used for different tasks. Fine-tuning a model for TLR or ID could yield higher performance than using a generic one.

Inconsistencies within the same artifact are also rarely considered, as most approaches focus on finding inconsistencies between different artifacts, like software architecture models and documents, respectively. However, inconsistencies can also arise within the same artifact, further exacerbating program comprehension for developers.

Bibliography

[1] Adetokunbo Adenowo and Basirat Adenowo. “Software Engineering Methodologies: A Review of the Waterfall Model and Object- Oriented Approach”. In: *International Journal of Scientific and Engineering Research* 4 (Aug. 2020), pp. 427–434.

[2] Sumayia Al-Anazi, Hind AlMahmoud, and Isra Al-Turaiki. “Finding Similar Documents Using Different Clustering Techniques”. In: *Procedia Computer Science* 82 (2016). 4th Symposium on Data Mining Applications, SDMA2016, 30 March 2016, Riyadh, Saudi Arabia, pp. 28–34. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2016.04.005. URL: https://www.sciencedirect.com/science/article/pii/S1877050916300199.

[3] Giuliano Antoniol et al. “Information Retrieval Models for Recovering Traceability Links between Code and Documentation”. In: *Software Maintenance, IEEE International Conference on* 0 (Aug. 2000), p. 40. DOI: 10.1109/ICSM.2000.883003.

[4] Iz Beltagy, Matthew E. Peters, and Arman Cohan. *Longformer: The Long-Document Transformer.* 2020. arXiv: 2004.05150 [cs.CL].

[5] Rishi Bommasani et al. “On the Opportunities and Risks of Foundation Models”. In: (Aug. 2021).

[6] Elke Bouillon, Patrick Mäder, and Ilka Philippow. “A Survey on Usage Scenarios for Requirements Traceability in Practice”. In: ed. by Andreas L Doerr Joerg and Opdahl. Springer Berlin Heidelberg, 2013, pp. 158–173. ISBN: 978-3-642-37422-7.

[7] Anthony Brohan et al. “RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control”. In: *arXiv preprint arXiv:2307.15818.* 2023.

[8] Tom B. Brown et al. “Language models are few-shot learners”. In: vol. 2020-December. 2020.

[9] Sébastien Bubeck et al. “Sparks of Artificial General Intelligence: Early experiments with GPT-4”. In: (Mar. 2023).

[10] Jeff Burke. *State of AI Development.* URL: https://blog.replit.com/ai-on-replit.

[11] Silei Cheng et al. “Prompting GPT-3 To Be Reliable”. In: May 2023. URL: https://www.microsoft.com/en-us/research/publication/prompting-gpt-3-to-be-reliable/.

[12] Kevin Clark et al. “ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators”. In: (Mar. 2020).

[13] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: (Oct. 2018).

[14] Tian-bao Du et al. “Automatic traceability link recovery via active learning”. In: *Frontiers of Information Technology & Electronic Engineering* (2020). DOI: 10.1631/FITEE.1900222.

[15] A. Egyed. “Scalable consistency checking between diagrams - the VIEWINTEGRA approach”. In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001).* 2001, pp. 387–390. DOI: 10.1109/ASE.2001.989835.

[16] Hugging Face. *Lexical Analysis - Tokenizer.* URL: https://huggingface.co/docs/transformers/main_classes/tokenizer (visited on 05/08/2023).

[17] Hugging Face. *Using Transformers - Summary of the tasks - Language Modeling.* URL: https://huggingface.co/transformers/v3.4.0/task_summary.html#language-modeling (visited on 05/08/2023).

[18] Dominik Fuchß et al. *Establishing a Benchmark Dataset for Traceability Link Recovery between Software Architecture Documentation and Models.* 46.23.01; LK 01. Karlsruher Institut für Technologie (KIT), 2022. DOI: 10.5445/IR/1000151962.

[19] Leo Gao et al. “The Pile: An 800GB Dataset of Diverse Text for Language Modeling”. In: (Dec. 2020).

[20] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. “Semantically Enhanced Software Traceability Using Deep Learning Techniques”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* 2017, pp. 3–14. DOI: 10.1109/ICSE.2017.9.

[21] J.H. Hayes, A. Dekhtyar, and J. Osborne. “Improving requirements tracing via information retrieval”. In: *Proceedings. 11th IEEE International Requirements Engineering Conference, 2003.* 2003, pp. 138–147. DOI: 10.1109/ICRE.2003.1232745.

