Enabling Architecture Traceability by LLM-based Architecture Component Name Extraction

Dominik Fuchß, Haoyu Liu, Tobias Hey, Jan Keim, Anne Koziolek
KASTEL - Institute of Information Security and Dependability
Karlsruhe Institute of Technology (KIT), Germany
{dominik.fuchss, haoyu.liu, hey, jan.keim, koziolek}@kit.edu

***Abstract*—Traceability Link Recovery (TLR) is an enabler for various software engineering tasks. One important task is the recovery of trace links between Software Architecture Documentation (SAD) and source code. Here, the main challenge is the semantic gap between the two artifact types. Recent research has shown that this semantic gap can be bridged by using Software Architecture Models (SAMs) as intermediates. However, the creation of SAMs is a manual and time-consuming task. This paper investigates the use of Large Language Models (LLMs) to extract component names as simple SAMs for TLR based on SAD and source code. By doing so, we aim to bridge the semantic gap between SAD and source code without the need for manual SAM creation. We compare our approach to the state-of-the-art TLR approaches TransArC and ArDoCode. TransArC is the currently best-performing approach for TLR between SAD and source code, but it requires SAMs as an additional artifact. Our evaluation shows that our approach performs comparable to TransArC (weighted average F1 with GPT-4o: 0.86 vs. TransArC's 0.87), while only needing the SAD and source code. Moreover, our approach significantly outperforms the best baseline that does not need SAMs (weighted average F1 with GPT-4o: 0.86 vs. ArDoCode's 0.62). In summary, our approach shows that LLMs can be used to make TLR between SAD and source code more applicable by extracting component names and omitting the need for manually created SAMs.**

***Index Terms*—Traceability Link Recovery, Large Language Models, Software Architecture, Model Extraction**

I. Introduction

In software development, numerous artifacts are produced, each representing different levels of abstraction and addressing distinct aspects of the system. Challenges for architects and developers arise because the relationships between these artifacts are often unclear, preventing their effective use. To address this, traceability link recovery (TLR) techniques are used to establish, maintain, and manage explicit trace links between artifacts. Improving software quality is closely tied to the creation and management of trace links [1], [2].

One challenge in linking artifacts is the semantic gap between different types of artifacts. Bridging this gap is difficult and automated approaches often misinterpret the underlying semantics. To address this, some methods suggest using intermediate artifacts to reduce the semantic gap, making it easier to link related artifacts [3]–[6]. For example, the descriptions in design documentation are semantically closer to design artifacts like software architecture models (SAMs) than to code, and SAMs in turn are closer to code. Consequently, specialized approaches can more easily establish links between software architecture documentations (SADs) and SAMs, or between SAMs and code [7], [8]. Based on these insights, transitive approaches like Transitive links for Architecture and Code (TransArC) [6] have been developed. TransArC recovers trace links between SADs and source code based on manually created SAMs as intermediate artifacts. However, these methods are not always applicable, as intermediate artifacts are often unavailable.

We propose a novel approach to address this challenge for TLR between SAD and source code. Since, in practice, SAMs are often not available, we aim to recover trace links between SAD and code without the need for manually created SAMs. To achieve this, we leverage the strength of large language models (LLMs) in understanding natural language and code. We design an approach that uses LLMs to recover SAMs in the form of component names from SAD and/or code. Component names contribute the required information to apply TransArC without manually creating SAMs. In doing so, we bridge the semantic gap between SAD and code and empower TLR between these artifacts. We emphasize that the component names define a simple SAM that provides exactly the information for TLR. In contrast to the research field of architecture recovery that aims to recover more detailed architecture models, we focus on information needed to enable state-of-the-art SAD to code TLR approaches such as TransArC.

Consequently, we have the following research questions:

RQ1 Is the performance of architecture TLR with LLM-extracted component names as intermediate artifacts comparable to using manually created SAMs?

RQ2 Does our approach perform better than state-of-the-art TLR between SAD and code without SAMs as intermediates?

RQ3 Is the performance of current open-source LLMs comparable to the performance of closed-source ones?

RQ4 How does the performance of the approach differ when using different artifacts to generate the SAMs?

The main contribution of this paper is our novel approach to extract architecture component names for transitive TLR between SAD and code, removing the need for manually created SAMs. Moreover, we provide an exhaustive evaluation of our approach with different LLMs, projects, modes, and state-of-the-art approaches for TLR. We provide code, baselines, evaluation data, and results in a replication package [9].

We structure the remainder of the paper as follows: Related work is examined in Section II. Our approach is presented in Section III including a description of how we incorporated TransArC [6] in Section III-C. In Section IV, we describe our experimental design. Section V presents the results of our experiments, analyzes the LLMs' errors, and discusses threats to validity. Lastly, we conclude this paper in Section VI.

II. Related Work & Foundations

In this section, we first discuss the ideas and benefits of transitive links, comparing previous works with ours. Afterward, we focus on the application of LLMs for TLR. Here, we concentrate on the information extraction ability of LLMs that also motivates our use of LLMs. Finally, we discuss architecture recovery and its differences from our component name recovery approach.

A. Transitive Trace Links

Automated TLR approaches primarily involve comparing different terms across textual artifacts, to find terms referring to the same concept. Consequently, researchers have leveraged semantic similarity techniques developed within the Natural Language Processing (NLP) field to facilitate this process.

Among those techniques, methods like vector space model (VSM) and latent semantic indexing (LSI) are commonly used [10]. However, software artifacts exist at various levels of abstraction. This complexity challenges NLP models, as they often struggle to handle cross-level artifacts effectively, limiting their accuracy in recovering trace links. Thus, researchers have explored different techniques to address this challenge. Approaches include incorporating fine-grained information [11], considering dependencies [12], [13], and using enriched vocabularies [14]. More recently, transitive linking through intermediate artifacts has shown to be promising in bridging those semantic gaps [6], [15].

The underlying idea of transitive links is that intermediate artifacts can find implicit tracing relationships [16]. An early work by Nishikawa et al. [4] showed the importance of choosing suitable intermediate artifacts for transitive link recovery. Their approach focuses on establishing transitive links between two artifacts using a third artifact [4]. It uses VSM to generate initial trace links between various pairs of artifacts. These links include pairs between use cases, interaction diagrams, code, and test cases. The experiment was conducted under various settings, including scenarios with no intermediate artifacts and with different artifacts used as intermediates. Their findings highlight that suitable intermediate artifacts can significantly influence TLR performance. For example, interaction diagrams are better intermediates than use cases when recovering trace links between code and test cases. Instead of only using VSM to generate initial links for later transitive linking, various NLP techniques can be combined [15], [16]. COMET [16] uses a Bayesian inference framework to treat the recovery process from a probabilistic view. It uses multiple similarity scores to estimate the model's parameter. In contrast to COMET, Rodriguez et al. [15] first combines multiple scores from different sources into a single score, and then evaluates the link based on the final score.

So far, previous works mainly focused on traceability in requirement [10]–[12], [14]–[16] or test cases [4], [13], [16], leaving architectural traceability [6], [8] less explored. TransArC [6] has achieved the best performance in the architectural traceability benchmark. It leverages SAMs as intermediate artifacts to recover trace links between SAD and code. Although it achieves significant improvement, its reliance on the existence of well-maintained SAMs limits its application to a wider range of projects.

Transitive approaches have shown promising results but cannot work when the intermediate artifacts are missing. Therefore, it remains a challenge how to leverage transitive links when the intermediate artifacts are not completely present. To enable transitive trace link recovery of SAD and source code in more settings, we explore using LLMs to recover the needed information of a SAM for TLR.

B. LLMs for Traceability

With the rich general knowledge obtained from pretraining, LLMs have been effective at knowledge-intensive software engineering tasks [17]–[19]. Besides, LLMs have been used for various architecture and modeling tasks [20], [21], including supporting design decision making [22], [23], modeling tasks [24], [25], and software architecture analysis and generation [26], [27]. In the following, we discuss the role of LLMs in advancing traceability research.

