Technische Universität München

TUM School of Engineering and Design

Lehrstuhl für Computergestützte Modellierung und Simulation

Using Graphs to Improve the Interpretability of API Documentation for BIM Authoring Software.

Master Thesis

For the Master of Science study program Information Technologies for the Built Environment

Author:	Tamira Wrabel
StudentID:	
Supervisor:	Dr. Sebastian Esser
	Changyu Du

Date of Issue:

Date of Submission:

Abstract

This master thesis investigates the use of Knowledge Graphs (KGs) to improve the interpretability and usability of API documentation for Building Information Modeling (BIM) authoring tools. By integrating KGs with Large Language Models (LLMs), the research addresses challenges in understanding complex and unstructured API documentation, contributing to enhanced workflows for developers in the construction industry. Set within the framework of BIM and its APIs, this study combines qualitative and quantitative methodologies to construct a graph-based structure for API documentation. The primary objective is to transform unstructured data into a queryable format using a Knowledge Graph and evaluate its effectiveness when integrated with an LLM-based Retrieval-Augmented Generation (RAG) Agent. The method was test with the Vectorworks API, and a question set comprised of 36 questions.

The method demonstrates that deterministic graph construction methods ensure reliable relationships and well-defined nodes, facilitating accurate querying and information retrieval. Furthermore, it highlights the flexibility of LLM-embedding-based graph construction, which adapts to unstructured and incomplete documentation. It shows that developing a graph-based RAG Agent can effectively answer user queries by leveraging the semantic richness of the constructed graphs. Comparative analysis reveals that a hybrid graph combining a deterministic and embedded node generation approach outperforms exclusively LLM-generated graphs in text accuracy and code suggestion reliability.

This research bridges the gap between developers and advanced BIM tools by offering a novel approach to API documentation interpretation. By integrating KGs with LLMs, the study provides a robust framework for improving productivity and decision-making in BIM workflows. Limitations such as dependence on data quality and computational intensity of embeddings are recognized, paving the way for future investigations of hybrid graph construction methods. This study advocates further exploration of KGs and LLMs to improve the interpretability and usability of BIM API in the construction domain.

Key Terms: Knowledge Graphs, Large Language Models, BIM APIs, Retrieval-Augmented Generation, graphRAG

Zusammenfassung

Diese Masterarbeit untersucht die Verwendung von Wissensgraphen (Knowledge Graphs, KGs) zur Verbesserung der Interpretierbarkeit und Nutzbarkeit von API-Dokumentation für Autorenwerkzeuge für Building Information Modeling (BIM). Durch die Integration von KGs mit Large Language Models (LLMs) werden die Herausforderungen beim Verständnis komplexer und unstrukturierter API-Dokumentation angegangen, was zu verbesserten Arbeitsabläufen für Entwickler in der Bauindustrie beiträgt. Im Rahmen von BIM und seinen APIs kombiniert diese Studie qualitative und quantitative Methoden, um eine graphenbasierte Struktur für API-Dokumentation zu erstellen. Das Hauptziel ist die Umwandlung unstrukturierter Daten in ein abfragbares Format unter Verwendung eines Wissensgraphen und die Bewertung seiner Effektivität bei der Integration mit einem LLM-basierten Retrieval-Augmented Generation (RAG) Agent. Die Methode wurde mit der Vectorworks API und einem Fragesatz von 36 Fragen getestet.

Die Methode zeigt, dass deterministische Graphen Konstruktionsmethoden verlässliche Beziehungen und Knoten gewährleisten, was die genaue Abfrage und das Abrufen von Informationen erleichtert. Darüber hinaus wird die Flexibilität der LLM-embedding-basierten Graphenkonstruktion hervorgehoben, die sich an unstrukturierte und unvollständige Dokumentation anpasst. Es wird gezeigt, dass die Entwicklung eines graphenbasierten RAG-Agenten Benutzeranfragen effektiv beantworten kann, indem er den semantischen Reichtum der konstruierten Graphen nutzt. Eine vergleichende Analyse zeigt, dass ein hybrider Graph, der einen deterministischen und einen eingebetteten Knotengenerierungsansatz kombiniert, ausschließlich LLM-generierte Graphen in Bezug auf Textgenauigkeit und Zuverlässigkeit von Codevorschlägen übertrifft.

Diese Forschung überbrückt die Lücke zwischen Entwicklern und fortschrittlichen BIM-Tools, indem sie einen neuartigen Ansatz zur Interpretation der API-Dokumentation bietet. Durch die Integration von KGs mit LLMs bietet die Studie einen robusten Rahmen zur Verbesserung der Produktivität und Entscheidungsfindung in BIM-Workflows. Einschränkungen wie die Abhängigkeit von der Datenqualität und die Aussagekraft von Einbettungen werden erkannt und ebnen den Weg für zukünftige Untersuchungen hybrider Graphenkonstruktionsmethoden. Diese Studie befürwortet die weitere Erforschung von KGs und LLMs, um die Interpretierbarkeit und Nutzbarkeit der BIM-API im Baubereich zu verbessern.

List of Contents

List of Abbreviations

API	Application Programming Interface
BIM	Building Information Modeling
GPT	Generative Pre-trained Transformer
GraphRAG	Graph Retrieval-Augmented Generation
IFC	Industry Foundation Classes
JSON	JavaScript Object Notation
KG	Knowledge Graph
LLM	Large Language Model
LPG	Labeled Property Graph
RAG	Retrieval-Augmented Generation
RDF	Resource Description Framework
URI	Uniform Resource Identifier
XML	eXtensible Markup Language

1 Introduction

1.1 Motivation

The rapid development of digital technologies has fundamentally changed the construction industry, with Building Information Modeling (BIM) emerging as the cornerstone of the innovations. BIM has revolutionized the way architects, engineers, and contractors collaborate by enabling advanced 3D modeling, improving decision-making processes, and streamlining project execution by combining all relevant data in one model. The technique of 3D modeling has become increasingly established in the daily planning and construction business. Another significant aspect of BIM is that it offers, apart from the 3D geometrical modeling aspect, the opportunity to enrich the modeled elements with extensive semantic information. Leading software companies offer BIM authoring tools that combine geometric and semantic modeling, such as Autodesk Revit[1], Graphisoft Archicad[2], Nemetschek Vectorworks,[3] and Allplan[4]. Those tools improve the design process by reducing the planning time, indicating clashes in planning before even coming to the construction site, and offering a platform for collaboration. Connecting geometric data with semantic knowledge makes multidisciplinary design activities and site scheduling more time-efficient and visually understandable.

Despite the advantages of these advancements, the complexity of these tools has led to new challenges. Many architects and engineers need to learn to work with the latest tools, which is considered an extra effort most users resist. Furthermore, the user interface of those tools is quite complex, and understanding all functions takes in-depth training. However, developers who add new functionality to the software or develop individual features also have difficulties understanding the application programming interfaces (APIs) documentation as they are as complex as the software itself. Nevertheless, the integration of programming interfaces has transformed the usability of the software itself, not only in BIM authoring software. Tools such as visual programming applications, script-based systems, and plug-in development environments enable the

[1] Atuodesk Revit: https://www.autodesk.com/de/products/revit/architecture
[2] Graphisoft Archicad: https://graphisoft.com/de/archicad
[3] Nemetschek Vectorworks: https://www.vectorworks.net/en-US/fundamentals
[4] Nemetschek Allplan: https://www.allplan.com/de/

customization of workflows to the specific needs of the user. One example is Dynamo[5] for Revit from Autodesk, which provides a visual programming interface that utilizes the functions of the Revit API in a simplified visual way. Although detailed, API documentation from BIM tools is often inconsistent, unstructured, and difficult to interpret. Developers struggle to extract relevant information and find connections, leading to inefficiencies in creating extensions and automation.

As in many other sectors, the emergence of AI is another significant change the construction industry has experienced in recent years (Abioye et al., 2021). Large Language Models (LLMs) demonstrated the ability to process and generate human-like text. They can address similar challenges across industries by facilitating natural language interactions and summarizing complex information. However, LLMs often fail to provide accurate answers and correct code implementations when applied to domain-specific scenarios. To address these challenges, this research proposes integrating Knowledge Graphs (KGs) with LLMs to enhance the interpretability and usability of BIM API documentation. This would support developers in writing new plugins as it would be easier to find relevant functions and retrieve coding suggestions for the specific functions. Knowledge graphs enable structure data with nodes and relationships, offering insights into connections between functions previously unseen. This study aims to develop a knowledge graph structure for API documentation to enable intelligent queries and provide better coding suggestions. This is achieved by extracting the data from the BIM-API into a BIM-API-Graph, which then can be utilized by a graph-based Retrieval-Augmented Generation (RAG) Agent to answer users' questions about the implementation of the API precisely.

1.2 Research objectives

This thesis explores the intersection of Knowledge Graphs, Large Language Models, and BIM APIs to tackle challenges associated with interpreting API documentation. The explicit target is to develop a comprehensive approach that enables developers to process and transform API documentation into Knowledge Graphs and use these graphs as a basis for a large language model (LLM). This approach aims to improve

[5] Dynamo BIM: https://dynamobim.org/

the graph-based RAG agent's answers regarding the developer's questions and provide correct coding implementation examples.

The research aims to answer the following questions:

- What kind of knowledge graph schema design can effectively represent BIM API documentation?
- What are the comparative advantages of Knowledge Graphs over raw textual data when utilized as a knowledge base for an LLM-based RAG Agent in the context of BIM API queries?
- What challenges emerge while transforming BIM API documentation into a Knowledge Graph, and how can these be addressed?

By addressing these objectives, the study seeks to bridge the gap between developers and the increasingly complex tools they use, providing a practical solution to improve their workflows and productivity.

1.3 Reading guide

This thesis is structured in the following chapters:

- Chapter 2 – Theoretical Background: overviews knowledge graphs, LLMs, RAG and Graph RAG methods, BIM APIs, and their applications in the built environment.
- Chapter 3 – Current State-of-the-art literature reviews existing research on LLMs, knowledge graphs, and their applications in BIM workflows.
- Chapter 4 – Methodology outlines the process of constructing knowledge graphs from BIM API documentation and the chatbot's design.
- Chapter 5 – Case Study showcases the methodology on a real-world example.
- Chapter 6 – Results and Analysis presents the outcomes of the knowledge graph generation and the chatbot implementation and evaluation of the results.
- Chapter 7 – Discussion examines the implications of the findings and evaluates the research objectives.
- Chapter 8 – Conclusion and Future Works summarizes the key contributions and outlines potential directions for future research.

