Retrieval-Augmented Generation (RAG): Strategies and possible Applications in Software Engineering

Seminar Thesis of

Jonas Strittmatter

Institute of Information Security and Dependability (KASTEL)
Advisor: M.Sc. Dominik Fuchß

The growing capabilities of Large Language Models (LLMs) have made them valuable across diverse domains. Retrieval-Augmented Generation (RAG) systems can adapt these models to specific contexts, such as working with existing code repositories to perform software engineering tasks. External data sources and documents, including knowledge graphs, can be used to augment LLMs by giving them access to knowledge not present in their training data. This seminar thesis explores various applications of RAG in software engineering, such as requirements traceability, code and documentation generation, test creation, and vulnerability detection. It also examines the challenges that current RAG techniques face.

Die wachsenden Fähigkeiten von Large Language Models (LLMs) machen sie in einer Vielzahl von Anwendungsbereichen nützlich. Retrieval-Augmented Generation (RAG)-Systeme können diese Modelle an spezifische Kontexte anpassen, wie beispielsweise die Arbeit mit bestehenden Code-Repositories, um Software-Engineering-Aufgaben zu erfüllen. Externe Datenquellen und Dokumente, darunter Wissensgraphen, können genutzt werden, um LLMs Zugang zu Wissen zu geben, welches nicht Teil ihrer Trainingsdaten ist. Diese Seminararbeit untersucht verschiedene Anwendungen von RAG im Software-Engineering, wie Rückverfolgbarkeit von Anforderungen, Code- und Dokumentationsgenerierung, Test-Erstellung, und Schwachstellenerkennung. Darüber hinaus werden die Herausforderungen analysiert, denen aktuelle RAG-Techniken ausgesetzt sind.

1 Introduction

The advent of Large Language Models (LLMs) has enabled performing complex tasks across a wide range of domains. Examples include AI-generated content in general [61], medical question-answering [53], and software engineering [25, 46]. State-of-the-art LLMs like GPT-4o, GPT-o1, or Google's recent Gemini Flash models are trained on enormous amounts of data, including literature, web articles, and code. A set of benchmarks such as SWE-bench [31] have been proposed to test the performance of LLMs across a variety of topics, including software engineering.

These benchmarks demonstrate a consistent increase in performance over time as more powerful models are released. With growing capabilities comes the desire to utilize these models in narrower contexts, such as specific business domains or individual code repositories.

However, to perform some tasks reliably, LLMs must have access to up-to-date and domain-specific knowledge. Without this knowledge, the relevance and accuracy of results cannot be guaranteed. To address this limitation, an LLM must have access to external documents. One possibility is to enrich the prompt given to the LLM with additional information relevant to the task, a process called Retrieval-Augmented Generation (RAG) [33].

In software engineering, many tasks require considering multiple components of a codebase rather than focusing on a single class or module. A naive approach of integrating an entire codebase into the model's context is impractical. First, most current LLMs have limited context windows – OpenAI's GPT-4o and o1 models, for instance, support a maximum of 128,000 tokens as of January 2025 [43]. Concretely, OpenAI currently supports at most 128.000 tokens for their GPT-4o and o1 models as of January 2025 [43]. Second, even with extensive context window sizes, such as Google's Gemini 1.5 Pro, which supports up to 2 million tokens [20], identifying the most relevant code snippets for a given query remains a significant challenge. Thus, effective retrieval mechanisms are necessary. Apart from the goal of increasing accuracy due to the availability of context-specific information, integration with external sources allows for novel applications, such as automated requirements tracing.

This thesis delves into the applications of RAG in software engineering, with a special focus on graph-based RAG approaches. The foundational concepts of knowledge graphs and their integration with LLMs are explored in Section 2 to establish the technical background for understanding graph-based RAG systems. Subsequently, the thesis presents several applications of Graph RAG within software engineering in Section 3. Covered topics include requirement analysis and traceability, design, code and documentation generation, unit test generation and software verification, as well as vulnerability detection. Moreover, the practical challenges and limitations of RAG with current LLMs are discussed in the fourth section. Finally, the key takeaways are summarized and potential areas of future research are given.

2 Foundations

This section introduces large language models and prompting techniques referenced in later parts, as well as retrieval-augmented generation and knowledge graphs as a preliminary step toward understanding graph-based retrieval augmented generation.

2.1 Large Language Models

Large Language Models are the primary technology behind recent advances in artificial intelligence systems like ChatGPT. Users can interact with an LLM using text, by verbal communication or via images depending on the modalities the model supports. The process of writing natural language instructions to influence a model's output is called prompting [59].

Effective prompting can have a great influence of the output quality of LLMs [47, 59]. Several paradigms have emerged that aim to increase the output quality of LLMs, including few-shot prompting, chain-of-thought (CoT), and tree-of-thought (ToT) [47]. Few-shot prompting refers to the strategy of including several examples of how to solve a task in the prompt. This contrasts with zero-shot prompting where no examples are given [7].

The CoT prompting strategy improves on few-shot prompting by including intermediate reasoning steps in the examples [52]. This is shown to increase performance across several tasks.

For complex tasks that require explorations or strategic lookahead, the previous prompting techniques fall short. Yao et al. [56] propose the tree of thoughts framework that encourages exploration over thoughts that serve as intermediate steps for general problem solving with LLMs. The LLM's ability to generate and evaluate thoughts is combined with search algorithm to enable a systematic exploration of thoughts.

2.2 Retrieval-Augmented Generation

LLMs such as ChatGPT are based on the Generative pre-trained transformer (GPT) architecture. As such, they are trained with large amounts of data during the training phase, which is computationally and energy-intensive. Although capable open-source models are available [22], the most powerful models such as OpenAI's GPT-4o or Anthropic's Claude Sonnet 3.5 are proprietary [40, 5]. Due to their proprietary nature, it is infeasible for external parties to update these models with the latest available knowledge or retrain them for improved accuracy on specific tasks. However, this leads to a loss of flexibility when applying the models in narrow application domains, where they often lack access to problem-specific knowledge. This limitation frequently results in hallucinations, a known issue with current models. Hallucinations refer to the observation that models will generate the most likely output to a prompt, even if the answer is incorrect or cannot be derived due to the lack of evidence in the data available to the LLM [57].