T-BERT is an early adoption of LLMs for TLR in an issue-commit setting [28]. The BERT language model was tested using three variants of neural architecture: Twin, Siamese, and Single. They demonstrated that T-BERT outperforms previous VSM and recurrent neural network approaches.

With the rise of decoder-only LLMs like GPT, prompt phrasing greatly influences the language model's output [29]. Rodriguez et al. [29] explored the performance of those LLMs on recovering links with different types of prompts: classification (two artifacts are linked or not), ranking (rank all related artifacts), and Chain-of-thought (recovery step-by-step while giving reasons). Besides, they showed that LLMs can understand domain-specific terms from the general knowledge obtained by pre-training. Motivated by this understanding abilities, we explore LLMs to extract intermediate structures from architecture documents and source code to help TLR.

Hassine [30] explored LLMs' zero-shot ability to trace security-related requirements to goal models. The prompts are tailored to the Goal-oriented Requirements Language's peculiarities. Although achieving positive results, the approach's high dependence on the task and data limits its further application to our architectural TLR task. North et al. [31] considered that using requirements-to-code links helps LLMs generate code by iteratively reformatting prompts. They use the gradients between generated code and requirements sentences to identify which part of the requirement is overlooked. This overlooking link is later used to reformat the prompt.

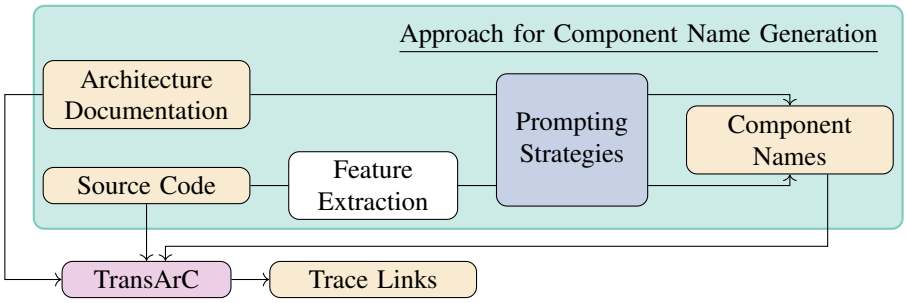

Fig. 1. Overview of the Approach for TLR. Artifacts in Orange, Prompting in Blue, Extraction of Features in White, and TransArC in Purple.

Existing LLM approaches for TLR show that LLMs can have the software knowledge needed for our task. However, our task is different from the tasks already tackled, so these approaches can't be directly applied. Our key contribution is leveraging the information extraction capabilities of LLMs in zero-shot mode within a tailored approach for SAD to source code traceability.

C. Architecture Recovery

Our approach's generation of component names is conceptually related to software architecture recovery research. Therefore, we give a brief overview of architecture recovery approaches and describe the differences. In the end, we discuss the difference between ours and previous approaches.

Structural information, including system dependencies [32] and folder structure, is an important source for architecture recovery. An early approach, ACCD [33], uses folder and dependency information gathered through static analysis. Building upon ACCD, dynamic dependency information has been shown to help recovery in some cases [34]. Lutellier et al. [32] examined the impact of code dependency in detail, finding that symbol dependencies yield better recovery accuracy than include dependencies. In addition to structural information, textual information can also help. The primary idea is that artifacts belonging to the same component may have similar variable names. ARC [35] recovers architecture using system concerns, which are application specific features extracted from software corpora. Corazza et al. [36] studied the impact of six different types of identifier names. They found it would be better to treat identifiers separately by their types and assign suitable weights than to treat them equally in a big vocabulary. Structural and semantic information can also be combined. They can be integrated to build weighted call graphs [37], create recovery patterns [38], and be combined using information fusion models [39]. Rukmono et al. [40] showed architecture could also be recovered deductively. They start from a reference architecture and iteratively use LLMs to refine it with implementation details.

Although our approach shares similar ideas to architecture recovery, its purpose and final result are different from those of previous works. We aim to recover a high-level architecture that serves as intermediate between architecture documentation and code for the purpose of TLR. So we recover the simple SAM with only component names without any behavior definition. In contrast, previous approaches focus on recovering detailed software architecture for more general purposes, like understanding the system's functionality or performance prediction.

III. Approach

This section describes our approach to generate a simple SAM from SAD and source code to enable transitive TLR.

Figure 1 provides an overview of our approach. The approach consists of two main parts: Generating the component names for a SAM and recovering trace links. The input for the approach is the SAD and the software project's source code. Since the whole source code is typically too large to be used as input for an LLM, we extract features from the source code. We then use prompting strategies to extract and generate component names for the intermediate SAM. Using these SAMs, we apply the TransArC [6] approach to recover trace links between the SAD and the source code. In the following, we describe the individual steps in detail.

A. Feature Extraction for Source Code

This section describes the feature extraction process for generating component names from source code. We extract features from the source code to reduce the input size. Since we aim to recover component names from the source code, we do not need to maintain all the information.

In object-oriented programming languages like Java, the package structure can provide valuable information about the architecture of a system [41]. Thus, we extract the package structure from the source code. Our approach is not limited to this feature, but the feature extraction in this paper focuses only on providing a list of non-empty source code packages.

B. Prompting Strategies

This section describes the prompting strategies we use to generate the SAM. We consider three modes for this paper: First, we extract the component names only from the SAD. Second, we extract the component names only from the source code. Third, we incorporate both to generate the simple SAMs.

a) Extract Component Names from Documentation: We use Chain-of-thought prompting to generate the simple SAM from the architecture documentation. We use two prompts:

> **Prompt 1: Documentation to Architecture (1)** Your task is to identify the high-level components based on the software architecture documentation. In a first step, you shall elaborate on the following documentation: {Software Architecture Documentation}

> **Prompt 2: Documentation to Architecture (2)** Now provide a list that only covers the component names. Omit common prefixes and suffixes in the names in camel case. Output format:
> - Name1
> - Name2

Prompt 1 queries the LLM to identify high-level components based on the architecture documentation. We also instruct the LLM to elaborate on the architecture documentation. Thus, the LLM is not restricted to generating output in a defined format. In the second step, we use Prompt 2 to generate a list of component names. Here, we instruct the model to only provide a list of component names without common prefixes and suffixes. We want to ensure that the component names only contain the name, not prefixes or suffixes like *Component*. We aim to reduce the complexity to facilitate the retrieval of component names by defining an output format.

b) Generate Component Names from Source Code: We also use Chain-of-thought prompting to generate the simple SAM from the source code in two prompting steps.

> **Prompt 3: Code to Architecture (1)** You get the {Features} of a software project. Your task is to summarize the {Features} w.r.t. the high-level architecture of the system. Try to identify possible components. {Features}: {Content}

Prompt 3 queries the model to summarize the features of a software project w.r.t. the system's architecture. Here, we instruct the model to identify possible components based on the features extracted from the source code. In this paper, we used 'Packages' as the feature. This feature names all non-empty packages. We then re-use Prompt 2.

c) Generate Component Names from SAD and Source Code: Lastly, we consider a combination of the SAD and the source code to generate the simple SAM. Here, we decided to use two modes for the combination.

First, if we extract the SAM from the documentation and the source code, we can aggregate the results by using LLMs. For this purpose, we use the following prompt:

> **Prompt 4: Aggregation** You get a list of possible component names. Your task is to aggregate the list and remove duplicates. Omit common prefixes and suffixes in the names in camel case. {Output Format (cf. Prompt 2)} Possible component names: {Possible Component Names}

In Prompt 4, we ask the model to aggregate the list of possible component names and remove duplicates. Afterward, we use the same statements regarding prefixes, suffixes and output format as in the other prompts.