2 Theoretical Background

This chapter presents the theoretical background for the methods used in this study. The technologies' characteristics are briefly outlined, and the methods relevant to this study are introduced. Possible advantages and disadvantages are also briefly mentioned. The relevant topics are large language models (LLMs), knowledge graphs (KG), Retrieval Augmented Generation (RAG) and Graph Retrieval-Augmented Generation (Graph-RAG) methods, and BIM APIs.

2.1 Knowledge Graph Characteristics and Use Cases in the Built Environment

Knowledge graphs have emerged as powerful tools for representing data in a structured and interconnected way. Semantic graphs capture information with entities (nodes) and relationships (edges), enabling advanced querying and reasoning capabilities over classical serialized data representation standards. Knowledge graphs build on classical techniques such as JSON and XML but go beyond their capabilities by addressing inherent limitations.

JSON[6] (JavaScript Object Notation) is a lightweight, text-based format that stores data using key-value pairs. It is widely used for transmitting data between servers and clients in web applications due to its simplicity and human-readability (W3Schools, 2025a). XML[7] (eXtensible Markup Language), similar to HTML but with no predefined tags, provides a more flexible and hierarchical structure for encoding documents (W3Schools, 2025b). Both formats are well-suited for hierarchical data representation and serialization, making them suitable for many applications. However, they struggle to capture complex relationships and interconnected data effectively. This limitation becomes particularly evident in domains like the built environment, where data is highly interdependent, requiring advanced relation reasoning and semantic context. Knowledge graphs address these challenges by representing data as entities (nodes)

[6] https://www.w3schools.com/whatis/whatis_json.asp
[7] https://www.w3schools.com/XML/xml_whatis.asp

and their connections (edges). Therefore, KGs are central to improving data interoperability, semantic enrichment, and supporting collaboration in the built environment (Pauwels et al., 2022).

There are two primarily used knowledge graph types enabling those features: Labeled Property Graphs (LPG) (Needham & Hodler, 2019) and the Resource Description Framework (RDF) graphs. RDF[8] is a standard from W3C for exchanging data on the Web (W3C, 2025b). The graphs capture data using a triple structure (subject-predicate-object), and each triple is identified by a Uniform Resource Identifier (URI). Such a URI is built from the subject-predicate-object structure. An example of an RDF graph can be seen on the left side of Figure 1. RDF graphs can be stored in triple-stores and queried using SPARQL[9] (W3C, 2025a). While they offer high semantic expressiveness and interoperability, their strict schema can be limiting in rapidly changing environments (Pauwels et al., 2022).

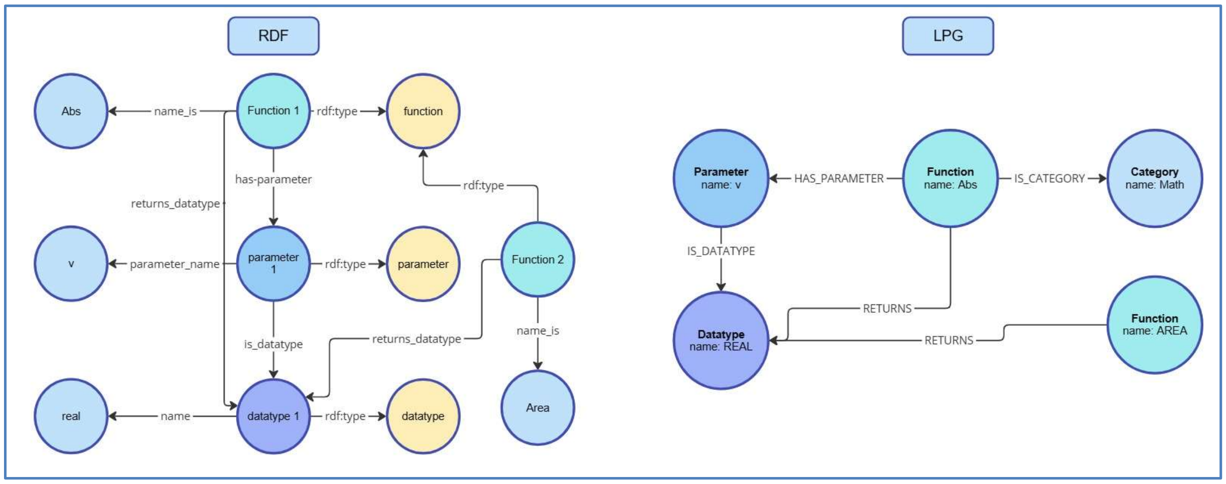


Figure 1 Comparison of the structure of an RDF graph with an LPG graph

In comparison, LPGs are designed to be flexible and are typically used schema-free, providing more adaptability to changes in the information captured in the graph. The nodes and edges can be labeled and defined by properties, drastically reducing the number of nodes and edges (Barrasa & Webber, 2023). An example of an LPG graph can be seen on the right side of Figure 1. The graphs consist of nodes connected by relationships, classified by relationship types. Properties are attributes for nodes and

[8] RDF: https://www.w3.org/TR/rdf12-concepts/
[9] SPARQL: https://www.w3.org/TR/sparql12-query/

relationships, giving extra information, and are stored as key-value pairs (Needham & Hodler, 2019). LPG graphs can be queried using the query language Cypher[10].

Knowledge graphs are relevant to the built environment as they can enhance interoperability and enable semantic enrichment. KGs facilitate advanced reasoning and enable users to uncover hidden insights. Using graph-based queries, retrieving specific information becomes more efficient. Approaches showcasing the application of semantic graphs in the domain will be described in Chapter 3.2.

This thesis aims to utilize the strengths of knowledge graphs, particularly LPGs, to transform unstructured BIM API documentation into a structured, queryable format.

2.2 Characteristics of LLMs

Large Language Models (LLMs) have significantly improved in recent years, supporting users by answering diverse questions across various topics. However, their knowledge strongly depends on the scope and quality of their training data. As a result, specialized domains like Building Information Model (BIM) are not captured to the full extent. To address this limitation, knowledge graphs have proven to be an effective tool for grounding LLMs in specific subject areas. Moreover, LLMs can also simplify information retrieval from knowledge graphs, as they can provide user-friendly access to data, eliminating the need for the user to understand complex query languages.

LLMs have redefined natural language processing (NLP) by enabling machines to process, understand, and generate human-like text (Jeon & Lee, 2025). These models are based on deep learning principles and are trained on large-scale datasets using unsupervised learning to solve tasks like text generation, summarization, and question-answering (Ibrahim et al., 2024).

A structured overview and understanding of LLMs is essential to fully leverage the potential benefits of integrating KG with LLMs. The following section briefly examines the fundamental mechanism of transformers, the differences between open- and closed-source LLMs, and the categorization of these models. An overview of this chapter is shown in Figure 2.

[10] Cypher: https://neo4j.com/product/cypher-graph-query-language/

The foundation of an LLM is transformer architecture, which enables contextual learning from texts, as examined in detail by Vaswani et al. (2017). The attention mechanism, central to the transformer architecture, computes a weighted representation of the input sequence by dynamically assessing the relevance of each word to the task at hand. Specifically, the models calculate attention scores by comparing query, key, and value vectors derived from the input tokens. In Scaled Dot-Product Attention, these scores are scaled by the square root of the key vector's dimension and normalized using a Softmax function to generate attention weights, a key component of the mechanism described by Vaswani et al. (2017). This process allows the model to focus on relationships between tokens regardless of their position in the sequence, effectively capturing local and global dependencies. Multi-head attention further enhances this capability by projecting the input into multiple subspaces, enabling the model to simultaneously learn diverse aspects of the sequence. Through empirical evaluation, Vaswani et al. demonstrated the transformer's ability to achieve state-of-the-art performance in tasks requiring deep contextual understanding, such as natural language generation and translation.

Most currently used LLMs are based on this transformer design, containing encoder and decoder modules. They can be categorized into three categories, as visualized in Figure 2: decoder-only, encoder-only, and encoder-decoder LLMs (Pan, 2024).

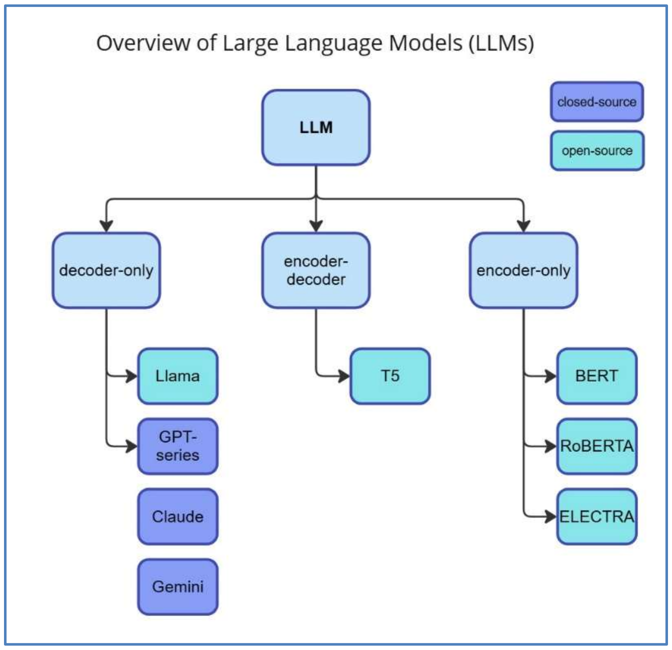


Figure 2 - Overview of Large Language Models (LLMs)

Encoder-only LLMs rely on the encoder to process input sentences and capture relationships between words. They are pre-trained using unsupervised methods. These models are optimized for comprehension-focused tasks like text classification. Examples of open-source encoder-only LLMs are BERT (Devlin et al., 2018), RoBERTa (Liu et al., 2019), and ELECTRA (Clark et al., 2020).