Retrieval-Augmented Generation (RAG) attempts to mitigate these issues. The term was first introduced by the authors of a 2020 paper titled "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" [33]. It refers to the process of retrieving relevant

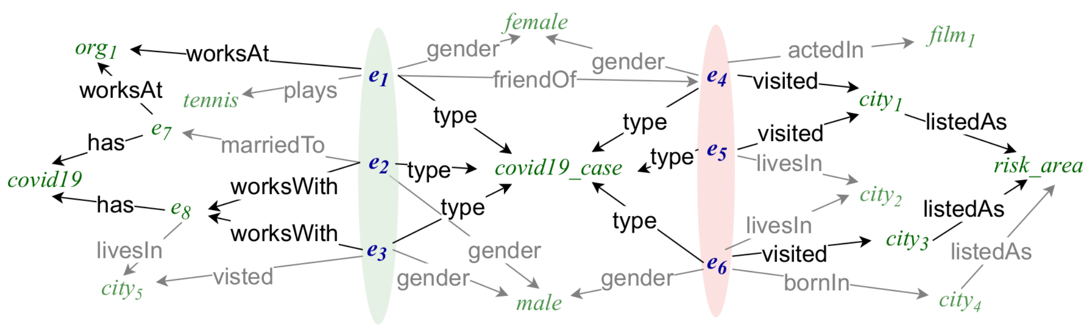

Figure 1: An example knowledge graph [18]

external knowledge and integrating it into the prompt given to the model to generate more accurate responses.

The survey paper by Gao et al. [19] categorizes common RAG paradigms into Naive RAG, Advanced RAG, and Modular RAG. Naive RAG employs a simple retrieve-read paradigm. It involves preprocessing external data into small chunks stored as vector embeddings in a vector database (indexing). Given a user query, the most relevant chunks are retrieved based on semantic similarity (retrieval) and are used to form the final prompt given to the LLM, which then produces an output (generation).

Advanced RAG refines this process by introducing steps before and after the retrieval stage. In the pre-retrieval stage, the original query is rewritten or expanded to clarify user intent. Additionally, the database index can be adjusted to better fit the given task. Retrieved chunks from the vector database are then reranked based on relevance and shortened to ensure that the model is not overloaded with information.

Jeong [28] lists guidelines for implementing RAG systems in practice. Proposals include trying out diferent indexing strategies (indexing-based vs. embeddings), applying query transformations or rewriting after unsuccessful retrievals, or using different indices per specialized for different types of questions.

2.3 Knowledge Graphs

Although there is no universally accepted definition [24], a knowledge graph (KG) can be seen as a graph designed to model knowledge about the real world [24]. In this graph, nodes correspond to entities of interest, and edges depict the relationships between these entities. In their simplest form, knowledge graphs can be stored in common relational databases as a set of triples of the form (subject, predicate, object). For example, the triple (Paris, isCapitalOf, France) represents the fact that Paris is the capital of France. For some use cases, one may want to enrich a node with additional metadata or properties, resulting in a property graph. Neo4j is a popular graph database that supports storing and querying such property graphs [16, 4].

Figure 1 depicts an example of a knowledge graph linking potential cases of Covid-19 to related information. Nodes consist of target entities e_1 - e_6 shown in green and red, locations such as $city_1$ or properties like *covid19_case*. The edges are labeled and describe relationships between nodes such as *worksWith* or *type*.

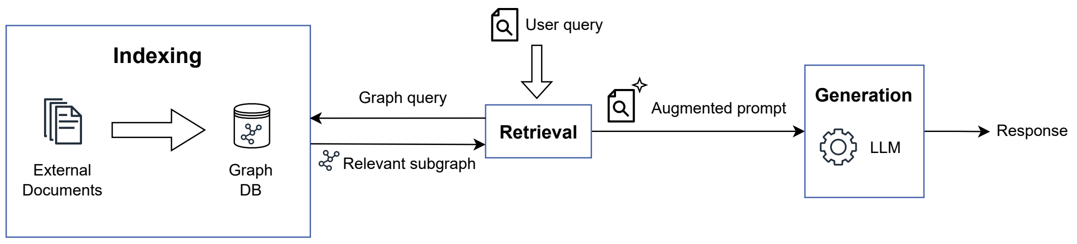

Figure 2: Graph-Based Retrieval-Augmented Generation, adapted from [19], [58]

Various KGs have been constructed to facilitate information retrieval or question-answering such as those used in Google's search engine [27], or social network analysis [1]. Wikidata [51] is a notable example of a publicly accessible knowledge graph that comprises more than 1.57 billion tuples. It simplifies the extraction of Wikipedia's data and allows users to perform queries on the data.

To construct knowledge graphs in practice, a multitude of approaches can be employed. Some knowledge graphs, such as Wikidata, are compiled and updated by contributors. However, ideally, the state of a knowledge graph should be synchronized with the underlying data base. This motivates the development of tools to automate both the initial construction and continuous updating of knowledge graphs.

In their journal article, Zhong et al. [62] survey more than 300 methods for constructing knowledge graphs automatically. They identify knowledge acquisition, refinement and evolution as the three main tasks of knowledge graph construction. Knowledge acquisition includes identification of entities in the original text – a process called Named Entity Recognition (NER) – along with entity typing and linking. Linking refers to fine-grained categorization of entities and connecting them to existing nodes in the knowledge graph. Coreference resolution is another step that tries to detect different terms that refer to the same entity. Finally, semantic relationships between entities are discovered, forming the edges in the knowledge graph.

The initial knowledge graph obtained after the first step can be incomplete. To improve it, missing relationships can be predicted using techniques such as embeddings and deep learning models. Other knowledge graphs can also be used for this task.

As a final step, knowledge evolution involves handling temporal or conditional changes in entities. A simple solution is to also store the timestamp of a tuple such as (Scholz, chancellor, Germany).

Trajanoska, Stojanov, and Trajanov [50] show that LLMs such as ChatGPT can be used to simplify the automatic creation of knowledge graphs. The authors utilize LLMs to perform named entity recognition, relation extraction, and entity linking. They compare relation extraction using the specialized model REBEL (Relation Extraction by End-to-End Language Generation) [26] but find that ChatGPT often fails to extract meaningful relationships, resulting in a less structured and larger knowledge base.

2.4 Graph-Based RAG

Knowledge graphs are shown to enhance the reasoning abilities of LLMs [29]. Microsoft Research introduces the term GraphRAG as an improvement over existing RAG systems for question answering in both a blog post [1] and a preprint paper [14]. Their approach comprises the phases of initial knowledge graph construction, hierarchical community detection within the graph, and modular summarization. An LLM is used for entity recognition and relation extraction, as described in Section 2.3 on knowledge graph construction. The nodes in the graph are entity summaries, which distinguishes their graph structure from other knowledge graphs. Additionally, GraphRAG uses weighted edges, where the weights are proportional to the number of detected relationships between node pairs. The edge weights facilitate community detection using the Leiden algorithm, which groups nodes that are more strongly connected to each other than to nodes in other parts of the graph. A summary of each community at different hierarchical levels is used to answer the query in parallel, resulting in the global answer given to the user.