In the second mode, we aggregate the results using word similarity metrics. We calculate the similarity between the component names generated from the documentation and the source code. To merge the component names, we process them sequentially, starting with the component names from the documentation. The normalized Levenshtein distance is already known from TLR tasks [8], [42]. We use this Levenshtein distance [43] to calculate the similarity between the possible component names. If the similarity is above a certain threshold, we consider the component names as equal and omit the new one. Since we assume that the SAD is closer to the actual SAM than the source code, we start aggregation with the extracted component names from SAD. In doing so, we want to complement the component extracted from SAD with component names extracted from the source code. Moreover, this also merges similar component names.

d) Interpretation of Responses: Among the most challenging aspects of using LLMs is interpreting the generated responses because there is no guarantee that they adhere to the requested output format. In our approach, the final output of the LLM should generate a list of component names. This output is determined by the final prompt in each mode (cf. Prompt 2). To parse the responses, we consider every line of the final response. First, we check that the trimmed line starts with the character '-'. If this is the case, we consider the line as a component name. We remove the occurrences of 'components' and 'component' from the component names. Thus, we aim to only get the actual component names. Third, we remove any space from the component's name, as we requested the component names to be in camel case. Finally, we remove any duplicates from the list of component names.

C. TransArC Approach

To be able to analyze the effect of LLM-extracted intermediate models on transitive TLR between SAD and code, we make use of the TransArC approach by Keim et al. [6]. TransArC links SAD to source code by using SAMs as intermediate artifacts to bridge the semantic gap. The approach consists of two linking phases: linking documentation to SAMs, and the models to source code. The remainder of this section gives a brief overview of them.

In the first phase, TransArC uses the existing ArDoCo approach [8] to create trace links between SAD and SAMs. ArDoCo employs NLP techniques to analyze the SAD to

identify architectural elements such as components. ArDoCo uses various similarity measures and heuristics to link the sentences in the SAD and components in the SAM.

In the second phase, TransArC uses ARchitecture-to-COde Trace Linking (ArCoTL) to establish trace links between SAMs and the source code. ArCoTL first transforms the input artifacts into intermediate representations. For SAMs, this includes identifying components, while for code, it involves extracting elements like classes, methods, and packages. ArCoTL then applies a series of heuristics to identify correspondences between architectural elements and code entities. A computational graph combines these heuristics to aggregate the confidence levels of candidates to form the final trace links.

Combining the results from both phases, TransArC generates transitive trace links between SAD and source code. It uses the intermediate SAMs to enhance the accuracy of trace link recovery, effectively reducing the semantic gap between the documentation and code. The evaluation of (cf. [6]), demonstrated its high performance in recovering trace links (weighted average F_1-score of 0.87).

We use the TransArC approach to create trace links between the SAD and the source code. Therefore, we use the component names generated by our approach as SAM to enable trace linking between architecture documentation and source code without needing a manually created component model.

IV. Experimental Design

In this section, we present the evaluation setup, including the dataset, evaluation metrics, and the used LLMs. Additionally, we present the baselines to which we compare our approach.

A. Evaluation Setup: Data and Metrics

This section describes the datasets and evaluation metrics, we use for evaluation. We use the same setup as in the original TransArC publication [6], i.e., we use the same dataset and evaluation metrics to compare the results. As our approach generates SAMs, we do not use the manually created SAMs.

a) Benchmark Dataset: The TransArC approach uses a benchmark dataset from Fuchß et al. [42] comprising SAD to SAM trace links. Keim et al. [6] extended the dataset with trace links between SAD and source code. The dataset consists of five open-source projects, each differing in size and domain. The projects are MediaStore (MS), TeaStore (TS), TEAMMATES (TM), BigBlueButton (BBB), and JabRef (JR). The dataset contains the artifacts themselves and the ground truth for trace links. Table I provides an overview of the dataset. Every project has, at most, 14 components. The number of source code files ranges from around 100 to roughly $2,000$, and the SADs comprise 13 up to 198 sentences.

b) Evaluation Metrics: We use commonly used metrics for TLR tasks [44], [45]: precision, recall, and their harmonic mean F_1-score (see Equation 1 and Equation 2). This way, we can compare our approach's performance to the reported results of other state-of-the-art approaches.

TABLE I
Number of Artifacts per Artifact Type and Number of Trace Links in the Gold Standard for each Project Based on [6].

Artifact Type		MS	TS	TM	BBB	JR
Arch. Docs	# Sentences	37	43	198	85	13
Arch. Model	# Components	14	11	8	12	6
Source Code	# Files	97	205	832	547	1,979
Docs-Code	# Trace links	50	707	7,610	1,295	8,240

$$\text{Precision} = \frac{\text{TP}}{\text{TP}+\text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP}+\text{FN}} \tag{1}$$

$$\text{F}_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2}$$

We define the following: True positives (TPs) are found trace links between the SAD and the source code that are also contained in the gold standard. False positives (FPs) are found trace links that are not contained in the gold standard. False negatives (FNs) are trace links contained in the gold standard but not identified by the approach.

Additionally, we use two different averaging methods. First, we report the overall (macro) average across all projects without considering their size. This average provides useful insights into the expected performance on a per-project basis. Second, we calculate a weighted average based on the number of expected trace links in the gold standard [6]. This weighting offers more in-depth insights into the anticipated effectiveness of the approach for each trace link.

c) Large Language Models (LLMs): We use various LLMs to generate the simple SAMs. We decided to use both, closed-source models by OpenAI and locally deployed open-source models. For OpenAI, we use the following models: *GPT-4o mini*, *GPT-4o*, *GPT-4 Turbo*, *GPT-4*, and *GPT-3.5 Turbo*. As local models, we use *Codellama 13b*, *Meta AI Llama 3.1 8b*, and *Meta AI Llama 3.1 70b*.

B. Baselines

We reuse the baseline approaches of Keim et al. on TransArC [6]. Thus, we can directly compare their results to ours. The descriptions of the baselines are based on [6].

TAROT [14] and FTLR [46] are both recent, state-of-the-art IR-based solutions designed for linking requirements to code. CodeBERT [47] is an LLM trained to find the most semantically related source code for a given natural language description. Consequently, all three methods show promising results for similar TLR problems.

Keim et al. [6] also introduced the ArDoCode approach that uses heuristics to recover trace links between SAD and source code without using intermediate artifacts, that is based on the ArDoCo approach [8]. Keim et al. reported that, on average, ArDoCode was the best-performing approach that does not need SAMs. Thus, this approach is one of the important baselines. Lastly, we compare to the original TransArC approach [6] using (manually created) SAMs (cf. Section III-C).

TABLE II
RESULTS FOR TLR BETWEEN SAD AND CODE (SAM DERIVED FROM SAD)

	MS			TS			TM			BBB			JR			Avg.			w. Avg.		
Approach	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
TAROT	.09	.24	.13	.19	.44	.27	.06	.32	.11	.07	.18	.10	.32	**1.0**	.49	.15	.44	.22	.19	.63	.29
FTLR	.15	.26	.19	.19	.25	.21	.06	.30	.10	.04	.42	.07	.32	.93	.48	.15	.43	.21	.19	.59	.28
CodeBERT	.29	.12	.17	.26	.57	.36	.09	.22	.12	.07	.49	.12	.49	.83	.61	.24	.45	.28	.28	.53	.36
ArDoCode	.05	**.66**	.09	.20	**.74**	.31	.37	**.92**	.53	.07	.57	.13	.66	**1.0**	.80	.27	.78	.37	.47	.92	.62
TransArC	**1.0**	.52	**.68**	**1.0**	.71	**.83**	**.71**	.91	**.80**	**.77**	**.91**	**.84**	**.89**	**1.0**	**.94**	**.87**	**.81**	**.82**	**.81**	**.94**	**.87**
GPT-4o mini	.49	**.52**	.50	.95	.67	.78	**.71**	**.90**	**.80**	.71	.64	.68	**.89**	**1.0**	**.94**	.75	.75	.74	.80	.92	.85
GPT-4o	.49	**.52**	.50	**.96**	.67	.79	**.71**	**.90**	**.80**	.74	.77	**.75**	**.89**	**1.0**	**.94**	**.76**	.77	**.76**	**.81**	**.93**	**.86**
GPT-4 Turbo	.45	.40	.43	**.96**	.71	**.82**	**.71**	**.90**	**.80**	.71	.66	.69	**.89**	**1.0**	**.94**	.75	.73	.73	.80	.92	**.86**
GPT-4	.49	**.52**	.50	**.96**	.71	**.82**	**.71**	**.90**	**.80**	.63	**.84**	.72	**.89**	.99	**.94**	.74	**.79**	**.76**	.80	**.93**	**.86**
GPT-3.5 Turbo	.49	**.52**	.50	**.96**	.67	.79	**.71**	**.90**	**.80**	**.75**	.54	.63	**.89**	**1.0**	**.94**	**.76**	.73	.73	**.81**	.91	.85
Codellama 13b	**.81**	**.52**	**.63**	**.96**	.67	.79	.66	.49	.56	.05	.32	.08	**.89**	.99	**.94**	.67	.60	.60	.73	.72	.71
Llama 3.1 8b	.49	**.52**	.50	.21	.71	.33	.66	.34	.45	.73	.47	.57	**.89**	**1.0**	**.94**	.60	.61	.56	.75	.67	.68
Llama 3.1 70b	.48	.50	.49	.62	**.80**	.70	.62	.31	.41	.62	.43	.51	**.89**	**1.0**	**.94**	.65	.61	.61	.74	.66	.68