Encoder-decoder LLMs utilize both encoder and decoder components. They encode input sentences into a hidden representation, which the decoder uses to generate the output text. Use cases for these models are summarization, translation, and question-answering tasks. Examples based on this approach are open-source LLMs like T5 (Raffel et al., 2019) and GLM-130B (Zeng et al., 2022).

The third part is the decoder-only LLMs, which use the decoder module exclusively to generate target output text. These models are pre-trained to predict the next word in a sequence and are used to create text from context. Examples of closed-source decoder-only LLMs include OpenAI’s Generative Pre-trained Transformer (GPT) series (OpenAI, 2025), Google’s Gemini (Mallick & Korevec, 2024), and Claude (Antropic, 2025). They can perform downstream tasks using simple instructions without finetuning. An example of an open-source decoder-only LLM is Meta’s LLaMA (Ollama, 2025a).

This thesis utilizes decoder-only LLMs for its approach, as these models are used for open-end text generation, conversational AI, content creation, and code generation. Proprietary LLMs, such as OpenAI’s GPT-4o (OpenAI, 2025), offer state-of-the-art performance and support for multimodal tasks but are often closed systems with limited transparency and lacking the possibility of fine-tuning (Jeon & Lee, 2025). In comparison, open-source models offer the possibility of fine-tuning. Touvron et al. trained the open-source model LLaMA-13B on public data and outperformed GPT-3 (Brown et al., 2020) with their approach (Touvron et al., 2023). They based their work on the zero and few-shot learning methods presented by Brown et al., offering adaptation for specific tasks for LLMs.

Despite their increasing capabilities, LLMs still have limitations. They lack domain-specific knowledge, real-time updated information, and proprietary knowledge (Pan, 2024). Another downside is that these models still could generate factually inaccurate text, also known as “hallucinations” (Xu et al., 2024). Therefore, methods for improving the answers from LLMs are constantly being developed. Methods for solving potential issues and limitations of these models are presented in the following chapter.

2.3 Comparison of RAG and GraphRAG

As previously mentioned, LLMs have specific capabilities and limitations. LLMs have significant potential for natural language understanding and generation. However, they also have limitations, such as hallucinations and a lack of domain-specific knowledge. Retrieval-Augmented Generation (RAG) and Graph Retrieval-Augmented Generation (Graph-RAG) were developed and introduced to overcome these limitations. This section compares RAG with GraphRAG, highlighting their differences, advantages, and relevance to the context of BIM APIs.

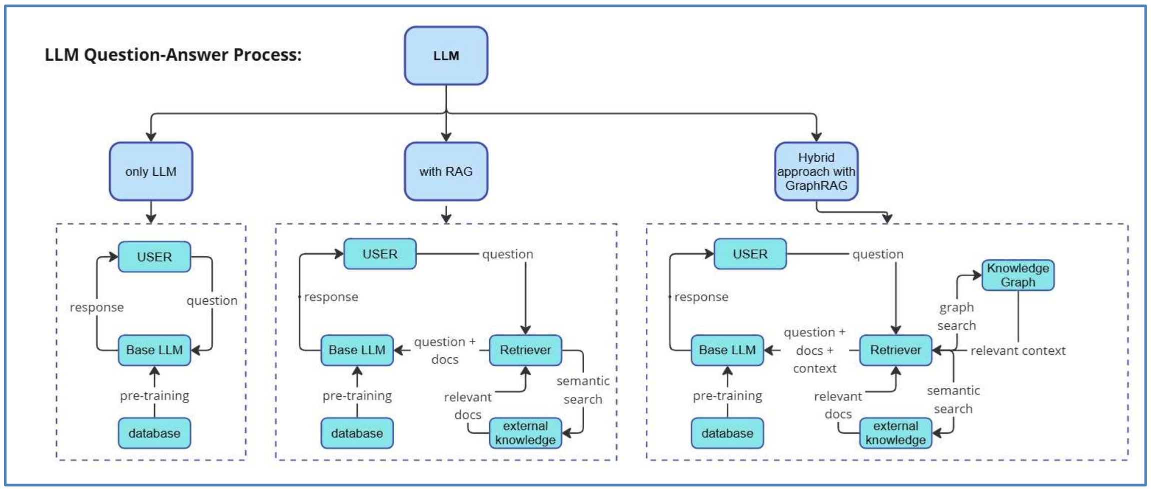


Figure 3 - LLM Question Answering Process: comparing RAG vs. GraphRAG

Figure 3 summarizes the different processes described in the following paragraphs.

Overview RAG

Retrieval-augmented generation (RAG) enhances large language models (LLMs) by dynamically integrating external knowledge bases into the reasoning process. RAG retrieves relevant information from external sources based on vector similarity by encoding the input query and database content into embeddings. This allows for efficient retrieval by identifying the most relevant documents for the question. Combining LLM's inherent reasoning capabilities with factual knowledge retrieved using RAG methods improves the response's accuracy, reliability, and topicality (Peng et al., 2024).

The typical retrieval process is illustrated in Figure 3; the user's question is processed by a retriever, which embeds the question and performs a semantic search to identify the most relevant documents from an external source. These documents are then forwarded to the LLM, together with the question, which generates a response based on its training data and the additional retrieved information.

The quality and effectiveness of RAG's outputs depend heavily on the relevance and quality of the retrieved data, as it directly influences the generated response. While RAG effectively captures semantic similarity, it is limited in representing relationships between pieces of information (Jeon & Lee, 2025).

An upcoming solution for the challenges with RAG involves Graph Retrieval-Augmented Generation (GraphRAG), where information is retrieved from graph elements containing relational knowledge (Peng et al., 2024).

Overview GraphRAG

GraphRAG can be seen as an extension of RAG methodologies, retrieving information from a graph database instead of solely relying on unstructured text input. Those knowledge graphs represent information with entities and relationships, enabling the model to reason about connections between the data (Peng et al., 2024). This explicit relational modeling allows for multi-hop reasoning and explainability. Multi-hop reasoning uses the edges between nodes to perform contextual reasoning utilizing multiple entities. Explainability is enabled by the graph structures as a logical retrieval path can be traced and explained.

A structured overview literature review regarding methods for the step of the RAG process is presented by Peng et al. The main steps presented are Graph-Based Indexing, Graph-Guided Retrieval, and Graph-Enhanced Generation.

GraphRAG relies strongly on the previously generated graph; therefore, a high responsibility lies in the information extraction process and node generation (Jeon & Lee, 2025). Numerous studies focused on improving the automated knowledge graph construction leveraging LLMs to enable a broader adoption of GraphRAG. A selection of those approaches is presented in Chapter 3.3.

Nonetheless, it increases the system's complexity as it has construction, and maintaining a knowledge graph adds extra steps and effort.

Comparison of RAG with GraphRAG regarding their relevance for BIM APIs

In the context of BIM APIs, GraphRAG has some advantages over RAG. APIs often involve connected functions, parameters, and datatypes, requiring a detailed understanding of their relationships. RAG would retrieve information about individual text snippets, such as separate functions, in a thorough manner. Whereas, GraphRAG could retrieve the description and the dependent functions or parameters, providing a more holistic view.

2.4 Characteristics of BIM Authoring Tool's APIs

Building Information Modeling (BIM) APIs play a crucial role in the construction industry by enabling developers to extend, customize, and automate tools and workflows in their 3D modeling software. Therefore, most software products have a programming interface for developers to use to develop their add-ons. Furthermore, software vendors offer documentation for their BIM authoring software's API as guidance for the developers; for example, Nemetschek Allplan offers documentation for PythonParts[11] extension, Nemetschek Vectorworks has documentation for VectorScript and Python APIs[12], and Autodesk Revit offers Revit API Docs[13]. Those APIs define how to interact with programs at a coding level and document the common structures, standards, and terminologies. However, their complexity and the lack of standardization between the different tools and documentation pose challenges for developers.

2.5 Summarization

In the context of BIM APIs, LLMs offer the potential for automating and simplifying complex documentation. However, their limitations, particularly hallucinations and lack of relational reasoning, make them less effective when applied in a domain-specific scenario. Augmenting LLMs with knowledge graphs has the potential to fill these gaps by integrating structured knowledge to improve the accuracy and contextual relevance of responses for developers. The following chapter examines current approaches leveraging LLMs and KGs in general and in the BIM field.

[11] Allplan PythonParts: https://pythonparts.allplan.com/2025/manual/getting_started/
[12] Vectorworks: https://developer.vectorworks.net/index.php?title=VS:Function_Reference
[13] Revit API: https://www.revitapidocs.com/

3 The current state-of-the-art in literature

Over the past decades, Building Information Modeling (BIM) technology has significantly transformed the construction industry. What began with the creation of basic CAD drawing has evolved into the development of complex multidisciplinary models that integrate advanced programing interfaces (APIs). These programming interfaces enable an upgrade in the time and efficiency of model construction. However, as the reliance on APIs grows, so does the complexity of their documentation, often leading to inefficiencies in their adoption and use. Addressing these challenges requires innovative approaches that integrate advanced technologies to manage and interpret this information.