[22] Tobias Hey et al. “Improving Traceability Link Recovery Using Fine-grained Requirements-to-Code Relations”. In: Institute of Electrical and Electronics Engineers (IEEE), 2021, pp. 12–22. ISBN: 978-1-66542-882-8. DOI: 10.1109/ICSME52107.2021.00008.

[23] Krystal Hu. *ChatGPT sets record for fastest-growing user base - analyst note.* Feb. 2023.

[24] Ziwei Ji et al. “Survey of Hallucination in Natural Language Generation”. In: *ACM Computing Surveys* 55 (12 Dec. 2023), pp. 1–38. ISSN: 0360-0300. DOI: 10.1145/3571730.

[25] Chao Jia et al. “Scaling Up Visual and Vision-Language Representation Learning With Noisy Text Supervision”. In: ed. by Marina Meila and Tong Zhang. Vol. 139. PMLR, Aug. 2021, pp. 4904–4916. URL: https://proceedings.mlr.press/v139/jia21b.html.

[26] Yuchen Jiang et al. “Quo vadis artificial intelligence?” In: *Discover Artificial Intelligence* 2 (1 Mar. 2022), p. 4. ISSN: 2731-0809. DOI: 10.1007/s44163-022-00022-8.

[27] 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, pp. 141–152. DOI: 10.1109/ICSA56044.2023.00021.

[28] Suntae Kim and Dongsun Kim. “Automatic identifier inconsistency detection using code dictionary”. In: *Empirical Software Engineering* 21 (2 2016). ISSN: 15737616. DOI: 10.1007/s10664-015-9369-5.

[29] Per Kroll and Philippe Kruchten. *The Rational Unified Process Made Easy.* 2003, p. 15.

[30] Raúl Lapeña et al. “Leveraging execution traces to enhance traceability links recovery in BPMN models”. In: *Information and Software Technology* 146 (June 2022), p. 106873. ISSN: 09505849. DOI: 10.1016/j.infsof.2022.106873.

[31] Angie Lee. *What Are Large Language Models Used For?* Jan. 2023. URL: https://blogs.nvidia.com/blog/2023/01/26/what-are-large-language-models-used-for/.

[32] Yalin Liu, Jinfeng Lin, and Jane Cleland-Huang. “Traceability Support for Multi-Lingual Software Projects”. In: Association for Computing Machinery, 2020, pp. 443–454. ISBN: 9781450375177. DOI: 10.1145/3379597.3387440. URL: https://doi.org/10.1145/3379597.3387440.

[33] Tomas Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: (Jan. 2013).

[34] Niklas Muennighoff et al. “MTEB: Massive Text Embedding Benchmark”. In: *arXiv preprint arXiv:2210.07316* (2022). DOI: 10.48550/ARXIV.2210.07316. URL: https://arxiv.org/abs/2210.07316.

[35] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. “Making inconsistency respectable in software development”. In: *Journal of Systems and Software* 58 (2 2001), pp. 171–180. ISSN: 01641212. DOI: 10.1016/S0164-1212(01)00036-X.

[36] Long Ouyang et al. “Training language models to follow instructions with human feedback”. In: *arXiv preprint arXiv:2203.02155* (2022).

[37] Guilherme Penedo et al. *The RefinedWeb Dataset for Falcon LLM: Outperforming Curated Corpora with Web Data, and Web Data Only.* 2023. arXiv: 2306.01116 [cs.CL].

[38] Sebastian Raschka. *Understanding Large Language Models – A Transformative Reading List.* Feb. 2023. URL: https://sebastianraschka.com/blog/2023/llm-reading-list.html (visited on 05/05/2023).

[39] Alberto D Rodriguez, Katherine R Dearstyne, and Jane Cleland-Huang. *Prompts Matter: Insights and Strategies for Prompt Engineering in Automated Software Traceability.* 2023.

[40] Danissa V Rodriguez and Doris L Carver. “Comparison of Information Retrieval Techniques for Traceability Link Recovery”. In: 2019, pp. 186–193. DOI: 10.1109/INFOCT.2019.8710919.

[41] Kaz Sato and Tomoyuki Chikanaga. *Find anything blazingly fast with Google’s vector search technology.* Dec. 2021. URL: https://cloud.google.com/blog/topics/developers-practitioners/find-anything-blazingly-fast-googles-vector-search-technology (visited on 08/20/2023).