In the following paragraphs, the term Graph RAG (GRAG) shall not refer to the original GraphRAG paper but to any RAG system that uses a graph in some form. Figure 2 visualizes the general structure of a GRAG system. It depicts the naive RAG paradigm as described in Section 2.2 (see [19]) but adjusts the indexing and retrieval parts. The external documents are stored in a graph database instead of a vector database. The exact structure of the constructed graph depends on the application. Nodes may be “discovered” (in the GraphRAG paper [14], nodes are summarized entities) or explicitly defined (e.g., in later described papers nodes are classes ([3]) or other code entities such as functions ([35])).

During retrieval, the graph database is queried to fetch relevant parts of the knowledge graph that help answering the user question.

A natural question to ask is how LLMs can query data structures such as knowledge graphs to answer user queries. StructGPT is a framework enabling LLMs to interface with structured data in the form of tables, databases consisting of multiple tables, and knowledge graphs [30]. The framework facilitates question answering on tables and knowledge graphs. To answer questions on a KG, the LLM can invoke two operations in their initial response: *Extract_Neighbor_Relations(e)* and *Extract_Triples(e, {r})*. These extract all outgoing neighboring relations of entity e as a list of edge names and all triples where e is the subject, respectively. The two operations are called in order and their results are linearized after each operation to formulate a prompt. The objects of the most relevant triples are selected as the final answer. The authors note some shortcomings of StructGPT, highlighting that errors can occur at several points during the retrieval and generation phases. Errors might occur during the retrieval of relevant relations or triples, leading to incomplete or irrelevant evidence for reasoning. Even with correct triples, the LLM sometimes struggles to generate accurate answers due to reasoning complexity or ambiguities in the question.

[1]`https://www.microsoft.com/en-us/research/blog/graphrag-unlocking-llm-discovery-on-narrative-private-data/`

Phase	Query	Discussed papers
Requirement analysis	*base query* AND ("requirement engineering" OR "requirements engineering" OR "requirement analysis" OR "requirements analysis" OR "requirement specification" OR "requirement traceability" OR "requirements traceability")	[3, 17, 38]
Design	*base query* AND ("software design" OR "system design" OR "program design" OR "architectural design" OR "system architecture")	
Implementation	*base query* AND ("implementation" OR "coding" OR "code generation" OR "documentation generation")	[35, 37]
Testing	*base query* AND ("test generation" OR "test case generation" OR "test automation" OR "software verification")	[48, 55]
Operation or maintenance	*base query* AND ("software maintenance" OR "bug fixing" OR "change management" OR "legacy systems" OR "software evolution" OR "system updates" OR "vulnerability detection")	[12]

Table 1: Phase in the waterfall model, Google Scholar search query, and papers discussed in the thesis. The *base query* is: ("software engineering") AND ("retrieval augmented generation" OR "RAG") AND "graph"

3 Applications of Graph RAG in Software Engineering

The tasks of a software engineer span requirements engineering, analysis, design, testing, debugging, and maintenance, as well as broader activities like project management, stakeholder communication, and environment administration [54]. These tasks encompass both technical work, such as coding and configuring software, and organizational responsibilities, such as team collaboration and server management.

In this section, the waterfall model serves as a framework to explore the applications of RAG in the major phases of software development. For each of the phases requirement analysis, design, implementation, testing and operation or maintenance [2], related papers are searched using the literature search engine Google Scholar.

Table 1 shows the queries for each phase and the number of results for each query as of February 2025. Only publications since 2022 are considered as November 2022 marks the initial release of GPT-3.5 [41].

It should be highlighted that the queries are not chosen to cover the full range of software engineering tasks. Rather, they are chosen to discover interesting applications of GRAG systems across the breadth of a typical development cycle. Additionally, some tasks such as ensuring the traceability of requirements might not only be practiced during a single phase in the waterfall model but be an ongoing effort. Still, it can be regarded as part of managing requirements in general. Thus it is included in the query for the first phase of the waterfall model.

For the design phase of the waterfall model, no papers are discussed in this thesis as the search query yielded no directly relevant results. One possible reason for this scarcity of research in this area could be the unstructured nature of design artifacts compared to artifacts in other phases. Unlike requirements, source code, or test cases, which often follow standardized formats can be represented as text, design and architecture artifacts are often expressed through informal diagrams. Therefore, it is harder to come up with an appropriate retrieval mechanism for these artifacts.

In the following subsections, papers are discussed for all phases except for the design phase.

3.1 Requirements Traceability

Requirement engineering involves eliciting functional and non-functional requirements that a system must meet to fulfill its intended purpose and satisfy stakeholder expectations [32]. Requirement traceability refers to the practice of maintaining the connections between individual requirements and their origins, related documentation, design, implementation, and testing artifacts throughout the project lifecycle [21]. These origins may include stakeholder needs, regulatory standards or technical specifications. Traceability is essential for change management, as it helps engineers assess the impact of modifications on the system architecture and components before implementation. Manually establishing and maintaining these connections is error-prone and impractical for large, dynamic codebases. Two recent conference papers [3, 17] and a preprint [38] are discussed that present RAG-based approaches to address this challenge. The GRAG approach by Ali, Naganathan, and Bork [3] is compared to the approach by Fuchß et al. [17] that uses RAG without a graph database.

The first approach by Ali, Naganathan, and Bork [3] called Retrieval Augmented Generation Requirements Traceability (RARTG) divides the process into indexing and querying stages, as shown in Figure 3. In the indexing stage, multiple indices are constructed from the code documents: a code documents keyword index, a vector index and a knowledge graph index. The code documents keyword index is created by preprocessing the code documents with stemming and stopword removal. These are stored in vector database to enable efficient querying later. To create the second vector index, a multi-lingual embedding model is used to encode the code documents into vector representations (emebddings) stored in a vector database for semantic similarity-based retrieval. A knowledge graph index represents relationships between classes and methods, where nodes correspond to class names (augmented with documentation strings, if available), and edges represent method calls, such as (ClassA, calls, ClassB). Nodes in the graph are class names coupled with existing documentation strings of that class if available.

In the retrieval stage, the constructed indices are used to retrieve relevant classes based on a requirement expressed in natural language. The requirement is inserted into a prompt template, which asks the LLM to identify related class names from the retrieved documents.