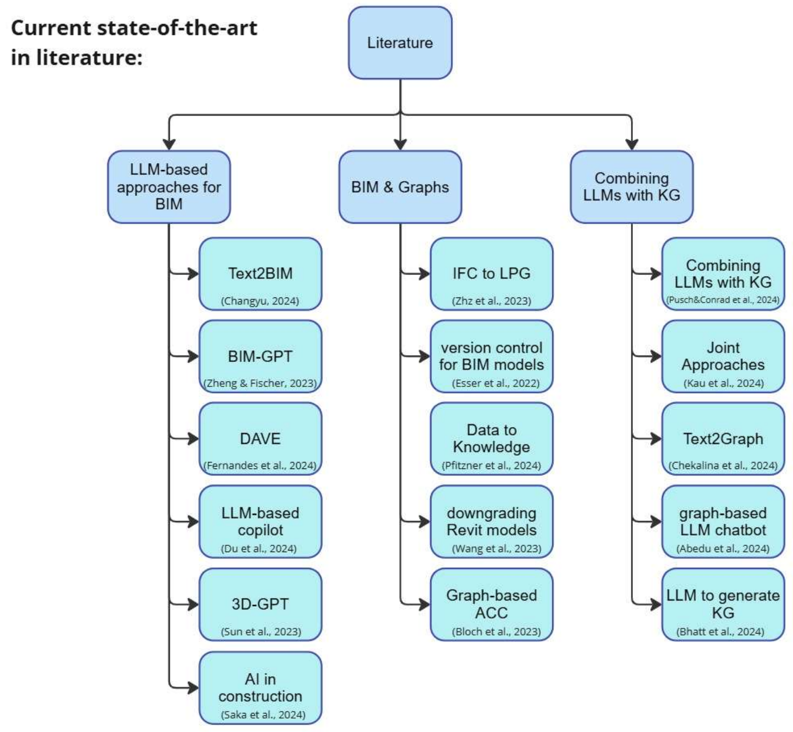

Current state-of-the-art in literature:
Literature
LLM-based approaches for BIM
BIM & Graphs
Combining LLMs with KG
Text2BIM (Changyu, 2024)
BIM-GPT (Zheng & Fischer, 2023)
DAVE (Fernandes et al., 2024)
LLM-based copilot (Du et al., 2024)
3D-GPT (Sun et al., 2023)
AI in construction (Saka et al., 2024)
IFC to LPG (Zhz et al., 2023)
version control for BIM models (Esser et al., 2022)
Data to Knowledge (Pfitzner et al., 2024)
downgrading Revit models (Wang et al., 2023)
Graph-based ACC (Bloch et al., 2023)
Combining LLMs with KG (Pusch&Conrad et al., 2024)
Joint Approaches (Kau et al., 2024)
Text2Graph (Chekalina et al., 2024)
graph-based LLM chatbot (Abedu et al., 2024)
LLM to generate KG (Bhatt et al., 2024)

Figure 4 - Overview literature review

This study builds on three areas in the literature to explore solutions for improving the usability of BIM API documentation, as visualized in Figure 4. In the first part, Chapter 3.1., recent advancements in Large Language Models (LLMs) will be explored, demonstrating their potential to process natural language queries, generate contextual responses, and automate tasks. In the context of BIM, LLMs have been applied to tasks in different stages of design, data retrieval, and compliance checking (Saka et al., 2024). Furthermore, the possibilities of automated model construction have been examined. However, these approaches often struggle with domain-specific challenges, such as hallucinations and a lack of relational reasoning.

Chapter 3.2. introduces current approaches using graphs to make BIM data accessible. By leveraging Labeled Property Graphs (LPGs) and Resource Description Frameworks (RDFs), researchers have developed methods to address issues such as interoperability, traceability, and version control in BIM data.

Chapter 3.3. highlights the opportunities of combining knowledge graphs with LLMs. Integrating knowledge graphs with LLMs represents a promising approach to overcoming the limitations of each technology when used in isolation. Literature in this area explores how KGs can enhance the accuracy and contextual relevance of LLM outputs.

Those reviews lead to a methodology for combining LLM with KG to retrieve information and assist with specific documents in the field of BIM.

3.1 LLM-based approaches for BIM

Since the emergence of different LLMs, such as OpenAI's GPT series, Google's Gemini, and Meta's LLaMA, this technology has been widely spread in many fields, including business, education, and the construction industry. Integrating large language models (LLMs) into the construction sector, particularly in Building Information Modeling (BIM), has opened unforeseen possibilities to optimize design, management, construction, planning, and operational processes. Numerous studies explore the upcoming opportunities emerging by integrating LLM into the BIM processes. Aiming to improve and automate the design process, improve usability, and assist users with BIM authoring software.

Text2BIM

Studies like Text2BIM have demonstrated a multi-agent LLM framework to convert natural language inputs into code to generate building models in BIM authoring software, enabling the automation of early-stage design processes (Du, Esser, et al., 2024). Their framework achieved native BIM Models with internal layouts, external wrappers, and semantic information. They used four collaborating LLM agents to generate a BIM model from textual design information without fine-tuning. The agents are controlled with prompt techniques, and a key innovation is the quality optimization loop, a rule-based model checker agent, enabling iterative conflict resolution. They test the framework with different models and design cases to demonstrate the efficiency and feasibility. Their approach showed the possibilities for utilizing LLM agents to automate the modeling process in BIM authoring software.

BIM-GPT

In the design phase, BIM models become complex and detailed, and untrained users possibly struggle to handle these models. Solving this issue, BIM-GPT provides a prompt-based virtual assistant framework to help practitioners retrieve and manage data from complex BIM models (Zheng & Fischer, 2023). The user interface from BIM authoring software is often complex, and lots of multi-disciplinary data are coming together in the model. Therefore, it can be difficult for non-experts to retrieve data from the model. The core module of their approach is a prompt library and a prompt manager generating answers to BIM-related questions. Their study defined several use cases with individual prompt goals in detail. Dealing with identifying the user's intent and generating the correct query for the correct answer. A complex prompt manager handles and chooses the right prompt. The prompts to be chosen by the Prompt Manager are categorized for different use cases: the intent prompt classifies the intent, then the parameter prompt identifies the parameters and filters, after that, the value prompt extracts the identified values, and lastly, summarizes the results. Their approach showed high accuracy rates for retrieving data efficiently, advancing the usability of authoring software.

DAVE

Another approach aiming to improve the usability of BIM authoring software focusing on interacting with the model is presented by Fernandes et al. (2024), a GPT-powered Digital Assistant for Virtual Engineering (DAVE). Their approach utilizes an LLM to interact with BIM models using the authoring software API and an AI chatbot application. The architecture comprises four components: Data Extraction, Python, JSON Bridge, and C# Revit API updater. The Data Extraction Component prepares structured project data (JSON/CSV) for the GPT Assistant, while the Python component handles the user interaction, natural language processing, and communication with OpenAI API. The JSON Bridge Component ensures real-time communication between the Python and C# Revit API Component, which executes updates in Autodesk Revit based on the user's command. What is interesting about their approach is the way they leveraged the Revit API and the AI component together. The scalability of their system is currently limited by the fact that they are mapping every Revit interaction to a unique Python function. This poses challenges for larger BIM models. A possible solution could be automating the function generation using AI again.

LLM-based copilot

LLM-based copilots, such as those proposed by Du, Nousias, and Borrmann (2024), can support developers and users of 3D-modeling software by answering technical questions about the usage of tools and providing suggestions. These tools are particularly valuable for navigating complex BIM software. Their method is based on an agent framework where an agent selects the appropriate tool according to the tool description from a predefined tool set.

3D-GPT

An approach not specific to BIM but also dealing with automated 3D modeling based on textual user instructions is 3D-GPT (Sun et al., 2023). They showcased that their 3D-GPT can interpret instructions and execute modeling tasks. The method is based on three agents: one for task dispatch, one for conceptualization, and one for modeling. Their work demonstrates how LLMs can streamline workflows in 3D modeling.

Overview AI in construction phases

An overall view of LLM-enhanced BIM application in the construction sector is provided by Saka et al. (2024). They analyzed the current approaches available and offered a structured overview. The approaches are sorted in the paper according to the construction phase they facilitate. Starting in the pre-design phase, LLMs facilitate decision-making by analyzing the project requirements and generating recommendations for optimal design strategies, construction techniques, and project schedules. This is followed by the design phase, where LLMs enhance BIM models by generating improved specifications and automating compliance checking. During the construction phase, many potential application fields exist for LLMs. They enhance project scheduling and logistics by analyzing and understanding textual information. LLMs automate rule verification in regulatory compliance, converting regulations into logical clauses for efficient compliance checking. Analyzing extensive documentation to identify hazards and predict outcomes is an excellent benefit for risk management. Furthermore, the ability to process textual and visual data helps to automate updated information in the construction process, saving time in progress monitoring and reporting. LLMs can support the operation and maintenance phase by analyzing energy consumption and predicting maintenance needs. Besides all these opportunities, the paper identifies several challenges, categorized as industry-related, LLM-related, and challenges arising from combining both. The LLM-related challenges are hallucinations, interoperability, liability, and ethical issues. Hallucination-related issues can be a severe danger in the construction sector, as a miscalculation or planning error can result in a tremendous cost increase. An industry-related issue is the limited ability to understand domain-specific knowledge, as the industry requires a deep technical understanding of different principles. Furthermore, many regulations in the building sector are changing from case to case. A solution to solve these issues could be utilizing RAG methods.

All of the previously mentioned approaches aim to improve the handling of BIM authoring software by incorporating the ability of LLMs to understand and process natural language. However, the large amount of knowledge data in the construction sector and the tendency of LLMs to hallucinate when it comes to topics out of their scope appear to be unsolved challenges. Other studies have already focused on processing large amounts of BIM data and gaining insight. The following chapter evaluates approaches for accessing BIM data through graphs.

3.2 Accessing BIM data through Graphs

Accessing and querying data in Building Information Modeling (BIM) remains challenging due to the complexity and heterogeneity of data sources; the same applies to BIM APIs. Graph-based approaches offer significant potential in addressing these challenges by leveraging the graph's structured and relational nature. This section discusses different methods for using graphs in BIM to improve workflows and access data.

IFC to LPG

Zhu et al. (2023) developed a graph-driven approach to converting Industry Foundation Classes (IFC) data into Labeled Property Graphs (LPGs) to reveal hidden relationships between data elements and make building information accessible and queryable. In the paper, they compared the suitability of RDF or LPG for their approach, favoring LPG as they are more suitable for access and query ability and performance reasons. RDF graphs are more suitable for use cases where linking heterogeneous data is the focus. After analyzing the structure of IFC and LPG, they conceptually mapped the attributes from IFC to the LPG concept. They validated their model-driven conversion by comparing the number of nodes with the number of IFC instances. The approach has proven that the various IFC-SPF models can be transformed into LPG without information loss. Nevertheless, they noticed during their process limitations, such as performance issues rising with larger files or Cypher queries not being tailored to query-building information.

Version control for BIM models

Another approach introduced by Esser et al. is the object-based version control for BIM models to overcome traceability issues for individual objects. The method is based on representing the object network, forming a BIM model as a property graph structure, and tracking changes as graph transformation (Esser et al., 2023). They presented a methodology for merging design decisions into a vendor-neutral data model. Their concept of “branch-and-merge” presents a solution for merging multiple diverging model states produced during the design process into a consistent coordination model. They merged the models' changes by expressing them as graph transformations, enabling updates through patterns context, push-out, and gluing. Their system identifies and resolves conflicts using semantic matching and structural matching. With their approach, they enhanced the overall collaboration in the construction industry and showcased the usability of graphs for BIM models.