[42] Aaron Schlutter and Andreas Vogelsang. "Trace Link Recovery using Semantic Relation Graphs and Spreading Activation". In: *2020 IEEE 28th International Requirements Engineering Conference (RE)*. 2020, pp. 20–31. DOI: 10.1109/RE48521.2020.00015.

[43] Sander Schulhoff and Community Contributors. *Learn Prompting*. URL: https://github.com/trigaten/Learn_Prompting (visited on 05/04/2023).

[44] Roie Schwaber-Cohen. *What is a Vector Database?* URL: https://www.pinecone.io/learn/vector-database/ (visited on 05/15/2023).

[45] Thibault Sellam, Dipanjan Das, and Ankur P. Parikh. "BLEURT: Learning Robust Metrics for Text Generation". In: (Apr. 2020).

[46] George Spanoudakis et al. "Rule-based generation of requirements traceability relations". In: *Journal of Systems and Software* 72 (2 July 2004), pp. 105–127. ISSN: 01641212. DOI: 10.1016/S0164-1212(03)00242-5.

[47] Theo Steiner and Rui Zhang. *Code Comment Inconsistency Detection with BERT and Longformer*. 2022. arXiv: 2207.14444 [cs.CL].

[48] Hendrik Strobelt et al. "Interactive and Visual Prompt Engineering for Ad-hoc Task Adaptation with Large Language Models". In: *IEEE Transactions on Visualization and Computer Graphics* 29 (1 2023), pp. 1146–1156. DOI: 10.1109/TVCG.2022.3209479.

[49] Shin Hwei Tan et al. "@tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012, pp. 260–269. DOI: 10.1109/ICST.2012.106.

[50] Rohan Taori et al. *Stanford Alpaca: An Instruction-following LLaMA model*. 2023.

[51] *Tokenizer - OpenAI API*. URL: https://platform.openai.com/tokenizer (visited on 05/10/2023).

[52] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL].

[53] P. D. Turney and P. Pantel. "From Frequency to Meaning: Vector Space Models of Semantics". In: *Journal of Artificial Intelligence Research* 37 (Feb. 2010), pp. 141–188. ISSN: 1076-9757. DOI: 10.1613/jair.2934.

[54] Ashish Vaswani et al. "Attention is all you need". In: vol. 2017-December. 2017.

[55] Xuezhi Wang et al. "Self-consistency improves chain of thought reasoning in language models". In: *arXiv preprint arXiv:2203.11171* (2022).

[56] Jason Wei et al. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". In: (Jan. 2022).

[57] Xin Xia et al. "Measuring Program Comprehension: A Large-Scale Field Study with Professionals". In: *IEEE Transactions on Software Engineering* 44 (10 2018), pp. 951–976. DOI: 10.1109/TSE.2017.2734091.

[58] Shunyu Yao et al. *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*. 2023. arXiv: 2305.10601 [cs.CL].

A. Appendix

The source code we produced for this thesis can be found at 10.5281/zenodo.8386823. For the sake of brevity, we chose to omit the full prompts and instead only show the most relevant parts. Any information denoted using <<this>> is a placeholder for the actual information. This usually includes SAD or SAM contents, as well as gold standards.

A.1. Representing UML Models

We translate UML models into a format that is easier to work with. We use the name and type of the element and work with that. The model shown in Figure A.1 is translated into the following format:
Command: Component
Caller: Component
ConcreteCommand: Component
Receiver: Component
Client: Component

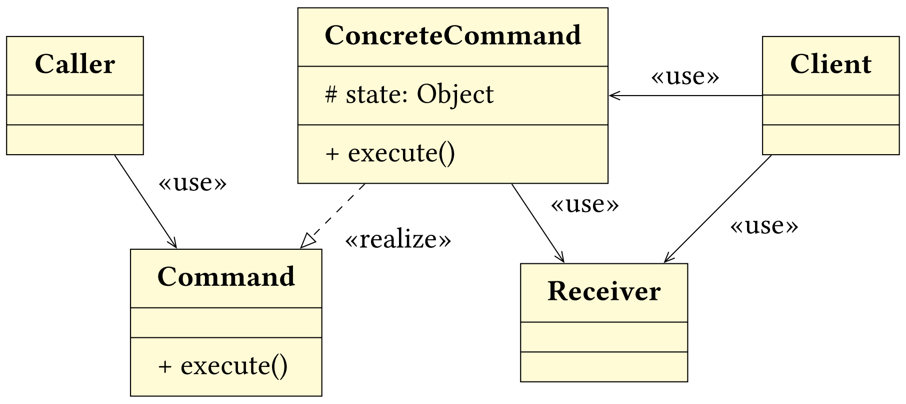


Figure A.1.: UML Model in drawn form

A.2. TLR Prompt

For TLR, we examine using Embeddings for pre-processing and using no pre-processing for the SAD.