Linking Software System Artifacts (LiSSA) [17] is a RAG-based approach for traceability link recovery (TLR) that aims to be more general than the approach by Ali, Naganathan,

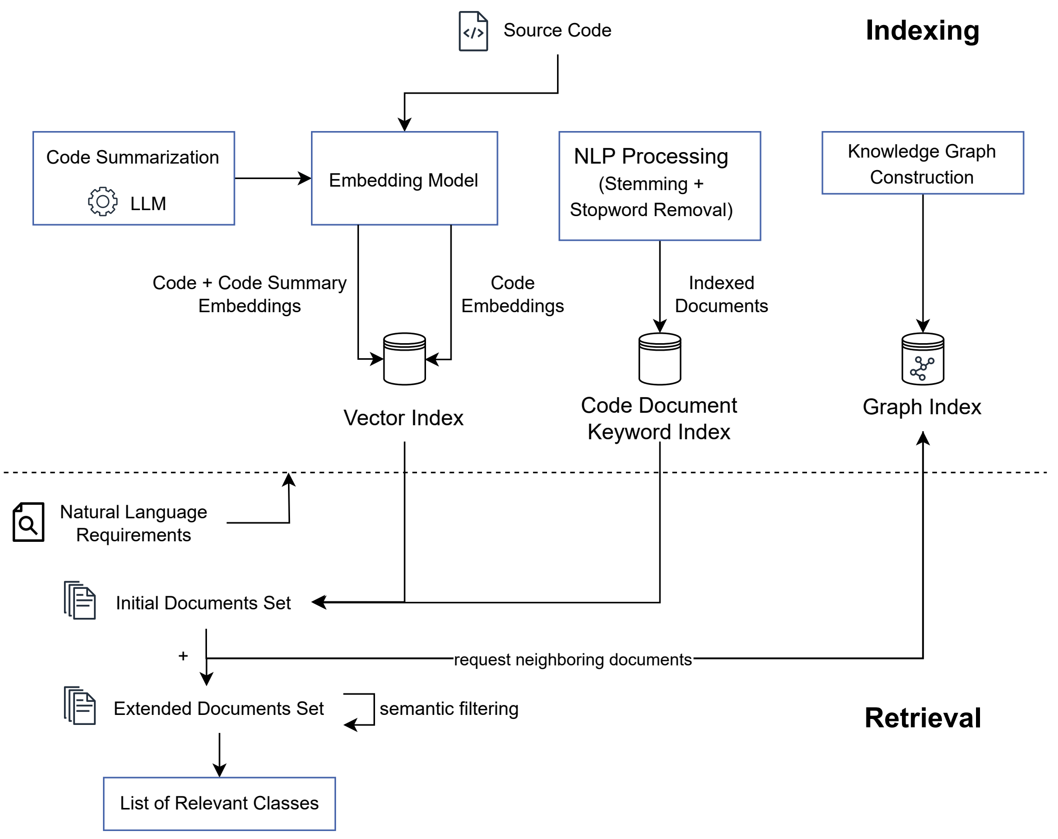

Figure 3: Retrieval Augmented Requirements Traceability Generation, adapted from [3]

and Bork [3]. It not only allows traceability of requirements to classes in the code but allows to trace other artifacts including source code, architecture documentation and architecture models as well. Different levels of granularity are supported for source and target elements, such as methods, classes and files for source code. All input elements are first transformed into a textual representation to facilitate embedding and storage into a vector database and later retrieval. The retrieved artifact pairs are formatted into prompts to an LLM to determine if a trace link exists between them. CoT prompting is compared to a simple yes-no question prompt.

Next, the two approaches are compared based on their published evaluation results. This comparison is feasible because both papers use the same benchmark datasets SMOS, iTrust and eTrust in their evaluation and report the metrics precision, recall, and the resulting F_1-score. These datasets, compiled by the Center of Excellence for Software & Systems Traceability (CoEST) [9], contain natural language requirements, source code, and trace links between them.

Figure 3.1 presents the evaluation results of both approaches on the three datasets. FTLR (Fine-Grained Traceability Link Recovery) [23] serves as a baseline, with FTLR_{OPT} representing an optimized configuration. RARTG-C1 refers to the configuration of RARTG

Approach \ Dataset	SMOS			iTrust			eTour		
	P	R	F_1	P	R	F_1	P	R	F_1
FTLR_{OPT}	.314	**.588**	.409	.234	.241	.238	.505	.597	**.548**
RARTG-C1	.608	.126	.209	**.289**	.292	**.290**	**.543**	.242	.334
LiSSA-C1	.479	.379	**.423**	.171	**.517**	.257	.107	.435	.172
LiSSA-C2	**.632**	.184	.285	.206	.493	**.290**	.378	.711	.493
LiSSA-C3	.590	.195	.294	.199	.451	.276	.409	**.734**	.526

Table 2: Precision, recall and F_1-scores for requirements to code traceability [3, 17]

[3] that achieves the highest F_1-score for the considered datasets. Additionally, three configurations of LiSSA [17] with the highest F_1-scores per project are displayed: LiSSA-C1 applies sentence-level granularity for the requirements, method-level granularity for the traced source code and a CoT prompt passed to the LLM. LiSSA-C2 performs no preprocessing, passing the entire artifact to the LLM while using a simple question-answer prompt. LiSSA-C3 also omits preprocessing but employs a CoT prompt that instructs the model to reason before answering. The results in Figure 3.1 show that LiSSA outperforms RARTG on the SMOS dataset in terms of F_1-score. On the iTrust dataset, the F_1-score of RARTG (0.290) is slightly higher or comparable to those of LiSSA. In contrast, LiSSA outperforms RARTG on the eTour dataset when using the C2 and C3 configurations. Across all datasets, LiSSA exhibits higher recall than RARTG, even for configurations not included in the table. Conversely, RARTG consistently achieves higher precision values for all projects. Specifically, for iTrust and eTour, precision is higher than all LiSSA variants evaluated in the paper [17].

In summary, while LiSSA outperforms RARTG on two of the three datasets, the results do not conclusively indicate a superior approach. Additionally, the results are highly sensitive to configuration parameters such as preprocessing granularity and prompting techniques. A configuration that performs well on one dataset does not necessarily generalize to another. Notably, among all three configurations, the LiSSA-C2 configuration achieves the highest F_1-scores on the iTrust and eTour datasets but the lowest on the SMOS dataset. This suggests that no universal solution exists; rather, configurations must be selected for each individual project.

The third preprint aims to help in automating the process of determining whether the set of requirements in a Software Requirements Specification (SRS) document adheres to a higher-level standard [38]. An initial graph is constructed from the higher-level standard which can then be queried to check for potential non-compliance of the SRS to the standard. The documents of the standard are chunked, entities and relationships are extracted and summarized using an LLM. Next, community detection and summarization is performed.