Downgrading Revit Models

Furthermore, graphs can be used when it comes to the problem that software formats cannot be opened in earlier versions. One study presents an approach to address this problem by using a generated BIM graph and software APIs, illustrating the process of downgrading a Revit model from version 2023 to 2022 through three main steps: exporting object geometries, generating the BIM graph, and reconstructing models using enriched semantics and software APIs (Wang et al., 2023). They proposed a pipeline to predict semantic types of objects and reorganize them into a KG, which can be used for reconstructing in an earlier version.

Data to Knowledge

Pfitzner et al. (2024) presented an approach to turn data into knowledge using a graph. They presented a pipeline that included data acquisition, processing, and extraction knowledge from the construction side. By developing a labeled property graph to represent construction objects, processes, and their interrelations, they integrated time- and location-specific data into a graph, enabling insights into the construction phases.

Graph-based automated code checking

Buildings and their models must fulfill many requirements, and checking them can be time-consuming and inefficient. Automated code checking (ACC) has been introduced to BIM to solve this problem. However, ACC is limited because the regulations in the construction field are often complex and not straightforward to check. Therefore, Bloch et al. (2023) introduced a graph-based ACC approach to overcome limitations such as regulations that do not address only geometric aspects but also relational ones.

Those approaches showcase the relevance of graphs for the area of BIM. There are different methods for utilizing graphs to enable information exchange, retrieval, and comparison. The presented studies all use the graph's benefit of relational information representation to connect different kinds of BIM data.

3.3 Enhancing LLM with a Knowledge Graph

LLMs and KG graphs are quite different technologies, but looking at their pros and cons, it becomes clear that they leverage each other. LLMs are proficient at processing language and have broad knowledge. However, they are a black box, hallucinations can occur, and they have little domain-specific knowledge. KGs offer structured, precise knowledge but are often incomplete, and knowledge can be overlooked. Furthermore, they cannot process natural language, which makes it occasionally difficult for users to retrieve specific information from the graphs. Looking at these individual aspects of the two technologies, it becomes clear that a combination of the two can bring many benefits. This chapter will evaluate the current approach of combining KG with LLMs to identify the potential for transferring their concept to the BIM field. The topic is split into three parts: Approaches for combining KG with LLMs, Approaches for enhancing LLMs with KGs, and Approaches utilizing LLMs to construct KGs.

Approaches for combining KG with LLMs

An approach of combining KG with LLMs was taken by Pusch & Conrad; their goal is to improve the reliability of question-answering systems. They compared the generated Cypher queries from different LLMs and implemented a query-checking algorithm to validate the results (Pusch & Conrad, 2024). In their approach, they used LangChain

Neo4jGraph for the graph schema generation. Then, an LLM generates a Cypher query, which a query checker will validate. The query checker identifies missing properties at the return statements, incorrect node types, and relationship direction errors. Then, the query will be executed. They devised three question types for testing: 1-hop questions involving only one relationship, 2-hop questions involving two relationships, and 3-hop questions with a chain of four entities and three relationships. They tested their approach for different scenarios and different LLMs. They compared Zero-shot, One-shot, and Few-Shot Prompting for GPT4-Turbo, and LLaMa3-70B. This results in a higher performance for GPT4-Turbo for Zero-Shot and an increased performance from LLaMA3-70B when using Few-Shot Prompting compared to Zero-Shot.

Kau et al. (2024) have done a deep literature review to answer how KG can enhance LLM capabilities, how LLMs can support KGs, and the advantages of combining KG and LLMs in a joint fashion. They categorize the found papers into approaches that use KG or LLMs as “Add-ons” versus “Joint” approaches. Joint approaches combine the strength of LLMs and KGs by fusing textual and knowledge embeddings in a joint encoder. This results in a better semantic understanding of knowledge.

Approaches for enhancing LLMs with KGs

As hallucinations of LLMs can be a significant problem when using LLM-based approaches, connecting LLMs with a structured knowledge source might favor the correctness of the results and improve the reasoning of the answer. Chekalina et al. (2024) developed an approach combining LLMs with KGs, showcasing that it can reduce hallucinations and improve accuracy. Their method transforms input text into a set of KG embeddings, and then they use an adapter to bring these embeddings into the language model space. In their approach, they trained a Text2Graph mapper and KG embedding Adapter. This results in adding a KG modality to the Large Language Model. The evaluation of the results comparing three LLM models with and without the KG modality has shown an improvement in accuracy. The accuracy depends on the LLM and the test data set and, therefore, varies.

A similar approach to what this study is trying to implement was done by Abedu et al. (2024); their study improved the accuracy of an LLM-based chatbot by augmenting it with a knowledge graph. They created a Knowledge Graph Constructor to generate the graph based on the repository data. Furthermore, they enable interaction with a combination of a Query Generator, a Query Executer, and a Response Generator.

They found that their KG-enhanced LLM answered 65% of repository-related questions correctly. When integrating chain-of-thought prompting, the correctly answered questions raised 84%.

Approaches utilizing LLMs to construct KG

Creating those KG from raw data and combining them with LLMs appears challenging. Different studies developed approaches utilizing LLMs to enhance and automate the KG construction process.

Yao et al. (2023) present an approach for using an LLM to complete KGs. Their study shows a significant improvement in LLM model answering performances when integrating KG data.

Trajanoska et al. (2023) presented a method for an automated process to extract information from unstructured data by constructing a KG using Natural Language Processing. Their approach focuses on creating KG from raw text by integrating advanced LLMs with semantic technologies.

A novel approach presented by Bhatt et al. (2024) leverages LLM to generate KGs directly from unstructured data without using traditional pipelines. Their pipeline consists of five steps: data selection, preprocessing of the data, KG generation, ground truth creation, and evaluation. The input text undergoes tokenization, cleaning, and formatting to ensure its compatibility with the different models. They compared three models for their approach: GPT-4, LLaMA 2, and BERT. Each of the models independently extracts entities and relationships to generate the graph. A manually constructed ground of truth KG serves as the benchmark to validate the generated KGs. In their evaluation, GPT-4 demonstrated the highest performance, with high precision, recall, and semantic alignment. Balanced performances suitable for limited resources scenarios showed LLaMA 2. Meanwhile, BERT struggled with contextual understanding and generating nuanced relationships.

3.4 Summary of the identified research gaps

The literature research revealed several application scenarios for intersecting LLMs, Knowledge Graphs (KGs), and Building Information Modeling APIs. In the field of LLM-based approaches for BIM, studies demonstrate the potential of LLMs to optimize and automate the BIM process. Showcasing how LLMs can automate the BIM modeling and assist practitioners in managing complex models. However, they face limitations, including domain-specific hallucinations and inadequate relation reasoning, resulting in less reliability and usability. Graph-based techniques promise robust storage for efficiently representing extracted information and retrieving relational knowledge, resulting in the consideration of combining KG with LLMs. The combination offers structured, precise knowledge to reduce LLM hallucinations and enable natural language querying, facilitating access to graph-based data. Even though current graph-based approaches face scalability challenges for constructing and querying graphs from complex, heterogeneous data sources. Approaches showed improvements in automated KG graph construction utilizing LLMs to enhance the entity and relationship extraction from unstructured data.

Considering the specifics of the APIs provided by standard BIM authoring tools, it seems promising to use both LLMs and KGs to foster later use cases such as agent-based interactions. This study proposes a dual approach, creating a semantically rich KG tailored to BIM API documentation and using this “BIM-API-Graph” to extend LLM reasoning. This methodology aims to overcome the limitations of both technologies and enable efficient, accurate, and contextualized search and interaction for developers.

4 Methodology

This chapter outlines the approach adopted to address the research objectives and answer the key questions posed in this study. The methodology is structured to tackle the three main challenges: interpreting complex BIM API documentation, generating graph construction methods, and retrieving relevant information from the graph using an LLM. It is separated into two main chapters: Knowledge Graph Construction and the graph-based RAG-Agent retrieval. The complete approach is shown in Figure 5, providing the main steps of both parts and visualizing their connection.

Chapter 4.1 focuses on the first part, “Graph Construction”, transforming unstructured BIM API documentation into structured Knowledge Graphs. First, the data is prepared by examining various sources, including JSON files, textual documentation, and web-based examples. This is followed by three partly automated approaches for generating the knowledge graph. Separated into one deterministic method for reliable node creation, one semi-automated approach based on LLM embeddings, ensuring semantic richness in the constructed graph nodes, and one method to integrate more complex coding examples from utilizing examples from webpage documentation.

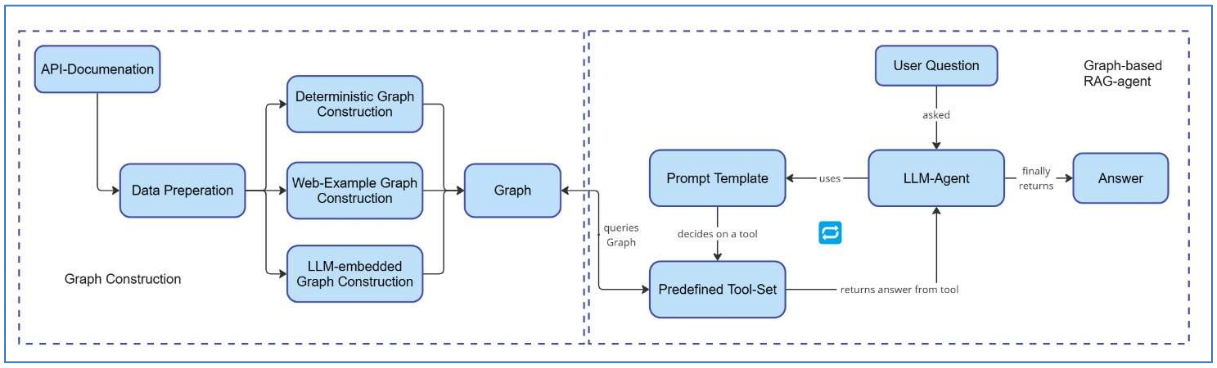


Figure 5 – Methodology Overview

The second part of this approach addresses graph-based RAG-Agent retrieval using the previously generated graph as a data source. Chapter 4.2 presents the development of the graph-based RAG-Agent, which involves designing an agent and a toolset that interacts with the constructed knowledge graph. The LLM agent is based on a custom prompt template, which guides the agent in choosing the appropriate tool from the predefined tool set. Then, the tool is used, and the result is returned to the agent, deciding whether the answer is sufficient or whether another tool is necessary to answer the user's question to its full extent. This decision is based on the predefined “Answering Guidelines” in the agent's prompt. The process is repeated until the results are sufficient, and then the agent forms a final answer, which will be returned to the user.