No Pre-Processing

We now look at performing TLR with the original SAD.

System Message
You are a system capable of traceability link recovery. The user provides a software architecture model (SAM), such as a UML diagram, and a software architecture document (SAD), such as a design paper. Your task is to generate tracelinks: For each model element, find the locations of their mentions in the document. Write in the style of a csv file: "model element", "document location". The model element is given as the name of the element, as found in the SAM model. The location is given as the line number inside the SAD document, where the corresponding element is found. Beware of context. Sometimes, the same model element is mentioned multiple times in the document and even mentioned indirectly.
The line number is strictly limited by the number of lines in the file!
At all cost, avoid exceeding the number of lines in the current SAD!
I am going to provide a template for your output:
Try to fit the output into the placeholders that I list.
Please preserve the formatting and overall template that I provide.
This is the template:
modelElementName,sentence
ELEMENT_NAME1,LINE_NUMBER1
ELEMENT_NAME2,LINE_NUMBER2
ELEMENT_NAME3,LINE_NUMBER3
Avoid running into the following errors:
- Giving a line number that is not in the document.
- Running off in increments of 1 and counting past the end of a file

Human Message 1-3
Find Trace Links for the given SAM and SAD. For each named entity, name every line it occurs, either mentioned implicitly or explicitly and generate tracelinks. Try to find implicitly mentioned components as well. Denote them under the modelElementName section using the entitiesńame and use the according line number. The line number may not exceed 7
SAD:
<<Example SAD>>
SAM:
<<Example SAM>>

AI Message 1-3
In line 1, we can not find any model elements.
In line 2, the IBoardElement model element is directly mentioned -> IBoardElement,2
In line 3, the IBoardElement model element, the Dome model element and the Stone model element are directly mentioned -> IBoardElement,3;Dome,3;Stone,3
...

modelElementName,sentence
<<Example TLR Gold Standard>>

Human Message 4 (Contains Benchmark Project)
Find Trace Links for the given SAM and SAD. For each named entity, name every line it occurs, either mentioned implicitly or explicitly and generate tracelinks. Try to find implicitly mentioned components as well. Denote them under the modelElementName section using the entitiesńame and use the according line number. The line number may not exceed <<Project SAD Line Number>>
.
SAD:
<<Project SAD>>
SAM:
<<Project SAM>>

With Pre-Processing

We now look at performing TLR with the the VSM-filtered SAD.

System Message
You are a system capable of traceability link recovery. The user will provide snippets of a software architecture model (SAM), such as a UML diagram, and a software architecture document (SAD), such as a design paper. Then, the relevant element will be provided.
Your task is to generate tracelinks for the relevant element. Find the sentences that mention the relevant element. Write in the style of a csv file: "model element", "document location". The location in the SAD is given as the line number inside the SAD document, where the corresponding element is found.
I am going to provide a template for your output:
Try to fit the output into the placeholders that I list.
Please preserve the formatting and overall template that I provide.
The sentences are ordered from highest to lowest confidence. The latter sentences are more likely to be irrelevant for this element.
This is the template:
modelElementName,sentence
ELEMENT_NAME,LINE_NUMBER1
ELEMENT_NAME,LINE_NUMBER2
ELEMENT_NAME,LINE_NUMBER3

Human Message 1-3
You are a system capable of traceability link recovery. The user will provide snippets of a software architecture model (SAM), such as a UML diagram, and a software architecture document (SAD), such as a design paper. Then, the relevant element will be provided.
Your task is to generate tracelinks for the relevant element. Find the sentences that mention

the relevant element. Write in the style of a csv file: "model element", "document location". The location in the SAD is given as the line number inside the SAD document, where the corresponding element is found.
I am going to provide a template for your output:
Try to fit the output into the placeholders that I list.
Please preserve the formatting and overall template that I provide.
The sentences are ordered from highest to lowest confidence. The latter sentences are more likely to be irrelevant for this element.
This is the template:
modelElementName,sentence
ELEMENT_NAME,LINE_NUMBER1
ELEMENT_NAME,LINE_NUMBER2
ELEMENT_NAME,LINE_NUMBER3