To evaluate their approach, the authors manually trace the requirements of the SRS documents of two projects. For each requirement, it is determined whether it is compliant, non-compliant or cannot be traced to the standard. During retrieval, the RAG system is tasked to reason on compliance to the standard based on a set of similar documents retrieved from the knowledge graph. The evaluation indicates that the graph RAG approach performs better than a baseline RAG system. However, there are several instances where the model misclassifies the requirements.

3.2 Code Generation

Automated code generation tools such as LLMs can help turn requirements into application code more quickly using natural language. To effectively integrate the new code into an existing codebase or infrastructure requires extensive knowledge about the domain, architecture, and coding standards.

The CodexGraph framework [35] attempts to facilitate this using graph-based RAG with a repository-level knowledge graph. The nodes of the graph represent code elements such as modules, classes, methods, functions, fields, and global variables that are obtained from the code. Nodes also contain metadata such as file names or code snippets. Edges describe the relationships between pairs of such elements and can be either "contains", "inherits", "has method", "has field", and "uses". While this schema of available node and edge types is specific to Python it can be adjusted to other languages that consist of different or additional kinds of code elements. The graph is stored in a Neo4j graph database that can be queried with the query language Cypher.

The code graph is constructed in two phases. In the first phase ("shallow indexing"), an initial graph is constructed in a single pass. As some relationships such as "inherits" cannot be directly inferred, the abstract syntax tree (AST) of each code file is further analyzed. This completes the code graph by inserting the missing edges.

Two specialized LLM agents faciliate querying. The primary LLM agent receives the original user query and transforms it into natural language instructions to the second LLM. An exemplary instruction may be "List all methods and global variables used by TaskManager". A second translation agent then devises Cypher graph queries from the instructions. The query is executed and relevant nodes and edges are retrieved. Multiple rounds of query generation may be performed if LLM agents determine that the retrieved context is still insufficient to answer the user's query. The CodexGraph approach shows increased benchmark performance compared to zero-shot prompting.

The RepoUnderstander [37] framework is able to resolve real-world GitHub issues using a graph RAG approach. During knowledge graph construction, the set of files, classes and functions present in the code base is first organized into a hierarchical tree. This tree is then extended into a reference graph that contains function call relationships. A unique aspect of RepoUnderstander is its use of Monte Carlo Tree Search to create multiple paths through the repository knowledge graph. As a path is expanded, the code snippet is given to an LLM that is asked to rate the relevance to solving a given issue. The most relevant paths are then used during generation.

3.3 Documentation Generation

Code documentation is a tangential facet of developing a system. Ideally, code documentation, both as comments within the code and as external documentation, should be complete and consistent [49]. However, keeping documentation up-to-date with the current state of the code can be time-consuming if done manually.

RepoAgent [36] is a framework to automate the generation and upkeep of repository-level documentation using a RAG approach. Initially, the source code files of the repository are individually parsed into an AST to obtain information about all classes and functions, including their types, names, and code snippets. This data is organized into a project tree that mirrors the repository's directory layout. The project tree is combined with bidirectional reference relationships between components, forming a directed acyclic graph.

During retrieval, relevant code from the project tree is fetched and given as context to the LLM. For each class and function, the LLM is instructed to generate Markdown documentation that includes detailed descriptions, parameter explanations, examples, and usage notes.

To address the problem of maintaining documentation, RepoAgent supports automatic updates when the code in a repository changes. Updates are localized to the parts of the codebase that have been modified. A human evaluation shows that the generated documentation is often preferred over human-written documentation.

3.4 Test Generation

Apart from application code generation, LLMs can also help create tests for existing code. In a recent preprint by Shin et al. [48], the authors measure the effects of RAG techniques on unit test generation compared to zero-shot prompting. They provide the model with different kinds of domain knowledge and ask it to generate unit tests for popular Python projects. The types are API documentation, GitHub issues, StackOverflow question and answers, and all of them combined. Four different LLMs (GPT-3.5-Turbo, GPT-4o, Mistral Mixture-of-Experts 8x22B, LLama 3.1 405B) are compared and the metrics AST parse rates, execution rates, pass rates and line coverage are collected.

The paper shows that OpenAI's model GPT 4o is able to achieve the highest parse and execution rates amongst the evaluated models. This coincides with the fact that this model is generally the most powerful model among the four [40]. Providing GPT-4o with additional documents largely improves execution rate and code coverage. However, for less powerful models such as the open-source Mistral model, injecting relevant documents into the prompt can even lead to a degradation of the metrics. The authors explain this with the fact that the lengthy additional documents provided to the models can distract them from focussing on the task of generating working unit tests.

One aspect where RAG seems particularly valuable is if the code to be tested requires further explanations or code examples. In that case, supplying GPT-4o with GitHub issues is shown to lead to coverage of code lines otherwise untested.

3.5 Fuzz Driver Generation

Fuzzing is a technique that can help uncovering vulnerabilities or bugs of an application or library by supplying it with unexpected or malformed data [39]. Various fuzzers exist that can create such inputs for a given program [34]. Fuzz drivers are components that define how input generated by a fuzzing tool interacts with the target software. It must ensure that functions to be tested are correctly invoked.

A recent preprint by Xu et al. [55] presents a tool called CKGFuzzer which is able to automate the fuzz driver generation by using GRAG. A knowledge graph is constructed for a given codebase using an AST of the code where nodes are either files, functions or external library functions. Edges can either indicate function calls or containment relationships for a file containing a function. Given an API to test, the KG is queried to generate relevant API combinations. Compared to testing an API individually, the aim is to increase code coverage by creating more complex execution paths. Based on the API combinations, an LLM is asked to generate a fuzz driver. If the resulting driver code is not executable, it is iteratively refined by an LLM. A notable detail is that a knowledge base is maintained that persists across iterations to store correct API usage patterns.

3.6 Vulnerability Detection

The last phase of the software development process in the waterfall model is concerned with operating and maintaining the software produced in the previous phases. One aspect of maintenance is the detection and removal of vulnerabities in the code.

The authors of the Vul-RAG paper [12] show how a RAG system combined with a knowledge base of known vulnerabilities can be used to perform vulnerability detection in a codebase. The knowledge base is constructed by analyzing how existing vulnerabilities in the Linux kernel were fixed. An LLM is instructed to extract three aspects from each vulnerability: the functionality of the code where the vulnerability is observed, the causes of the vulnerability and a change to the code that fixes the vulnerability. These are summarized and inserted into the knowledge base.

Given a code snippet to analyze for potential vulnerabilities, a second LLM (here: GPT-4) first retrieves knowledge on vulnerabilities in functionally similar code from the knowledge base. Using this knowledge on causes of the vulnerabilities and a solution to fix it, the LLM is asked to determine whether the given code is vulnerable and explain its reasoning.