V. EMPIRICAL RESULTS

This section presents the results of our empirical evaluation on the effect of our LLM-extracted architecture component names on TLR between SAD and source code. We present the results to answer our research questions, including significance tests. Afterward, we discuss the results and analyze the errors of the LLMs. Finally, we discuss threats to validity.

A. Extracting Component Names from SAD

This section presents the results of our approach if we only use the SAD to generate the component names of a SAM. Table II provides a detailed overview of the results.

The table shows the precision, recall, and F_1-score for each project and the average values. The table consists of two sections, one for the baseline approaches and one for our approach using different LLMs. We highlight the overall best results per project in each section. In the first section, we took the results of the approaches as presented in the work of Keim et al. [6]. Here, TransArC uses the manually created SAM, while the other baseline approaches only use the SAD and source code. Overall, the best-performing baselines are ArDoCode and TransArC.

The second section of the table presents the results of our approach using different LLMs. We can see that the performance varies across the projects. We observe that for the weighted average F_1-score, the models by OpenAI perform particularly well. Furthermore, we can see that they perform similarly to TransArC without the need for manually created SAMs. The best model w.r.t. weighted average F_1-score is *GPT-4o* with a weighted average F_1-score of 0.86 compared to 0.87 of TransArC. The other OpenAI models perform similarly. According to the classification of Hayes et al. [44], our approach *excellently* recovers trace links between SAD and code (GPT-4o). We can highlight that all models outperform the baseline approaches that also do not use manually created SAMs, and thus, require the same input as our approach.

Significance Tests: In this section, we present the results of the significance tests. We use Wilcoxon's signed-rank test (one-sided) to calculate the statistical significance of our approach's F_1-score compared to the other baselines. Since our approach works best using GPT-4o as LLM, we only compare the results of this configuration to the baselines. We present the results in Table III. We mark those p-values with an asterisk that cannot be calculated exactly due to ties in the data. The table shows that our approach significantly outperforms all baselines that only use SAD and code. Yet, our approach does not outperform TransArC using manually created SAM ($p = 0.97$). At the same time, TransArC is also not significantly outperforming our LLMs-based approach ($p = 0.09$). This shows that our approach performs comparably to TransArC without the need for manually created SAMs.

TABLE III
RESULTS OF A ONE-SIDED WILCOXON SIGNED-RANK TEST REGARDING IF OUR APPROACH (GPT-4O) USING SAD FOR RECOVERY PERFORMS BETTER THAN THE BASELINE APPROACHES (SIGNIFICANCE LEVEL $\alpha = 0.05$, P-VALUES WITH * CANNOT BE CALCULATED EXACTLY.)

Approach / Hypothesis	Requires SAM	p-value	Significant
TAROT	No	.031	Yes
FTLR	No	.031	Yes
CodeBERT	No	.029*	Yes
ArDoCode	No	.031	Yes
TransArC (ours better)	Yes	.970*	No
TransArC (ours worse)	Yes	.091*	No

Conclusion RQ1: Applying TransArC with LLM-extracted SAMs produces similar results as with manually created SAMs.

Conclusion RQ2: Our approach significantly outperforms state-of-the-art TLR approaches between SAD and code that do not use SAMs as intermediates.

Conclusion RQ3: On average, OpenAI's closed-source LLMs perform better than the open-source Llama-based models in this task.

TABLE IV
RESULTS FOR TLR BETWEEN SAD AND CODE (SAM DERIVED FROM CODE)

	MS			TS			TM			BBB			JR			Avg.			w. Avg.		
Approach	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
GPT-4o mini	.11	.28	.16	.65	.35	.45	**.71**	.91	**.79**	.18	.30	.23	.94	.35	.51	.52	.44	.43	.77	.58	.60
GPT-4o	.19	**.52**	.28	.54	**.80**	.64	.68	.91	.78	.13	.11	.12	**1.0**	.52	.69	.51	.57	.50	.78	.67	.68
GPT-4 Turbo	**.87**	.40	**.55**	**.86**	.17	.28	.67	.23	.34	**1.0**	.00	.00	.89	.99	**.94**	**.86**	.36	.42	**.81**	.56	.59
GPT-4	.22	**.52**	.31	.79	.20	.32	.46	.91	.61	**1.0**	.00	.00	.83	.67	.74	.66	.46	.40	.68	.70	.61
GPT-3.5 Turbo	.14	**.52**	.22	.61	.76	**.68**	.60	**.92**	.72	.15	.16	.16	.82	.20	.32	.47	.51	.42	.67	.52	.49
Codellama 13b	.85	.34	.49	.00	.00	.00	.34	.02	.03	.06	.42	.10	.89	.99	**.94**	.43	.35	.31	.56	.50	.46
Llama 3.1 8b	.81	.26	.39	.85	.31	.45	.43	.02	.04	.00	.00	.00	.89	.99	**.94**	.60	.32	.37	.63	.48	.47
Llama 3.1 70b	.07	.28	.11	.54	**.80**	.64	.70	.80	.75	.36	**.64**	**.46**	.89	**1.0**	**.94**	.51	**.70**	**.58**	.76	**.88**	**.81**

TABLE V
F_1-SCORE FOR TLR BETWEEN SAD AND CODE (SAM DERIVED FROM SAD & CODE - AGGREGATED VIA PROMPT)

Approach	MS	TS	TM	BBB	JR	Avg.	w. Avg.
GPT-4o mini	.05	.62	.79	.34	.51	.46	.62
GPT-4o	.06	.58	.78	.42	**.94**	.56	.82
GPT-4 Turbo	.43	**.82**	**.80**	**.63**	**.94**	**.72**	**.85**
GPT-4	.31	.80	.62	.53	**.94**	.64	.77
GPT-3.5 Turbo	**.44**	.64	.74	.44	.93	.64	.80
Codellama 13b	**.44**	.56	.42	.13	**.94**	.50	.65
Llama3.1 8b	.43	.00	.00	.00	**.94**	.27	.44
Llama3.1 70b	.11	.64	.71	.22	**.94**	.52	.78

B. Generate Component Names from Source Code

This section focuses on **RQ4** and presents the results of our approach if we only use the source code to generate the component names. Table IV provides a detailed overview of the results. As described in Section III, we provide a list of all non-empty packages as features for the prompts. We argue that the packages can be a good representation of the high-level structure of a software project. Nevertheless, comparing the results to our approach using the SAD to extract the component names, the average performance using only source code is lower. Especially, the performance for the project *BBB* is bad, i.e., for many models, the F_1-score is 0 or close to 0. The overall best model is *Llama 3.1 70b* with a weighted average F_1-score of 0.81. Notably, this performance is better than its performance when using the SAD-extracted SAMs. Nevertheless, the overall average F_1-score is worse.