SAM:*
<<Example SAM>>

Relevant Element: <<Example Element i>>

SAD:*
<<Example SAD Sentences most relevant to Element i >>

AI Message 1-3
In line 3, the Stone model element is directly mentioned -> Stone,3
...

modelElementName,sentence
<<Example UME Gold Standard>>

Human Message 4 (Contains Benchmark Project)
Find Trace Links for the given relevant element and SAD. Name every line it occurs, either mentioned implicitly or explicitly and generate tracelinks. Try to find when it is mentioned implicitly. Denote them under the modelElementName section using the entities' name and use the according line number.

SAM:*
<<Project SAM>>

Relevant Element: <<Element i of Project SAM>>

SAD:*
<<Retrieved Sentences for Element i of Project SAM>>

Disclaimer* All Human Message Prompts have been marked with an asterisk (*) to indicate that they have been corrected for presentation here. At the time of evaluation, we erroneously included used a structure like the following:
SAM:*
...

Relevant Element: ...

SAD:*
...
However, when we re-evaluate part of the benchmark, the results fall in line with what we have already seen during evaluation. Therefore, we have decided to present the corrected prompts here, even though they do not reflect the actual prompts used during evaluation.

A.3. UME-ID Prompt

For UME, we examine the cases of having traceability links and not having traceability links.

Without Traceability Links

We first look at performing UME without any additional traceability links.

System Message

You are a system capable of inconsistency detection. The user provides a software architecture model (SAM), such as a UML diagram, and a software architecture document (SAD), such as a design paper. Your task is to find inconsistencies between the SAM and SAD: For each component model element, which is not mentioned within the documentation, note the name of the component. Use the same name as found in the SAM model. Unmentioned interfaces are allowed!!
DO NOT CONSIDER INTERFACE TYPE MODEL ELEMENTS!
When there are no unmentioned elements, output "None".
I am going to provide a template for your output:
Try to fit the output into the placeholders that I list.
Please preserve the formatting and overall template that I provide.
This is the template:
unmentionedModelElementName
ELEMENT_NAME1
ELEMENT_NAME2
ELEMENT_NAME3

Human Message 1-3

Find inconsistencies for the given SAM and SAD. For each named component from the

SAM, check if it is mentioned in SAD. If it is not mentioned, append it to the list of missingModelElementIDs. Try to find implicitly mentioned components as well. Do not consider model elements that are of Interface type! 6
SAD:
<<Example SAD>>
SAM:
<<Example SAM>>

AI Message 1-3
Of the mentioned components in the SAM, only the Dome component appears in the SAD. There is no mention of the Stone component in the SAD. Therefore, we add the Stone component to the list of missing model elements.

unmentionedModelElementName
<<Example UME Gold Standard>>

Human Message 4 (Contains Benchmark Project) Find inconsistencies for the given SAM and SAD. For each named component from the SAM, check if it is mentioned in SAD. If it is not mentioned, append it to the list of missingModelElementIDs. Try to find implicitly mentioned components as well. Do not consider model elements that are of Interface type!
SAD:
<<Project SAM>>
SAM:
<<Project SAD>>

With Traceability Links

We now look at performing UME with additional traceability links.

System Message
You are a system capable of inconsistency detection. The user provides a software architecture model (SAM), such as a UML diagram, and a software architecture document (SAD), such as a design paper. Additionally, they will provide a list of traceability links. For every model element, they point to the location that model element can be found in the SAD. Your task is to find inconsistencies between the SAM and SAD: For each component model element, which is not mentioned within the documentation, note the name of the component. Use the same name as found in the SAM model and traceability links. Unmentioned interfaces are allowed!!
DO NOT CONSIDER INTERFACE TYPE MODEL ELEMENTS!
When there are no unmentioned elements, output "None".
I am going to provide a template for your output:
Try to fit the output into the placeholders that I list.
Please preserve the formatting and overall template that I provide.