In their evaluation, they find that the explanation given by Vul-RAG helps developers better understand a given vulnerability in a code snippet compared to no additional system. Further, it outperforms GPT-4 that does not use RAG, and a GPT-4 RAG system that can find similar code snippets but does not include additional vulnerability knowledge. This highlights the usefulness of supplying a RAG system with additional knowledge. There are instances in the dataset where out of the three systems, only Vul-RAG is able to correctly identify the root cause of a vulnerability in the given code. However, there are also instances where Vul-RAG is unable to precisely explain the reason why a code snippet is vulnerable.

4 Current Limitations of Graph RAG

The premise of extending RAG systems with knowledge graphs is to improve the accuracy of query answering in narrow or context-specific domains compared to RAG without knowledge graphs. Yet the current approaches have indicated a range of shortcomings that limit the applicability of RAG systems to real-world tasks such as software engineering.

In this section, four limitations of current (G)RAG systems are presented based on an aggregation and analysis of issues found in the literature.

Insufficient Data Quality. The effectiveness of a RAG system heavily depends on the quality of retrieved data [15]. Many of the presented works list concrete cases where the low quality of data leads to incorrect answers. For instance, the VulRAG paper mentions a case where an imprecise vulnerability description lead to a false negative [12]. Apart from incomplete knowledge, highly similar documents that are not directly relevant to the query can harm the effectiveness of a RAG system [10]. However, the inclusion of random documents is unexpectedly shown to increase accuracy.

Cost vs. Output Quality. A general decision one must make when designing a RAG system is the choice of the model to use. Several of the discussed works mention this tradeoff [38, 48, 45]. Less powerful models such as GPT-4o-mini are typically more cost effective than models with increased reasoning capabilities such as GPT-o1 ($0.60 / 1M. output tokens for GPT-4o-mini vs. $60 / 1M. output tokens for GPT-o1 [44]). The authors of [48] also note that the input token cost can vary significantly depending on the structure of data injected into the prompt.

Unstructured Responses. Popular conversational models such as ChatGPT are inherently flexible in their style of response. Although this adaptability makes these models useful in a variety of domains and scenarios, it poses challenges in applications that require precise output that adheres to a specific syntax. This is particularly important when interacting with other software components that cannot parse unstructured text. For example, one may want to further process the answer given by an LLM in response to a request to provide the most relevant source code documents (see Section 3.1).

Some work has been done on addressing this issue. In a web article, OpenAI describes an API feature called Structured Outputs [42] that makes the model respond in the common data exchange format JSON. Users can provide a JSON schema that determines the structure of the output to which the model must adhere. To achieve this, a technique called constrained decoding is employed. During generation, the model generates a response by sampling arbitrary tokens. With constrained decoding, the set of possible tokens the model can choose from is restricted to follow a fixed structure formalized by a context-free grammar that is constructed based on the schema. Using this technique, conformance to a JSON schema can be guaranteed.

Privacy Concerns. Applying (graph) RAG in a business context requires careful attention to the protection of data. A paper by Bruckhaus notes that enterprises dealing with sensitive customer or patient data must ensure that any employed RAG systems must comply with security and privacy regulations [8].

Depending on company policy, some data cannot be shared with external parties, such as OpenAI, via platforms like the ChatGPT website or their API. This rules out the use of current state-of-the-art models such as GPT-4o. Instead, self-hosted, local models, such as LLama 3.1 [22] may be used, although they are less powerful than their proprietary alternatives.

Additionally, Zeng et al. [60] demonstrate that RAG systems are susceptible to data extraction attacks. These attacks aim to retrieve sensitive documents accessible to the LLM during the retrieval phase. Thus, it is important to ensure that the documents do not contain any sensitive data as they could be revealed to users by careful prompting.

Explainability. Explainable AI (XAI) is an active research field concerned with understanding the output and reasoning of AI models [13]. In the context of RAG systems, explainability means being able to identify the documents that were used to generate the final answer. This is particularly critical for highly-regulated indutries such as healthcare, law or finance [6]. Bruckhaus argues that in these industries, a RAG's system output must be explainable and interpretable to allow businesses to trust the RAG systems [8].

5 Conclusion and Future Developments

This thesis explored Retrieval-Augmented Generation and possible applications in software engineering across the various phases of the waterfall model, ranging from requirements traceability, code generation, documentation generation, test and fuzz-driver generation to vulnerability detection. Compared to using LLMs in isolation, a RAG approach offers several advantages, such as increased accuracy without the need for fine-tuning. In Section 3.1, a comparison between a RAG approach and a GRAG approach on the topic of requirements traceability was conducted. The findings indicate that augmenting a RAG implementation with a graph does not necessarily yield superior results.

As more powerful LLMs continue to be developed, their enhanced reasoning capabilities enable new applications. However, due to the novelty of the research field, many presented approaches lack generalization. Several papers only work for specific formats of the input data [38, 55, 36]. The discussion on limitations further showed that trade-offs exist regarding cost of operation, and that ensuring high quality of data is an important aspect for making RAG work well in practice. Deploying RAG system in a business context requires consideration of privacy concerns and, ideally, explainability of results. The recent release of capable open-source models, such as DeepSeek-V3 that rivals state-of-the-art proprietary models such as GPT-o1 [11] hints at the possibility that the dependence on major technological companies such as OpenAI, Google, or Meta could shrink.

Looking ahead, future LLMs could ingest large parts of the codebase as context, without leading to hallucinations. The techniques presented in the thesis could be combined into a unified system capable of automating common software engineering tasks, eliminating the need for specialized individual solutions.

References

[1] Bilal Abu-Salih et al. "Relational Learning Analysis of Social Politics using Knowledge Graph Embedding". In: *Data Mining and Knowledge Discovery* 35.4 (July 1, 2021), pp. 1497–1536. ISSN: 1573-756X. DOI: `10.1007/s10618-021-00760-w`.

[2] 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 (Sept. 10, 2020), pp. 427–434.

[3] Syed Juned Ali, Varun Naganathan, and Dominik Bork. "Establishing Traceability Between Natural Language Requirements and Software Artifacts by Combining RAG and LLMs". In: *Conceptual Modeling*. Ed. by Wolfgang Maass et al. Cham: Springer Nature Switzerland, 2025, pp. 295–314. ISBN: 978-3-031-75872-0. DOI: 10.1007/978-3-031-75872-0_16.

[4] Renzo Angles et al. "Foundations of Modern Query Languages for Graph Databases". In: *ACM Comput. Surv.* 50.5 (Sept. 26, 2017), 68:1–68:40. ISSN: 0360-0300. DOI: 10.1145/3104031.