Since, on average, all other models perform worse than with SAD, we conclude that the packages alone are insufficient to generate SAMs for TLR. Moreover, this mode's performance is worse than the mode that only considers documentation. Further, the results vary even more across projects and LLMs.

C. Generate Component Names from SAD and Code

This section presents the results of our approach if we use both the SAD and the source code to generate the component names of the SAM. Here, we consider the two different aggregation strategies described in Section III-B: the LLM-based strategy and the similarity-based one. We use a similarity threshold of $t = 0.5$ in the experiments for the normalized Levenshtein distance. Nevertheless, this threshold can also be adjusted to the specific needs of a project.

a) Aggregation via Prompting: We present our results in F_1-score regarding the aggregation via prompting in Table V. The data shows that the best-performing model is *GPT-4 Turbo* with a weighted average F_1-score of 0.85. The results are slightly worse in macro average, compared to the mode that only uses the SAD but mostly better than the mode that only uses the source code.

b) Aggregation via Similarity: Next, we present the results of our evaluation of the aggregation via similarity. As described in Section III-B, we use the normalized Levenshtein distance to aggregate common names.

We present our results in Table VI. The aggregation is better than the results when the code is used only. The best-performing model is *GPT-4 Turbo* with a weighted average F_1-score of 0.86. Nevertheless, the aggregation via similarity also performs not better than the mode that only uses the SAD, particularly when considering the non-weighted average F_1-score. However, on average, the performance is often better than the aggregation via prompt.

Conclusion RQ4: On average, using only the SAD to generate the simple SAM performs best.

Overall, we conclude that to extract performing component names for TLR, the SAD is a better source than the source code. LLMs promise to have great language comprehension capabilities. Thus, we assume that the task of summarizing and extracting component names based on SAD is easier for the LLMs than inferring component names from source code.

D. Discussion & Error Analysis

This section discusses our results and analyzes some errors of the LLMs. First, we discuss exemplary differences between the extracted SAMs from the SAD using different LLMs. Second, we analyze exemplary differences between extracted SAMs from SAD and source code. Finally, we discuss the traceability from the SAD to the generated SAM.

a) Differences between extracted SAMs from SAD using different LLMs: As seen in Section V-A, the performance of the different LLMs varies across the projects. To provide insights into the reasons for that, we perform a detailed analysis on the *MediaStore* project with a particular focus on the difference between the results of *Codellama 13b* and

TABLE VI
RESULTS FOR TLR BETWEEN SAD AND CODE (SAM DERIVED FROM SAD & CODE - AGGREGATED VIA SIMILARITY ($t = 0.5$))

	MS			TS			TM			BBB			JR			Avg.			w. Avg.		
Approach	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
GPT-4o mini	.19	**.52**	.27	.68	.40	.51	**.71**	.91	.79	.35	**.81**	.49	.87	**1.0**	.93	.56	.73	.60	.75	.92	.82
GPT-4o	.19	**.52**	.28	.54	**.80**	.64	.68	.91	.78	.43	.73	.54	.81	**1.0**	.89	.53	**.79**	.63	.71	.93	.81
GPT-4 Turbo	.45	.40	.43	**.96**	.71	**.82**	**.71**	.90	**.80**	.71	.66	.69	**.89**	**1.0**	**.94**	**.75**	.73	**.73**	**.80**	**.92**	**.86**
GPT-4	.22	**.52**	.31	.93	.71	.80	.46	.91	.61	**.62**	.80	**.70**	**.89**	**1.0**	**.94**	.63	**.79**	.68	.69	.93	.78
GPT-3.5 Turbo	.14	**.52**	.22	.62	**.80**	.70	.60	**.92**	.72	.36	.57	.44	.88	**1.0**	.93	.52	.76	.60	.71	.92	.80
Codellama 13b	**.81**	**.52**	**.63**	.70	.47	.56	.65	.49	.56	.06	.42	.10	**.89**	.99	**.94**	.62	.58	.56	.72	.72	.70
Llama 3.1 8b	**.81**	**.52**	**.63**	.22	.75	.34	.64	.36	.46	.42	.47	.44	**.89**	**1.0**	**.94**	.60	.62	.56	.73	.68	.68
Llama 3.1 70b	.08	.34	.13	.54	**.80**	.64	.67	.91	.78	.34	.65	.45	**.89**	**1.0**	**.94**	.51	.74	.59	.74	.93	.82

GPT-4o. We picked this example because they show a rather large gap in precision (*Codellama 13b* with 0.81 vs 0.49 of *GPT-4o*) while having the same recall. The first column of Figure 2 shows the SAM extracted by *Codellama 13b*, the second column the component names of the original, manually created SAM, and the third column the SAM extracted by *GPT-4o*. While the SAMs are not the same, they share similar component names. The connections between the names indicate that we treat them as being similar. The highlighting of nodes indicate whether they are matched (green), not matched (red), or related (yellow). Both generated SAMs are missing components like *Cache* that are part of the manually created SAM but not described in the SAD. Besides that, the difference is that *Codellama 13b* generates a component *PersistenceTier* and *GPT-4o* generates a component called *DataStorage*. This small difference causes the drop in precision from 0.81 to 0.49 in the performance for the TLR task. Therefore, we also emphasize that according to model theory, the purpose of a model (e.g., an SAM) is important [48]. *Codellama 13b* creates component names that are more suitable for the considered TLR task. However, this does not mean that they are better for other tasks, and we do not assume that these SAMs can be directly used in reverse engineering tasks, as these tasks would require more details.

b) Differences between extracted SAMs from SAD and Code: In particular, we are interested in the actual differences between the extracted SAMs from the SAD and the source code. Since this requires manual analysis, we only focus on the projects with fewer components according to the manually created SAMs (cf. Table I), JabRef and TEAMMATES.

First, we analyze the differences between the extracted SAMs for the project *JabRef* using Llama 3.1 70b. Our evaluation shows that the performance regarding TLR is the same as TransArC using the manually created SAM. In Figure 3, we show the component names Llama 3.1 70b extracts. The first column defines the components extracted using only SAD, the second column the components of the manually created SAM, and the third column the components extracted from the source code. In the third column, we only show the elements that are extracted as main components. The LLM also generates a list of sub-components with no effects on performance. We use the same color scheme as in the previous sections. The names for the matching components of the modes only differ in their letter case, *globals* is missed by both modes, the SAD-extracted SAM additionally includes *EventBus*, and the Code-extracted adds *Networking*. The similar results in TLR suggest that these differences do not matter for the specific TLR task.

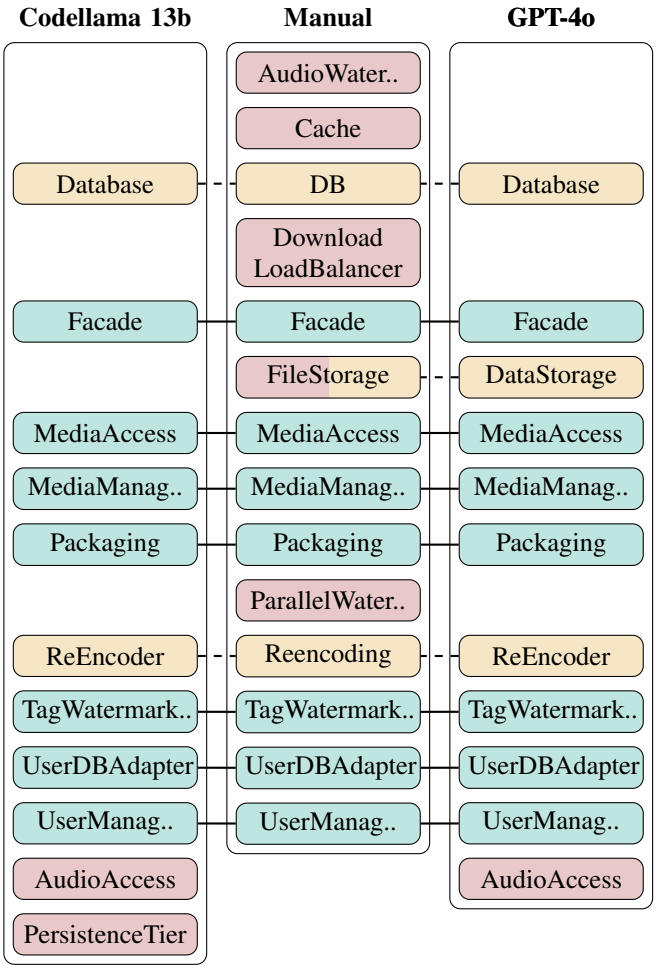

Fig. 2. Comparison of extracted SAMs for MediaStore using SADs

Second, we analyze the differences between the extracted SAMs for the project *TEAMMATES* using GPT-4 Turbo. Here, our evaluation shows that using SAD, the performance is comparable to TransArC, while using the source code, the performance is worse.