4.1 Knowledge Graph Construction

The construction of the Knowledge Graph is a crucial component of this research, which aims to transform unstructured BIM-API documentation into a structured, queryable format. The following section will explain two different generation and retrieval processes to identify the best-performing graph. The two approaches, A and B, differ in their LLM usage. Approach A uses a method based on an open-source embedding model and a graph transformer with Cypher queries to generate the embedded chunks and vector indexes. On the other hand, approach B uses a specific tool that combines the KG construction and embedding, and retrieving answers from the graph utilizes a specific GraphRAG tool. The overall process of both approaches enables the integration of domain-specific knowledge into a graph-based representation that can enhance the LLM-based retrieval capabilities. The ultimate objective of this process is to generate a BIM-API-Graph.

The methodology for knowledge graph construction is divided into four key stages: understanding the input data and generating graph nodes through three distinct methods. Each method leverages different data sources to ensure a comprehensive and semantically rich representation. The process is shown in Figure 6.

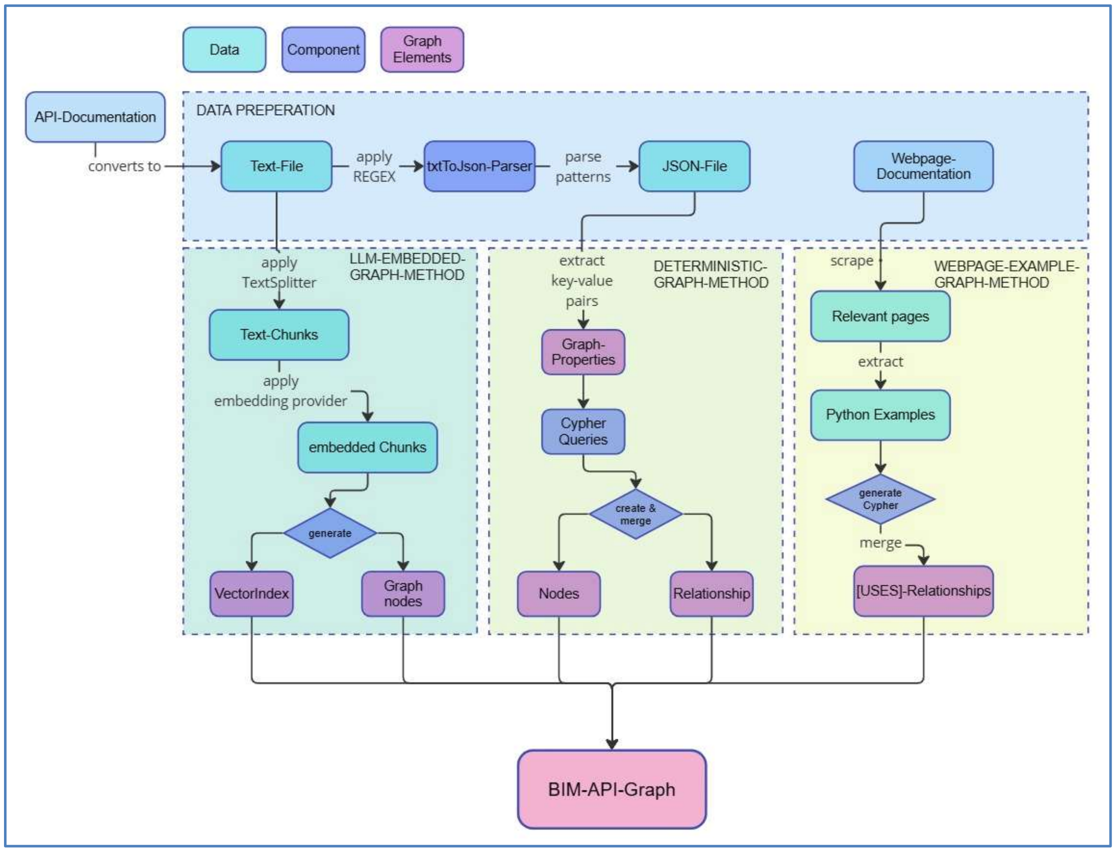


Figure 6 - Approach of the first part of the study generation of the BIM-API-Graph

The process starts with data preparation. First, the documentation is converted into a text file. Then, applying regular expression, the *txtToJSON*-Parser parses the text file into JSON.

From this on, the first method uses the *TextSplitter* to split the text file of the API into chunks, which are then embedded using an embedding model. The embedded chunks are then used to generate a vector index and graph nodes. This process ensures the capture of contextual information in a way that the LLM understands and enables the LLM to discover its node types and relations in the data. In this section, Approach A and B differ; Approach B uses a different tool for embedding and graph generation here. This is utilized to test a different generation-retrieval process to compare the approaches' graph generation abilities.

The deterministic graph method generates nodes by extracting the key-value pairs from the JSON. Those are then supplemented to the Cypher queries as graph properties. The nodes and relationships are then added to the BIM-API-Graph with the queries. This process enables a reliable and consistent extraction of key elements such as functions, parameters, and data types.

The third method enriches the graph by creating relationships between the function nodes based on implementation examples extracted from the webpage documentation. Therefore, the web documentation of the BIM-API is scraped to extract the relevant pages containing coding implementation examples. The function names from the examples are then merged into the BIM-API-Graph using a [USES]-relationship. This adds practical context and aims to enhance the coding implementation suggestions.

4.1.1 Data Preparation

Creating a high-quality knowledge graph begins with a thorough understanding and analysis of the input data, which can vary significantly depending on the structure and content of the API documentation. While some 3D-programming software documentation is well-structured, featuring strongly typed parameters and datatypes, others may lack uniformity or detail, adversely affecting the graph's accuracy. Consequently, the quality and reliability of the graph relies highly on the precision of input data.

In this section, different input data components are evaluated, and specific requirements for graph generation are outlined. Depending on the graph generation method chosen, the requirements for the data differ.

The deterministic approach relies the most on the quality and detail of the documentation data. Manual preprocessing of the input data is often necessary to transform unstructured ".txt" or other formats into structured ".json" files, enabling the extraction of key-value pairs. This step involves identifying specific keywords and elements commonly found in code documentation. Since coding languages vary in syntax and conventions, not all elements are necessarily present in every dataset, requiring adaptability during preprocessing. General elements usually present are functions, function names, input parameters, descriptions, methods, and return values. After identifying those key elements and their syntax, the next step is to decide which other information from the functions might be relevant to the graph, for example, documentation-specific data. Finally, it is necessary to determine how to connect the extracted elements and decide which elements will be nodes, node properties, relationships, and relationship properties. Based on those findings, the graph can then be constructed.

The reliance on manual input is minimized for the graph-creating process using LLM embeddings. Instead, the LLM generates embeddings based on the text chunks extracted from the documentation. These embeddings capture semantic relationships between data, enabling the automated construction of graph nodes. This is particularly effective for unstructured or incomplete documentation as it leverages the contextual understanding of LLMs to enhance the graph’s semantic richness.

Implementing the Webpage-Example-Graph is based on webpage extraction depending on the data from the webpage documentation, which can be quite different for each BIM-API. So, manual input is necessary to understand the structure of those pages and find the parts with the examples. The relevant sections with the coding examples from the webpage need to be identified and delivered to the tool as input.

4.1.2 Construction graph nodes from JSON with a deterministic approach

This part describes the deterministic approach that operates consistently and predictably, producing the same outcome every time given the same initial conditions. The limitation of this approach is that the resulting output accuracy relies strongly on the quality of the input data. This implies that the first step in the deterministic graph creation process is preparing the input data.

This approach is inspired by the pipeline presented by Bronzini, where a data preparation process was also implemented. In the method, a PDF parser was used to extract the text, and then regular expression optimizes the textual data to the specific needs (Bronzini, 2024).

For this approach, JSON is the preferred target format as it allows the storage of key-value pairs and objects. This format makes it convenient to construct the graph. The individual functions from the input data are saved as a list of objects, with each object consisting of the same key-value pairs. In addition, JSON allows the storage of nested elements so that the input parameter values are stored correctly. Then, when creating the graph, it iterates over the individual objects of the JSON and adds the values as node properties to the graph. Jeon and Lee (2025) converted their input data into a structured JSON to retrieve pair datasets.

The data processing in a JSON format differs depending on the input data type. The file from the API-Documentation Function is the original data source; the file is copied to a “.txt”-file and then transformed by the “*txtToJSON*”-Converter into a JSON file where each function element is represented as key-value pairs. The key-value pairs are extracted with regular expression patterns, which find the wanted terms in the textual file. However, for this to work, the input data must be written consistently, as different writing in the original data can cause misdetection. For example, a misspelling might break this process as the pattern would no longer detect the term. For that reason, breaking conditions are very relevant so the skipped lines from the code can be identified.

After the JSON file is correctly created, the node construction begins. The script processes each function object of the JSON, extracts the key-value pairs, and creates nodes and relationships in the graph database. This process establishes a graph-based representation of the data, which can be queried and visualized for insights into function dependencies and structure.

The graph nodes preferably created follow the schema in Figure 7.