[5] Anthropic. *Introducing Claude 3.5 Sonnet.* URL: https://www.anthropic.com/news/claude-3-5-sonnet (visited on 12/21/2024).

[6] Alejandro Barredo Arrieta et al. "Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI". In: *Information Fusion* 58 (June 1, 2020), pp. 82–115. ISSN: 1566-2535. DOI: 10.1016/j.inffus.2019.12.012.

[7] Tom Brown et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems.* Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.

[8] Tilmann Bruckhaus. *RAG Does Not Work for Enterprises.* May 31, 2024. DOI: 10.48550/arXiv.2406.04369. arXiv: 2406.04369[cs].

[9] Center of Excellence for Software and Systems Traceability (CoEST). *Datasets.* URL: http://sarec.nd.edu/coest/datasets.html (visited on 02/11/2025).

[10] Florin Cuconasu et al. "The Power of Noise: Redefining Retrieval for RAG Systems". In: *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval.* SIGIR '24. New York, NY, USA: Association for Computing Machinery, July 11, 2024, pp. 719–729. ISBN: 979-8-4007-0431-4. DOI: 10.1145/3626772.3657834.

[11] DeepSeek-AI et al. *DeepSeek-V3 Technical Report.* Dec. 27, 2024. DOI: 10.48550/arXiv.2412.19437. arXiv: 2412.19437[cs].

[12] Xueying Du et al. *Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG.* June 19, 2024. DOI: 10.48550/arXiv.2406.11147. arXiv: 2406.11147[cs].

[13] Rudresh Dwivedi et al. "Explainable AI (XAI): Core Ideas, Techniques, and Solutions". In: *ACM Comput. Surv.* 55.9 (Jan. 16, 2023), 194:1–194:33. ISSN: 0360-0300. DOI: 10.1145/3561048.

[14] Darren Edge et al. *From Local to Global: A Graph RAG Approach to Query-Focused Summarization.* Apr. 24, 2024. DOI: 10.48550/arXiv.2404.16130. arXiv: 2404.16130.

[15] Wenqi Fan et al. "A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models". In: *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining.* KDD '24. New York, NY, USA: Association for Computing Machinery, Aug. 24, 2024, pp. 6491–6501. ISBN: 979-8-4007-0490-1. DOI: 10.1145/3637528.3671470.

[16] Diogo Fernandes and Jorge Bernardino. "Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB". In: International Conference on Data Science, Technology and Applications. Vol. 2. SCITEPRESS, July 26, 2018, pp. 373–380. ISBN: 978-989-758-318-6. DOI: `10.5220/0006910203730380`.

[17] Dominik Fuchß et al. *LiSSA: Toward Generic Traceability Link Recovery through Retrieval-Augmented Generation.* ISSN: 1558-1225. 2025. DOI: `10.5445/IR/1000178348`. URL: `https://publikationen.bibliothek.kit.edu/1000178348` (visited on 02/06/2025).

[18] Mohamed H. Gad-Elrab et al. "ExCut: Explainable Embedding-Based Clustering over Knowledge Graphs". In: *The Semantic Web – ISWC 2020.* Ed. by Jeff Z. Pan et al. Cham: Springer International Publishing, 2020, pp. 218–237. ISBN: 978-3-030-62419-4. DOI: `10.1007/978-3-030-62419-4_13`.

[19] Yunfan Gao et al. *Retrieval-Augmented Generation for Large Language Models: A Survey.* Mar. 27, 2024. DOI: `10.48550/arXiv.2312.10997`. arXiv: `2312.10997[cs]`.

[20] Google. *Long context | Gemini API.* Google AI for Developers. URL: `https://ai.google.dev/gemini-api/docs/long-context` (visited on 01/30/2025).

[21] O.C.Z. Gotel and C.W. Finkelstein. "An analysis of the requirements traceability problem". In: *Proceedings of IEEE International Conference on Requirements Engineering.* Proceedings of IEEE International Conference on Requirements Engineering. Apr. 1994, pp. 94–101. DOI: `10.1109/ICRE.1994.292398`.

[22] Aaron Grattafiori et al. *The Llama 3 Herd of Models.* Nov. 23, 2024. DOI: `10.48550/arXiv.2407.21783`. arXiv: `2407.21783[cs]`.

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

[24] Aidan Hogan et al. "Knowledge Graphs". In: *ACM Comput. Surv.* 54.4 (July 2, 2021), 71:1–71:37. ISSN: 0360-0300. DOI: `10.1145/3447772`.

[25] Xinyi Hou et al. "Large Language Models for Software Engineering: A Systematic Literature Review". In: *ACM Trans. Softw. Eng. Methodol.* 33.8 (Dec. 3, 2024), 220:1–220:79. ISSN: 1049-331X. DOI: `10.1145/3695988`.

[26] Pere-Lluís Huguet Cabot and Roberto Navigli. "REBEL: Relation Extraction By End-to-end Language generation". In: *Findings of the Association for Computational Linguistics: EMNLP 2021.* Findings 2021. Ed. by Marie-Francine Moens et al. Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 2370–2381. DOI: `10.18653/v1/2021.findings-emnlp.204`.

[27] *Introducing the Knowledge Graph: things, not strings.* Google. May 16, 2012. URL: `https://blog.google/products/search/introducing-knowledge-graph-things-not/` (visited on 11/30/2024).

[28] Cheonsu Jeong. *A Study on the Implementation Method of an Agent-Based Advanced RAG System Using Graph.* Sept. 13, 2024. DOI: 10.48550/arXiv.2407.19994. arXiv: 2407.19994.

[29] Jinhao Jiang et al. *KG-Agent: An Efficient Autonomous Agent Framework for Complex Reasoning over Knowledge Graph.* Feb. 17, 2024. DOI: 10.48550/arXiv.2402.11163. arXiv: 2402.11163.

[30] Jinhao Jiang et al. *StructGPT: A General Framework for Large Language Model to Reason over Structured Data.* Oct. 23, 2023. DOI: 10.48550/arXiv.2305.09645. arXiv: 2305.09645.

[31] Carlos E Jimenez et al. "SWE-bench: Can language models resolve real-world github issues?" In: *The twelfth international conference on learning representations.* 2024.

[32] Axel van Lamsweerde. "Requirements engineering in the year 00: a research perspective". In: *Proceedings of the 22nd international conference on Software engineering.* ICSE '00. New York, NY, USA: Association for Computing Machinery, June 1, 2000, pp. 5–19. ISBN: 978-1-58113-206-9. DOI: 10.1145/337180.337184.