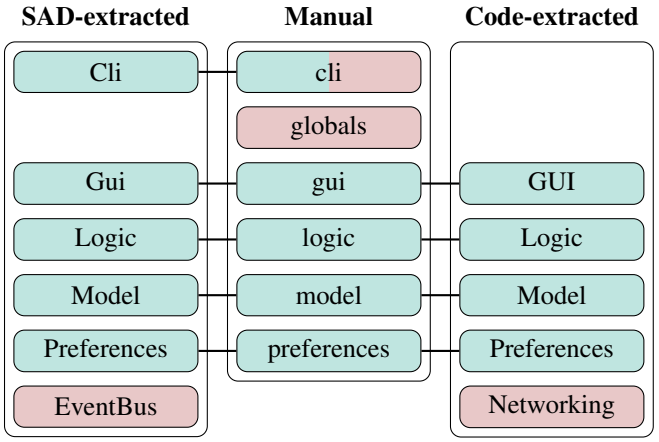

Fig. 3. Comparison of extracted SAMs for JabRef using Llama 3.1 70b

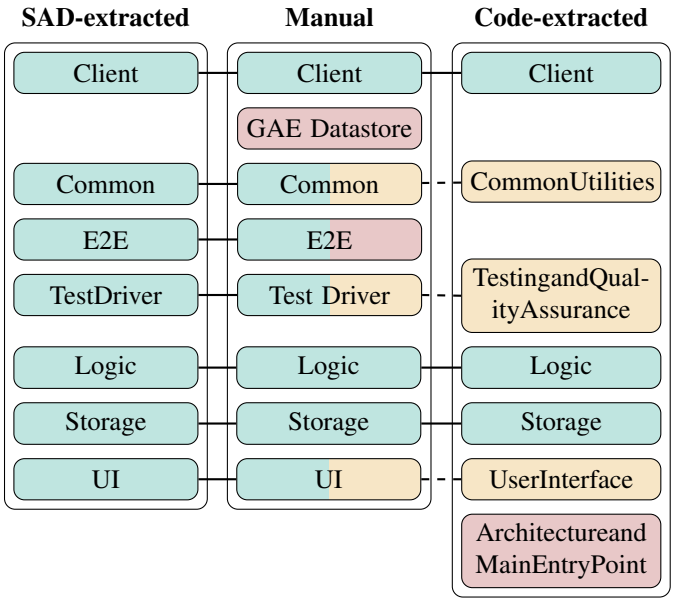

Fig. 4. Comparison of extracted SAMs for TEAMMATES using GPT-4 Turbo

In Figure 4, we show the component names GPT-4 Turbo extracts. We can see that our extracted SAM from SAD is only missing the *GAE Datastore* component. The extraction from the source code diverges more. Also, many components are correctly extracted, some components are abbreviated in the manually created SAM, while the LLM extracts the full name from the source code (e.g., *UserInterface*). However, the LLM also extracts additional components that are not present in the manually created SAM. The component *ArchitectureandMainEntryPoint* originates from the LLM's output "Architecture and Main Entry Point". Since we instructed the LLMs to provide component names in camel case, the final component name is as described. While there are the packages *teammates.architecture* and *teammates.main* in the source code, the LLM could not identify that these packages should not be mapped to a new component. The first package contains an architecture test and the second package contains the application's main entry point. The component *TestingandQualityAssurance* is also absent in the manually created SAM. While the source code contains packages for test cases, these are not part of the manually created SAM. We mark the components that are related to the manually created SAM with dashed lines. We assume that the testing-related components extracted via code belong to *Test Driver* component from the manually created SAM. This example shows that if we consider source code for extracting simple SAMs, distinguishing between code for production and code for testing is challenging.

c) Traceability from SAD to generated SAM: To discuss the performance of our approach, we briefly analyze the performance of the TLR approach from SAD to the generated SAM. Since we do not have a gold standard for this task, we decided to analyze our results using the following steps: First, we identify the components generated by the LLMs. Second, we manually match the components that are also present in the original SAM. Those components are assigned the same unique identifier as in the original SAM. All other components are assigned a new unique identifier. By doing so, we can use the evaluation of ArDoCo [8] to analyze the TLR from SAD to the generated SAM. Since we want to do this to get insights into the overall performance from SAD to source code, we only selected some projects and LLMs for analysis, based on our evaluation results in Table II.

We first analyze the SAD-generated SAM for *MediaStore* by *GPT-4o* (see Figure 2). We selected this model because the LLM is the overall best performing. Compared to the original SAM, the generated SAM contains 11 components, while the original SAM contains 14 components. It also contains two components that are not present in the original SAM: *AudioAccess* and *DataStorage*. Running ArDoCo with the generated SAM increases recall for the TLR between SAD and SAM from 0.62 to 0.79 compared to running with the manually created SAM. The precision decreases from 1.0 to 0.68. Our results for TLR from SAD to code also reflect this.

We then analyze the generated SAM for *BigBlueButton* by *Codellama 13b*. We selected this model because this is the best-performing open-source LLM, except for BBB. Comparing the generated SAM to the original SAM, the generated SAM contains 11 components, while the original SAM contains 12. The LLM additionally generates a component called *BigBlueButton*. Also, it slightly renamed the components *Apps* and *FSESL* to *AppsAkka* and *FSESLAkka*. The LLM could not generate the components *BBB web* and *Recording Service*. Also it generated names like *Joiningavoiceconference* that represent tasks but not a component name. Thus, the recall of the TLR from SAD to the generated SAM is 0.19 and the precision is 0.27. With the original SAM, the recall is 0.83 and the precision is 0.88. If we consider the package names of the code, one problem can be the generation of the *BigBlueButton* component. Since all Java packages of the project start with *org.bigbluebutton*, the component can lead to many false positive trace links that decrease precision.

E. Threats to Validity

In this section, we address potential threats to validity, drawing upon the guidelines provided by Runeson et al. [49]. We also consider the work of Sallou et al. [50], which discusses threats specific to experiments involving LLMs in the context of software engineering.

Regarding *construct validity*, one potential threat is dataset bias. To mitigate this, we use the same datasets as those employed in state-of-the-art approaches for TLR. By doing so, we ensure coverage of various domains and project sizes. Moreover, our study does not emphasize prompt engineering, meaning we do not extensively modify or compare different prompts. As a result, the selection of prompts potentially threatens the validity of our findings regarding our research questions. Another concern is the possible bias in metric selection. To address this, we employ commonly used metrics in TLR research. Overall, we strive to minimize confounding factors that could hinder our ability to address our research questions effectively.

Concerning *internal validity*, there is a risk that other factors influence our evaluation, potentially leading to incorrect interpretations or conclusions. We adhere to established practices for risk mitigation, utilizing datasets and projects from state-of-the-art approaches. We also clearly document the origins of these projects and the associated ground truths. Project quality and consistency variations may still impact the performance of our approach.