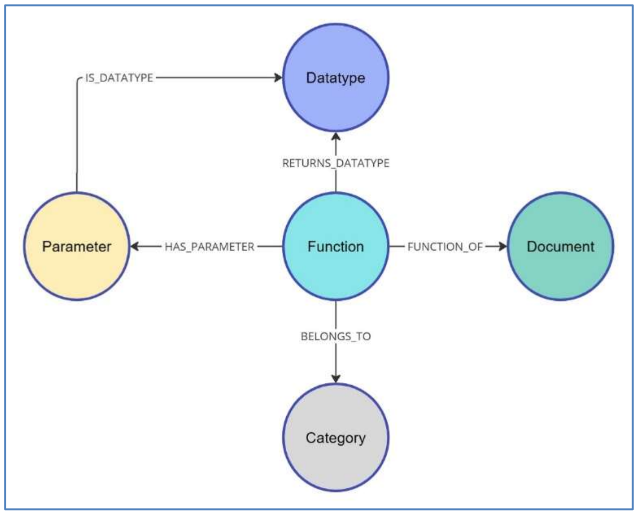


Figure 7 - Deterministic-Graph-Approach: nodes and relationships created with this approach

With this approach, the following relationships and entities will be created.

Table 1 - Nodes and Relationships of the deterministic approach

Relationship and Nodes	Description
(Function)-[HAS_PARAMETER]→(Parameter)	Indicates a Function that has a certain input parameter.
(Function)-[FUNCTION_OF]→(Document)	Indicates a Function that is from a certain document.
(Function)-[RETURNS_DATATYPE]→(Datatype)	Indicates a Function returns a certain datatype.
(Function)-[BELONGS_TO]→(Category)	Indicates a Function belonging to a certain category.

4.1.3 Construction graph nodes from textual files using LLM-embeddings

When the input data lacks proper documentation or the used coding language is not strongly typed, identifying and understanding connections between functions becomes complex. For instance, in a 3D modeling application, constructing a wall requires a sequence of functions to be called in a specific order. The input data, represented as a ".txt" file, will be embedded to address this challenge and enhance the comprehension of such connections. This step provides the LLM-based Agent with semantic context and is searchable with advanced information retrieval.

Approach A

This section outlines the structured approach of processing textual data into semantically rich knowledge graph nodes, integrating vector embeddings for advanced information retrieval. This method comprises five steps: data loading, text chunking, embedding generation, graph insertion, and vector index creation. A similar approach for automated graph construction was made by Jeon & Lee; they used an algorithm to extract nodes, relationships, and vector indices (Jeon & Lee, 2025).

Firstly, the data is loaded with a dedicated loader, in this case, a text loader. This ensures the raw data is accessible for further processing containing the relevant metadata. The loaded document will then be processed into manageable chunks using a predefined separator (e.g., the function keyword "\ndef ") and constraints on chunk size. Overlaps between the chunks are minimized to ensure distinct content for each function. Subsequently, the chunks will be transformed into a vector representation using a language model. The embedding of the data enables capturing semantic information to allow similarity-based comparisons. Additional metadata and the embedding are mapped to graph entities using a graph transformer. The text chunks are represented as nodes linked to the corresponding document nodes. Furthermore, the LLM extracts additional entities from the embeddings; those are added with a relationship to the chunk. The graph nodes schema by this process is shown in Figure 8.

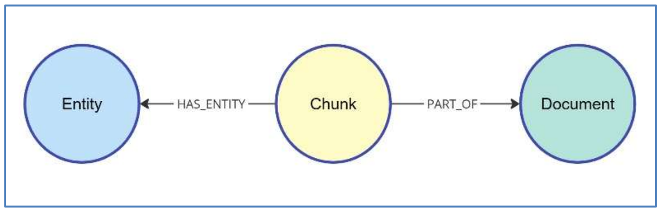


Figure 8 - Embedded Graph node schema

The vector embeddings are stored as a node property with the chunk node, enabling their use in similarity-based retrieval. Finally, a vector index is constructed, leveraging the stored embeddings. The index is configured with 1024 dimensions and uses cosine similarity as its metric. This enables advanced search functionalities, facilitating efficient and accurate retrieval of semantically related nodes.

Approach B

To compare the results, another approach to generate embeddings with an LLM was used to create the nodes with an LLM, which extracts different entities automatically with predefined labels. This approach loads the textual document into a KG-Builder, which then generates embeddings and uses an LLM to extract entities from the embeddings. For this approach, a prompt template is necessary to guide the LLM. Furthermore, certain entities and relations can be defined to guide the LLM regarding the final graph schema. The predefined node labels are chosen based on the detected entities in the input data from Chapter 4.1.1., allowing the LLM to generate a similar BIM-API-Graph to the one in Approach A:

```
basic_node_labels = ["Function", "Category", "Datatype", "Parameter",
"Document" ]
function_node_labels = ["ReturnType", "DataType", "Datatype",
"Parameter", "Document", "Description"]
rel_types = [
    "HAS_PARAMETER", "FUNCTION_OF", "BELONGS_TO", "RETURNS_DATATYPE",
    "IS_DATATYPE", "RETURNS", "USES"]
```

Prompt Template used:

```
Extract the entities (nodes) and specify their type from the following
Input text.
Also extract the relationships between these nodes. The relationship
direction goes from the start node to the end node.

Return the result as JSON using the following format:
{{
    "nodes": [
        {{"id": "0", "label": "the type of entity", "properties":
{{"name": "name of entity"}} }}
    ],
    "relationships": [
        {{"type": "TYPE_OF_RELATIONSHIP", "start_node_id": "0",
"end_node_id": "1", "properties": {{"details": "Description of the
relationship"}} }}
    ]
}}
```

The nodes generated by this approach can vary according to the LLM-identified entities.

4.1.4 Construction graph nodes from examples of the web-documentation

The third node extraction approach aims to enhance the implementation understanding of the functions from the documentation. This part focuses on the real-world implementation examples derived from the web-based program documentation. The process involves web scraping, data extraction, semantic analysis, and graph representation.

The process begins with identifying all relevant hyperlinks from a specific base URL. The web scraping techniques retrieve a list of fully qualified URLs, filtering for those containing references to functions. By combining HTTP requests, HTML parsing, regular expressions, and error handling, the script provides a framework for data-driven analysis. This step is designed to adapt to the variability of software documentation systems and various formats. The process starts by sending requests to the base URL to fetch the raw HTML content of the main page and any linked pages. The script uses parsing and filtering to extract and clean the required data, ensuring only function-related pages are processed. Subsequently, the extracted pages are analyzed to identify and extract code examples. Regular expressions identify specific function calls (e.g. prefixes with vs.) within these examples. The extracted examples are then formatted into Cypher queries, connecting the mentioned functions in the code snippet with a [USES]-relation. Figure 9 shows the connection schema.

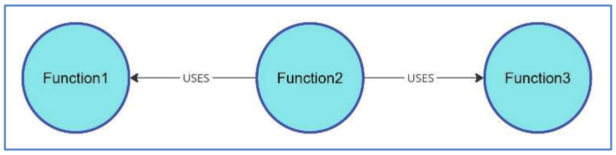


Figure 9 - Showcasing the Graph schema of the [USES] relationship

This script outlines a systematic approach for extracting relationships between functions from a web-based documentation resource and transforming them into a knowledge graph.

4.2 Graph-based RAG Agent

The graph-based RAG agent implements an interactive system to assist developers by leveraging semantic search, graph querying, and stateful conversational agents. The method includes different tools to retrieve relevant programming functions, parameters, and examples.

The application consists of one agent using a set of three tools. This guarantees the optimized quality of the query result. The agents decided which tool to use next based on the user input and the advice and information about the tool. The retrieval process is visualized in Figure 10, starting with a user question asked to the agent, which den decides, based on the Answering Guidelines in the prompt template, which tool to use to query the graph for the needed answer. All tools defined in the predefined toolset query the BIM-API-Graph with different methods and return then the answer to the agent, which generates the final answer or iterates again through the steps to use another tool until the answer is sufficient.

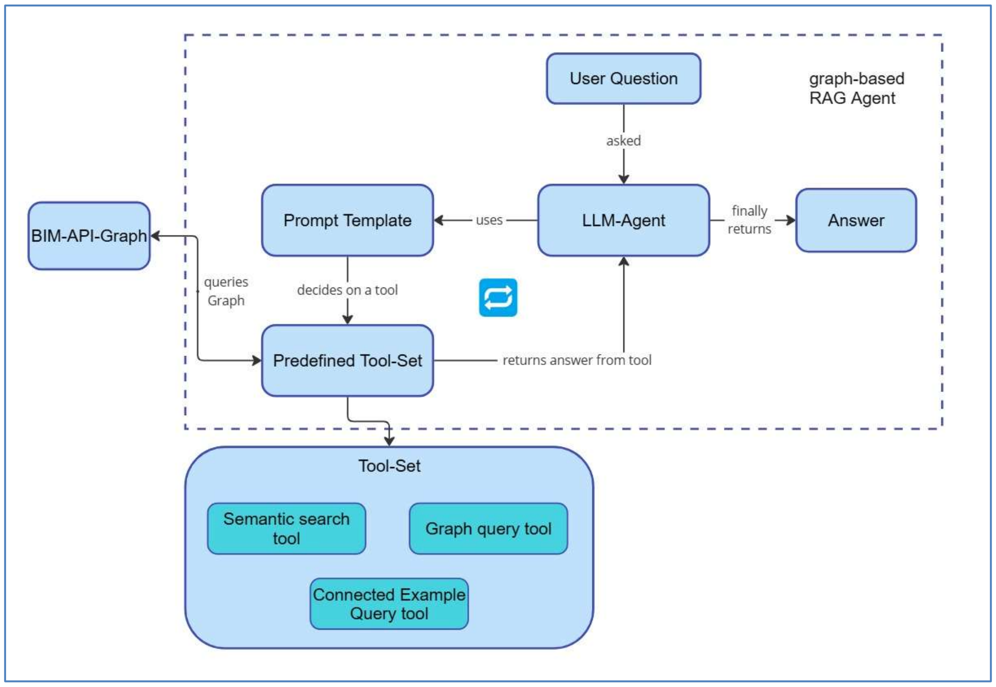


Figure 10 – Graph-based RAG agent methodology

A session manager is implemented to store session data, allowing the agent to maintain context throughout the interactions.

4.2.1 Question types

The agent will face different questions and aim for different results depending on the question asked. Therefore, different tools must be designed to meet user requirements. This section examines different user question types to support the agent later by identifying the right tools for the answer.