[33] Patrick Lewis et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in Neural Information Processing Systems.* Vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474.

[34] Hongliang Liang et al. "Fuzzing: State of the Art". In: *IEEE Transactions on Reliability* 67.3 (Sept. 2018). Conference Name: IEEE Transactions on Reliability, pp. 1199–1218. ISSN: 1558-1721. DOI: 10.1109/TR.2018.2834476.

[35] Xiangyan Liu et al. *CodexGraph: Bridging Large Language Models and Code Repositories via Code Graph Databases.* Aug. 11, 2024. DOI: 10.48550/arXiv.2408.03910. arXiv: 2408.03910.

[36] Qinyu Luo et al. *RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation.* Feb. 26, 2024. DOI: 10.48550/arXiv.2402.16667. arXiv: 2402.16667[cs].

[37] Yingwei Ma et al. *How to Understand Whole Software Repository?* June 3, 2024. DOI: 10.48550/arXiv.2406.01422. arXiv: 2406.01422.

[38] Arsalan Masoudifard et al. *Leveraging Graph-RAG and Prompt Engineering to Enhance LLM-Based Automated Requirement Traceability and Compliance Checks.* Dec. 11, 2024. DOI: 10.48550/arXiv.2412.08593. arXiv: 2412.08593[cs].

[39] P. Oehlert. "Violating assumptions with fuzzing". In: *IEEE Security & Privacy* 3.2 (Mar. 2005). Conference Name: IEEE Security & Privacy, pp. 58–62. ISSN: 1558-4046. DOI: 10.1109/MSP.2005.55.

[40] OpenAI. *Hello GPT-4o.* URL: https://openai.com/index/hello-gpt-4o/ (visited on 12/20/2024).

[41] OpenAI. *Introducing ChatGPT.* Nov. 30, 2022. URL: https://openai.com/index/chatgpt/ (visited on 01/20/2025).

[42] OpenAI. *Introducing Structured Outputs in the API.* URL: https://openai.com/index/introducing-structured-outputs-in-the-api/ (visited on 12/19/2024).

[43] OpenAI. *OpenAI Platform.* URL: https://platform.openai.com (visited on 01/30/2025).

[44] OpenAI. *Pricing.* URL: https://openai.com/api/pricing/ (visited on 01/25/2025).

[45] Siru Ouyang et al. *RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph.* Oct. 3, 2024. DOI: 10.48550/arXiv.2410.14684. arXiv: 2410.14684.

[46] Ipek Ozkaya. "Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications". In: *IEEE Software* 40.3 (May 2023). Conference Name: IEEE Software, pp. 4–8. ISSN: 1937-4194. DOI: 10.1109/MS.2023.3248401.

[47] Pranab Sahoo et al. *A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications.* Feb. 5, 2024. DOI: 10.48550/arXiv.2402.07927. arXiv: 2402.07927[cs].

[48] Jiho Shin et al. *Retrieval-Augmented Test Generation: How Far Are We?* Sept. 19, 2024. DOI: 10.48550/arXiv.2409.12682. arXiv: 2409.12682.

[49] Diomidis Spinellis. "Code Documentation". In: *IEEE Software* 27.4 (July 2010). Conference Name: IEEE Software, pp. 18–19. ISSN: 1937-4194. DOI: 10.1109/MS.2010.95.

[50] Milena Trajanoska, Riste Stojanov, and Dimitar Trajanov. *Enhancing Knowledge Graph Construction Using Large Language Models.* May 8, 2023. DOI: 10.48550/arXiv.2305.04676. arXiv: 2305.04676.

[51] Denny Vrandečić and Markus Krötzsch. "Wikidata: a free collaborative knowledgebase". In: *Commun. ACM* 57.10 (Sept. 23, 2014), pp. 78–85. ISSN: 0001-0782. DOI: 10.1145/2629489.

[52] Jason Wei et al. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". In: *Advances in Neural Information Processing Systems* 35 (Dec. 6, 2022), pp. 24824–24837.

[53] Junde Wu et al. *Medical Graph RAG: Towards Safe Medical Large Language Model via Graph Retrieval-Augmented Generation.* Oct. 15, 2024. DOI: 10.48550/arXiv.2408.04187. arXiv: 2408.04187.

[54] Włodzimierz Wysocki, Ireneusz Miciuła, and Marcin Mastalerz. "Classification of Task Types in Software Development Projects". In: *Electronics* 11.22 (Jan. 2022). Number: 22 Publisher: Multidisciplinary Digital Publishing Institute, p. 3827. ISSN: 2079-9292. DOI: 10.3390/electronics11223827.

[55] Hanxiang Xu et al. *CKGFuzzer: LLM-Based Fuzz Driver Generation Enhanced By Code Knowledge Graph.* Dec. 20, 2024. DOI: 10.48550/arXiv.2411.11532. arXiv: 2411.11532[cs].

[56] Shunyu Yao et al. "Tree of Thoughts: Deliberate Problem Solving with Large Language Models". In: *Advances in Neural Information Processing Systems* 36 (Dec. 15, 2023), pp. 11809–11822.

[57] Hongbin Ye et al. *Cognitive Mirage: A Review of Hallucinations in Large Language Models*. Sept. 13, 2023. DOI: `10.48550/arXiv.2309.06794`. arXiv: `2309.06794[cs]`.

[58] Hao Yu et al. "Evaluation of Retrieval-Augmented Generation: A Survey". In: *Big Data*. Ed. by Wenwu Zhu et al. Singapore: Springer Nature, 2025, pp. 102–120. ISBN: 978-981-9610-24-2. DOI: `10.1007/978-981-96-1024-2_8`.

[59] J.D. Zamfirescu-Pereira et al. "Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts". In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. New York, NY, USA: Association for Computing Machinery, Apr. 19, 2023, pp. 1–21. ISBN: 978-1-4503-9421-5. DOI: `10.1145/3544548.3581388`.

[60] Shenglai Zeng et al. *The Good and The Bad: Exploring Privacy Issues in Retrieval-Augmented Generation (RAG)*. Feb. 23, 2024. DOI: `10.48550/arXiv.2402.16893`. arXiv: `2402.16893[cs]`.

[61] Penghao Zhao et al. *Retrieval-Augmented Generation for AI-Generated Content: A Survey*. June 21, 2024. DOI: `10.48550/arXiv.2402.19473`. arXiv: `2402.19473`.

[62] Lingfeng Zhong et al. "A Comprehensive Survey on Automatic Knowledge Graph Construction". In: *ACM Comput. Surv.* 56.4 (Nov. 30, 2023), 94:1–94:62. ISSN: 0360-0300. DOI: `10.1145/3618295`.