There are also potential threats concerning *external validity*. First, our evaluation is limited to a small number of projects and TLR tasks, which may affect the generalizability of the results to other projects and tasks. We utilize an established dataset covering various domains and project sizes to reduce this threat. However, these datasets mainly consist of open-source projects, which may differ from closed-source projects in key characteristics. Second, we evaluate the approach using only a limited set of LLMs, which may limit the robustness of our conclusions. Nonetheless, we evaluated our approach using various LLMs to show the performance of fixed prompts using different LLMs. A third concern is the use of closed-source models in evaluation. Since the training data of these models is unknown, we cannot entirely rule out data leakage. To ensure transparency, we provide our code, prompts, and LLM responses in a replication package [9]. Furthermore, we also use locally deployed models. Finally, the non-determinism of LLMs poses another threat. To mitigate this, we set the LLMs' temperature to zero and used the same fixed seed for each run. By doing so, we make sure that future research can replicate our results.

By conducting our evaluation on established datasets and using widely accepted metrics, we address the *reliability* of our research. Therefore, we do not introduce a threat regarding the manual creation of the gold standards or datasets. However, the existing gold standards might be imperfect.

VI. Conclusion and Future Work

In conclusion, this paper investigated the use of LLMs to bridge the semantic gap between SAD and code. The approach combines the strengths of LLMs and the TLR approach TransArC. Utilizing SAMs as intermediates TransArC is effective in TLR from SAD to code. Since SAMs are often unavailable, our approach leverages LLMs to extract component names from SAD and code. Thereby, we make approaches like TransArC more applicable in practice.

We have evaluated three modes for the generation and extraction of component names. We found that the extraction based on SAD performs best for the investigated TLR task. On weighted average, our approach performs comparably to TransArC using manually created SAMs (GPT-4o 0.86 vs. 0.87 TransArC) without the need for these provided SAMs. Additionally, our approach significantly outperforms baselines that use the same inputs, i.e., only SAD and code. On average, the best baseline without SAMs achieves an F_1-score of 0.37, while our approach achieves 0.76 (GPT-4o). We also identified that extracting component names for TLR performs better from SAD than from code. We show that simple SAMs recovered with smaller LLMs can enable TransArC to perform better than many state-of-the-art approaches. We assume that the semantic gap between component names and SAD is smaller and, thus, easier to bridge for the LLM. In particular, LLMs can internally perform some kind of named entity recognition on the SAD to identify the component names. This is not possible in the code, as the component names are not always directly visible. Future work may have a look into improving the code-based extraction of SAMs by using more sophisticated techniques from the research field of architecture recovery. We think that especially information fusion techniques [39] could help here.

In future work, we plan to investigate additional context for the LLMs to improve the TLR performance. This includes additional project artifacts, such as architecture decision records, architecture diagrams, or other parts of the SAD. Moreover, we want to analyze how separating tests from source code could improve the extraction of component names from the code. We also want to research how a different preprocessing of code artifacts can improve the extraction of SAMs for TLR. Here, code summarization techniques are one example that could be used to provide more context about the projects to the LLMs. Finally, future work could investigate how LLMs with integrated Chain-of-thought prompting like *OpenAI o1* perform on this task.

Data Availability Statement

We provide our code, baselines, evaluation data, and results in a replication package [9].

Acknowledgements

This work was funded by Core Informatics at KIT (KiKIT) of the Helmholtz Assoc. (HGF) and the German Research Foundation (DFG) - SFB 1608 - 501798263 and supported by KASTEL Security Research Labs.

References

[1] P. Rempel and P. Mäder, "Preventing defects: The impact of requirements traceability completeness on software quality," *IEEE Transactions on Software Engineering*, vol. 43, no. 8, 2017. DOI:10.1109/TSE.2016.2622264

[2] F. Tian, T. Wang, P. Liang, C. Wang, A. A. Khan, and M. A. Babar, "The impact of traceability on software maintenance and evolution: A mapping study," *Journal of Software: Evolution and Process*, vol. 33, no. 10, p. e2374, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2374. DOI:https://doi.org/10.1002/smr.2374

[3] T. W. W. Aung, H. Huo, and Y. Sui, "A literature review of automatic traceability links recovery for software change impact analysis," in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 14–24. DOI:10.1145/3387904.3389251

[4] K. Nishikawa, H. Washizaki, Y. Fukazawa, K. Oshima, and R. Mibe, "Recovering transitive traceability links among software artifacts," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2015, pp. 576–580. DOI:10.1109/ICSM.2015.7332517

[5] K. Moran, D. N. Palacio, C. Bernal-Cárdenas, D. McCrystal, D. Poshyvanyk, C. Shenefiel, and J. Johnson, "Improving the effectiveness of traceability link recovery using hierarchical bayesian networks," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 873–885. DOI:10.1145/3377811.3380418

[6] J. Keim, S. Corallo, D. Fuchß, T. Hey, T. Telge, and A. Koziolek, "Recovering Trace Links Between Software Documentation And Code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639130. DOI:10.1145/3597503.3639130

[7] J. H. Hayes, A. Dekhtyar, S. K. Sundaram, E. A. Holbrook, S. Vadlamudi, and A. April, "Requirements tracing on target (retro): improving software maintenance through traceability recovery," *Innovations in Systems and Software Engineering*, vol. 3, pp. 193–202, 2007.

[8] J. Keim, S. Corallo, D. Fuchß, and A. Koziolek, "Detecting inconsistencies in software architecture documentation using traceability link recovery," in *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, Mar. 2023, pp. 141–152. DOI:10.1109/ICSA56044.2023.00021

[9] D. Fuchß, H. Liu, T. Hey, J. Keim, and A. Koziolek, "Replication Package: Enabling Architecture Traceability by LLM-based Architecture Component Name Extraction," 2024. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.14506935. DOI:10.5281/ZENODO.14506935

[10] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, Oct. 2002. DOI:10.1109/TSE.2002.1041053

[11] T. Hey, F. Chen, S. Weigelt, and W. F. Tichy, "Improving Traceability Link Recovery Using Fine-grained Requirements-to-Code Relations," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 12–22. DOI:10.1109/ICSME52107.2021.00008

[12] H. Kuang, P. Mäder, H. Hu, A. Ghabi, L. Huang, J. Lü, and A. Egyed, "Can method data dependencies support the assessment of traceability between requirements and source code?" *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 838–866, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1736. DOI:10.1002/smr.1736

[13] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "When and How Using Structural Information to Improve IR-Based Traceability Recovery," in *2013 17th European Conference on Software Maintenance and Reengineering*, Mar. 2013, pp. 199–208. DOI:10.1109/CSMR.2013.29

[14] H. Gao, H. Kuang, K. Sun, X. Ma, A. Egyed, P. Mäder, G. Rong, D. Shao, and H. Zhang, "Using Consensual Biterms from Text Structures of Requirements and Code to Improve IR-Based Traceability Recovery," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, p. 1. DOI:10.1145/3551349.3556948

[15] A. D. Rodriguez, J. Cleland-Huang, and D. Falessi, "Leveraging intermediate artifacts to improve automated trace link retrieval," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 81–92. DOI:10.1109/ICSME52107.2021.00014

[16] K. Moran, D. N. Palacio, C. Bernal-Cárdenas, D. McCrystal, D. Poshyvanyk, C. Shenefiel, and J. Johnson, "Improving the effectiveness of traceability link recovery using hierarchical bayesian networks," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 873–885. [Online]. Available: https://doi.org/10.1145/3377811.3380418. DOI:10.1145/3377811.3380418

[17] J. Cámara, J. Troya, J. Montes-Torres, and F. J. Jaime, "Generative ai in the software modeling classroom: An experience report with chatgpt and unified modeling language," *IEEE Software*, vol. 41, no. 6, pp. 73–81, 2024. DOI:10.1109/MS.2024.3385309

[18] N. Meissner, S. Speth, and S. Becker, "Automated programming exercise generation in the era of large language models," in *2024 36th International Conference on Software Engineering Education and Training*, 2024, pp. 1–5. DOI:10.1109/CSEET62301.2024.10662984