This is the template:
unmentionedModelElementName
ELEMENT_NAME1
ELEMENT_NAME2
ELEMENT_NAME3

Human Message 1-3
Find inconsistencies for the given SAM and SAD. For each named component from the SAM, check if it is mentioned in SAD. If it is not mentioned, append it to the list of missingModelElementIDs. Try to find implicitly mentioned components as well. Do not consider model elements that are of Interface type! 6
SAD:
<<Example SAD>>
SAM:
<<Example SAM>>
TLR:
modelElementName,sentence
<<Example TLR Gold Standard>>

AI Message 1-3
Of the mentioned components in the SAM, only the Dome component appears in the SAD. There is no mention of the Stone component in the SAD. Therefore, we add the Stone component to the list of missing model elements.

unmentionedModelElementName
<<Example UME Gold Standard>>

Human Message 4 (Contains Benchmark Project)
Find inconsistencies for the given SAM and SAD. For each named component from the SAM, check if it is mentioned in SAD. If it is not mentioned, append it to the list of missingModelElementIDs. Try to find implicitly mentioned components as well. Do not consider model elements that are of Interface type! SAD: <<Project SAD>>
SAM: <<Project SAM>>

A.4. MME-ID Prompt

Without Traceability Links

We first look at performing MME-ID without any additional traceability links.

System Message
You are a system capable of inconsistency detection. The user provides a software architecture model (SAM), such as a UML diagram, and a software architecture document (SAD), such as a design paper. Your task is to find inconsistencies between the SAM and SAD: For each component model element, which is mentioned within the SAD but not in the SAM, note the name of the component. Use the same name as found in the SAD model. Include potential interfaces.
I am going to provide a template for your output:
Try to fit the output into the placeholders that I list.
Please preserve the formatting and overall template that I provide.
This is the template:
missingModelElementName
ELEMENT_NAME1
ELEMENT_NAME2
ELEMENT_NAME3

Human Message 1-3
Find inconsistencies for the given SAM and SAD. For each named component from the SAD, check if it is mentioned in SAM. If it is not mentioned, append it to the list of missingModelElementIDs. Try to find implicitly mentioned components as well. 7
SAD:
<<Example SAD>>
SAM:
<<Example SAM with Element i removed>>

AI Message 1-3
IBoardElement is missing from the SAM. It is mentioned in lines: 2, 3, 4, 5, 7

missingModelElementName
<<Example Element i>>

Human Message 4 (Contains Benchmark Project)
Find inconsistencies for the given SAM and SAD. For each named component from the SAD, check if it is mentioned in SAM. If it is not mentioned, append it to the list of missingModelElementIDs. Try to find implicitly mentioned components as well. 43. SAD:
<<Project SAD>>
SAM: <<Project SAM with Element i removed>>

With Traceability Links

We now look at performing MME-ID with additional traceability links.

System Message
I am going to provide a template for your output:
You are a system capable of inconsistency detection. The user provides a software architecture model (SAM), such as a UML diagram, and a software architecture document (SAD), such as a design paper. Additionally, they will provide a list of traceability links. For every model element, they point to the location that model element can be found in the SAD. Your task is to find inconsistencies between the SAM and SAD: For each component model element, which is mentioned within the SAD but not in the SAM, note the name of the component. Each named entity in the documentation that would be a component in the model, but is not mentioned in the model, should be denoted. Use the same name as found in the SAD model. Include potential interfaces.
Try to fit the output into the placeholders that I list.
Please preserve the formatting and overall template that I provide.
This is the template:
missingModelElementName
ELEMENT_NAME1
ELEMENT_NAME2
ELEMENT_NAME3

Human Message 1-3
Find inconsistencies for the given SAM and SAD. For each named component from the SAD, check if it is mentioned in SAM. If it is not mentioned, append it to the list of missingModelElementIDs. Try to find implicitly mentioned components as well. 7
SAD:
<<Example SAD>>
SAM:
<<Example SAM with Element i removed>>
TLR:
modelElementName,sentence
<<Example TLR with Element i removed>>

AI Message 1-3
IBoardElement is missing from the SAM. It is mentioned in lines: 2, 3, 4, 5, 7

missingModelElementName
<<Example Element i>>

Human Message 4 (Contains Benchmark Project) Find inconsistencies for the given SAM and SAD. For each named component from the SAD, check if it is mentioned in SAM. If it is not mentioned, append it to the list of missingModelElementIDs. Try to find implicitly mentioned components as well. 43. SAD:
<<Project SAD>>
SAM:

<<Project SAM with Element i removed>>
Traceability Links:
modelElementID,sentence
<<Project TLR with Element i removed>>