[19] Z. Zhang, Z. Xing, X. Ren, Q. Lu, and X. Xu, "Refactoring to pythonic idioms: A hybrid knowledge-driven approach leveraging large language models," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3643776. DOI:10.1145/3643776

[20] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, Sep. 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3695988. DOI:10.1145/3695988

[21] D. Russo, S. Baltes, N. van Berkel, P. Avgeriou, F. Calefato, B. Cabrero-Daniel, G. Catolino, J. Cito, N. Ernst, T. Fritz, H. Hata, R. Holmes, M. Izadi, F. Khomh, M. B. Kjærgaard, G. Liebel, A. L. Lafuente, S. Lambiase, W. Maalej, G. Murphy, N. B. Moe, G. O'Brien, E. Paja, M. Pezzè, J. S. Persson, R. Prikladnicki, P. Ralph, M. Robillard, T. R. Silva, K.-J. Stol, M.-A. Storey, V. Stray, P. Tell, C. Treude, and B. Vasilescu, "Generative ai in software engineering must be human-centered: The copenhagen manifesto," *Journal of Systems and Software*, vol. 216, p. 112115, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121224001602. DOI:https://doi.org/10.1016/j.jss.2024.112115

[22] R. Dhar, K. Vaidhyanathan, and V. Varma, "Can llms generate architectural design decisions? - an exploratory empirical study," in *2024 IEEE 21st International Conference on Software Architecture (ICSA)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2024, pp. 79–89. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSA59870.2024.00016. DOI:10.1109/ICSA59870.2024.00016

[23] J. A. Díaz-Pace, A. Tommasel, and R. Capilla, "Helping novice architects to make quality design decisions using an llm-based assistant," in *Software Architecture*, M. Galster, P. Scandurra, T. Mikkonen, P. Oliveira Antonino, E. Y. Nakagawa, and E. Navarro, Eds. Cham: Springer Nature Switzerland, 2024, pp. 324–332.

[24] J. Cámara, L. Burgueño, and J. Troya, "Towards standarized benchmarks of llms in software modeling tasks: a conceptual framework," *Software and Systems Modeling*, Sep. 2024. [Online]. Available: http://dx.doi.org/10.1007/s10270-024-01206-9. DOI:10.1007/s10270-024-01206-9

[25] J. Cámara, J. Troya, L. Burgueño, and A. Vallecillo, "On the assessment of generative ai in modeling tasks: an experience report with chatgpt and uml," *Software and Systems Modeling*, vol. 22, no. 3, p. 781–793, May 2023. [Online]. Available: http://dx.doi.org/10.1007/s10270-023-01105-5. DOI:10.1007/s10270-023-01105-5

[26] A. Ahmad, M. Waseem, P. Liang, M. Fahmideh, M. S. Aktar, and T. Mikkonen, "Towards human-bot collaborative software architecting with chatgpt," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 279–285. [Online]. Available: https://doi.org/10.1145/3593434.3593468. DOI:10.1145/3593434.3593468

[27] T. Eisenreich, S. Speth, and S. Wagner, "From requirements to architecture: An ai-based journey to semi-automatically generate software architectures," in *Proceedings of the 1st International Workshop on Designing Software*, ser. Designing '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 52–55. [Online]. Available: https://doi.org/10.1145/3643660.3643942. DOI:10.1145/3643660.3643942

[28] J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang, "Traceability Transformed: Generating more Accurate Links with Pre-Trained BERT Models," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21. Madrid, Spain: IEEE Press, Nov. 2021, pp. 324–335. DOI:10.1109/ICSE43902.2021.00040

[29] A. D. Rodriguez, K. R. Dearstyne, and J. Cleland-Huang, "Prompts matter: Insights and strategies for prompt engineering in automated software traceability," in *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*, 2023, pp. 455–464. DOI:10.1109/REW57809.2023.00087

[30] J. Hassine, "An llm-based approach to recover traceability links between security requirements and goal models," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 643–651. [Online]. Available: https://doi.org/10.1145/3661167.3661261. DOI:10.1145/3661167.3661261

[31] M. North, A. Atapour-Abarghouei, and N. Bencomo, "Code gradients: Towards automated traceability of llm-generated code," in *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, 2024, pp. 321–329. DOI:10.1109/RE59067.2024.00038

[32] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger, "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 159–181, 2018. DOI:10.1109/TSE.2017.2671865

[33] V. Tzerpos and R. C. Holt, "Accd: an algorithm for comprehension-driven clustering," in *Proceedings Seventh Working Conference on Reverse Engineering*. IEEE, 2000, pp. 258–267.

[34] C. Xiao and V. Tzerpos, "Software clustering based on dynamic dependencies," in *Ninth European Conference on Software Maintenance and Reengineering*, 2005, pp. 124–133. DOI:10.1109/CSMR.2005.49

[35] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 552–555. DOI:10.1109/ASE.2011.6100123

[36] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *2011 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 35–44. DOI:10.1109/CSMR.2011.8

[37] C. Cho, K.-S. Lee, M. Lee, and C.-G. Lee, "Software architecture module-view recovery using cluster ensembles," *IEEE Access*, vol. 7, pp. 72 872–72 884, 2019. DOI:10.1109/ACCESS.2019.2920427

[38] Amarjeet and J. K. Chhabra, "Improving modular structure of software system using structural and lexical dependency," *Information and Software Technology*, vol. 82, pp. 96–120, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584916301951. DOI:https://doi.org/10.1016/j.infsof.2016.09.011

[39] Y. Zhang, Z. Xu, C. Liu, H. Chen, J. Sun, D. Qiu, and Y. Liu, "Software architecture recovery with information fusion," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1535–1547. [Online]. Available: https://doi.org/10.1145/3611643.3616285. DOI:10.1145/3611643.3616285

[40] S. A. Rukmono, L. Ochoa, and M. Chaudron, "Deductive software architecture recovery via chain-of-thought prompting," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER. New York, NY, USA: Association for Computing Machinery, 2024, p. 92–96. [Online]. Available: https://doi.org/10.1145/3639476.3639776. DOI:10.1145/3639476.3639776

[41] Z. T. Sinkala and S. Herold, "Hierarchical code-to-architecture mapping," in *Software Architecture*, P. Scandurra, M. Galster, R. Mirandola, and D. Weyns, Eds. Cham: Springer International Publishing, 2022, pp. 86–104.

[42] D. Fuchß, S. Corallo, J. Keim, J. Speit, and A. Koziolek, "Establishing a benchmark dataset for traceability link recovery between software architecture documentation and models," in *Software Architecture. ECSA 2022 Tracks and Workshops*, T. Batista, T. Bureš, C. Raibulet, and H. Muccini, Eds. Cham: Springer International Publishing, 2023, pp. 455–464. DOI:10.1007/978-3-031-36889-9_30

[43] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[44] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods," *IEEE TSE*, vol. 32, no. 1, pp. 4–19, Jan. 2006. DOI:10.1109/TSE.2006.3

[45] J. Cleland-Huang, O. Gotel, A. Zisman *et al.*, *Software and systems traceability*. Springer, 2012, vol. 2, no. 3. DOI:10.1007/978-1-4471-2239-5

[46] T. Hey, F. Chen, S. Weigelt, and W. F. Tichy, "Improving Traceability Link Recovery Using Fine-grained Requirements-to-Code Relations," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 12–22. DOI:10.1109/ICSME52107.2021.00008

[47] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. DOI:10.18653/v1/2020.findings-emnlp.139

[48] H. Stachowiak, *Allgemeine Modelltheorie*. Wien: Springer Verlag, 1973.

[49] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, p. 131, 2008. DOI:10.1007/s10664-008-9102-8

[50] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER'24. New York, NY, USA: Association for Computing Machinery, 2024, p. 102–106. [Online]. Available: https://doi.org/10.1145/3639476.3639764. DOI:10.1145/3639476.3639764