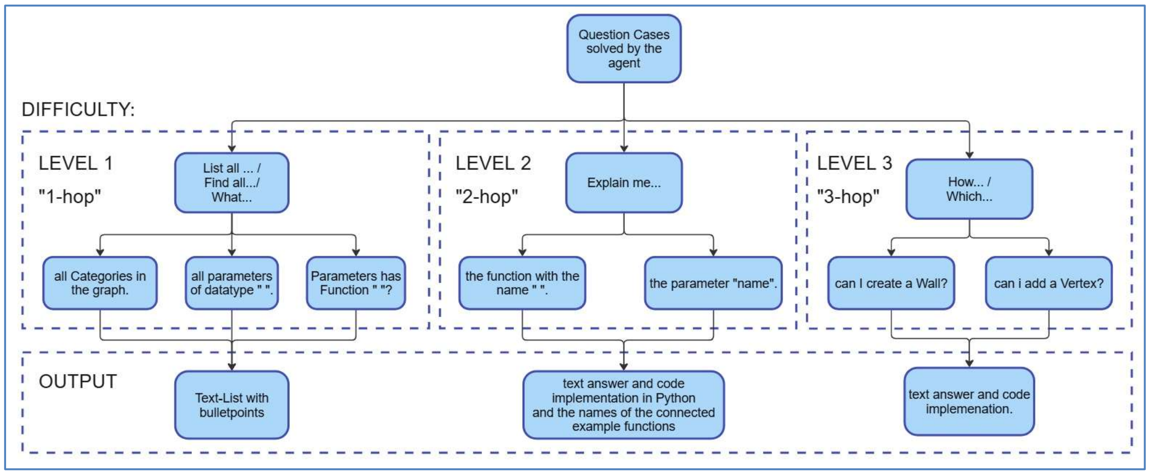

Question Cases solved by the agent
DIFFICULTY:
LEVEL 1
"1-hop"
List all ... / Find all.../ What...
all Categories in the graph.
all parameters of datatype " ".
Parameters has Function " "?
LEVEL 2
"2-hop"
Explain me...
the function with the name " ".
the parameter "name".
LEVEL 3
"3-hop"
How... / Which...
can I create a Wall?
can i add a Vertex?
OUTPUT
Text-List with bulletpoints
text answer and code implementation in Python and the names of the connected example functions
text answer and code implemenation.

Figure 11 - Schema for agent-prompt scenarios

The Schema in Figure 11 shows the types of Questions the LLM-based agent might encounter. The level of categorization is inspired by Pusch and Conrad, who presented an approach for LLM-based KG queries. They categorized their questions to answer by their approach into three sections, defined by how many “hops” the number of edges traversed between nodes in a graph the question answering requires (Pusch & Conrad, 2024). The system supports 1-hop, 2-hop, and 3-hop queries, which involve increasing complexity in nodes, relationships, and paths within the KG.

The question types are categorized into different difficulty levels; level 1 questions require only one tool and are a simple graph query. Level 2 questions are more challenging and typical “explain-questions.” Here, two tools are used: semantic search and the connected example query; those together output a complex text answer with a suggested code implementation and listing of the connected functions. The last is level 3, where general questions about the documentation are asked without background knowledge. Here, the agent needs to use three tools to retrieve possible relevant functions using the semantic search. Then, do a graph query with those functions to retrieve more information, and lastly, find connected functions using the last connect example query.

4.2.2 Agent

An LLM agent guided by a custom prompt will be employed for this approach. The task of the agent is to process and answer user questions. Therefore, it is supposed to use a set of tools interactively and combine the tool output into a logical, straightforward answer. The agent's behavior is guided by a detailed prompt that includes General Instructions, Tool-Usage Guidelines, and Answering Guidelines.

The agent operates interactively by analyzing the user input, deciding which tool to employ, and integrating outputs into a cohesive final response. Furthermore, it handles parsing errors gracefully and provides descriptive feedback when tools fail or unexpected inputs are encountered. The key features and advantages of the agent are that it combines outputs of multiple tools to address complex user questions. Furthermore, it adapts to query types and includes technical explanations and contextual recommendations. To maintain session continuity, the agents leverage Chat Message History, which integrates with the graph database to store and retrieve session-specific chat history. This ensures stateful interaction and improves the user experience.

While the current design is robust, challenges remain, as the user's questions can be very different, and the handling of those ambiguous queries can be advanced. This can be accomplished by expanding the toolset and improving tool coordination.

4.2.3 Prompt Templates

A custom prompt template is created to guide the agent’s behavior, as the prompt can influence the agent's behavior and outcome (Amatriain, 2024). The template defines guidelines for answering questions and the expected answer format.

First, the tool guidelines list the available tools and their descriptions.

```
## Tool-Usage Guidelines
Use the tools listed below as needed to find functions, parameters, data
types, or other requested information. Follow this sequence when using
tools:
1. **Semantic Search**: Use embeddings and a vector index to find
relevant functions, parameters, and data types.
2. **General Graph Query**: Given the input, query the graph database for
special nodes and relationships.
3. **Connected Example Query**: Use this after the other queries to find
functions related to the previously obtained results. This leverages
[USES] relationships.
```

Furthermore, the prompt template contains answering guidelines that guide the agent using the right tools for special questions.

```
## Answering Guidelines
Follow these case-specific guidelines:

- **List all/Find all:**
  Use the **Graph Query** tool to find categories, parameters, etc..
  Return the results as bullet-pointed lists or as clear, full sentences.

- **Explain me:**
  Use both **Semantic Search** and **Connected Example Query** tools to
gather definitions, usage details, and code implementations.
  Present a concise explanation, accompanied by a code snippet following
the Code Implementation Guideline.
  For the code implementation use **Graph Query** to find the function
node with the python property.

- **What/How:**
  Use the **Semantic Search** tools to determine input parameters or
return data types.
  Use the function_names from Semantic Search for tool input.
  Provide clear, full-sentence explanations and code implementation where
helpful.
  If related functions can be found, use the **Connected Example Query**
to list them under "Other Helpful Functions".
```

For the tools, there are also templates guiding the Cypher generation process. For the Cypher tool, an example is used to guide the generation process to the correct relationship.

```
# Find functions connected to a first function (by name case insensitive)
using the [USES] relation
MATCH (f:Function)
MATCH (f)-[r1:USES]->(f2:Function)-[r2:USES]->(f3:Function)
RETURN f.name AS FunctionName, f2 AS FunctionUsed, f3 AS FunctionUsed2
```

No examples are used for the general graph query prompt; there are just general guidance instructions.

4.2.4 Toolset

The application comprises a set of three tools, providing specific functionality for data retrieval. Each tool is encapsulated as a reusable component, enabling modularity in the answering process. For this approach, three structured tools were defined. In the table, the tools are described in terms of basic functionality.

Table 2 - Tool list and description

Tool name	Description
Semantic search	Finds related functions and elements using embeddings and vector indexing.
Graph query	Queries the relationships and nodes within a knowledge graph.
Connected Examples Query	Identifies relationships among results based on graph connections.

5 Case Study: BIM API Graph and Graph-based RAG Agent

In this chapter the Methodology proposed in Chapter 4 is tested on a real example. For this study, the Vectorworks API[14] serves as a case example.

5.1 Knowledge Graph Construction

The knowledge graph is constructed based on the Python file of the Vectorworks documentation. The first step in the construction process is to convert the text file into a JSON file. After that, the deterministic graph nodes are generated, followed by the embedded nodes. The last step is to add extra connections between the functions based on the examples on the webpage documentation.

For these approaches, LangChain components are used in the different components of the graph construction process. LangChain is a framework for developing LLM applications and implementing an interface for embeddings, vector stores, and other related technologies (LangChain, 2025a). They are offering open-source components to build and query graphs. Those components are designed to be modular and interoperable, allowing customization and extension. Furthermore, the framework seamlessly integrates third-party services and tools, which enables data retrieval, storage, and processing. For this approach, the fundamental is the standard interface, which allows the integration of different LLMs and services.

5.1.1 Analysis of Input Data

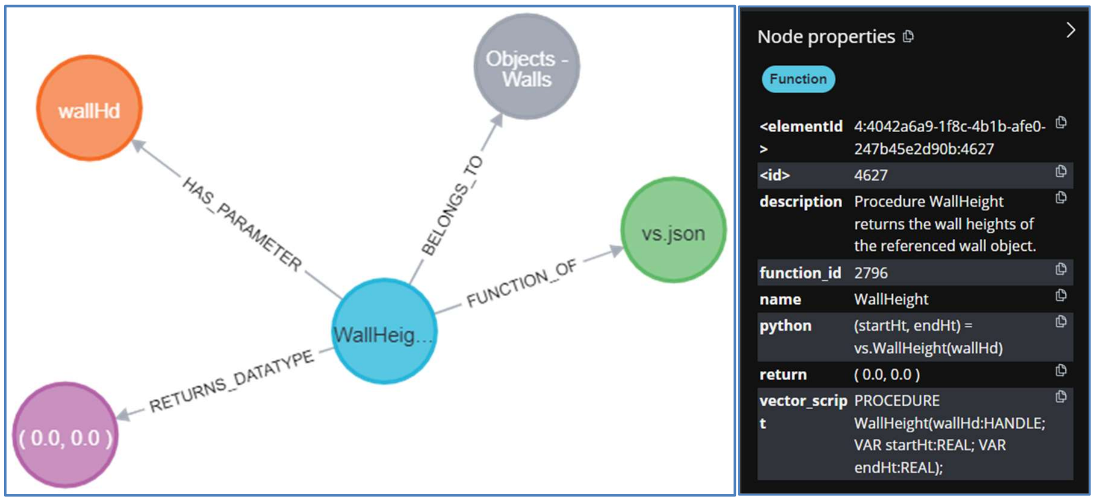
This section will analyze the input data and examine the Vectorworks Python documentation. In the snippet below, an example function shows what an element of code documentation looks like.

```
def WallHeight(
       wallHd  # HANDLE - Handle to wall.
       ):
    '''
       Python: (startHt, endHt) = vs.WallHeight(wallHd)
       VectorScript: PROCEDURE WallHeight(wallHd:HANDLE; VAR
       startHt:REAL; VAR endHt:REAL);
       Category: Objects - Walls
       Procedure WallHeight returns the wall heights of the referenced
       wall object.
    '''
```

[14] Vectorworks API: https://developer.vectorworks.net/index.php?title=VS:Function_Reference

```
    pass
    return ( 0.0,
             0.0 )
```

After a careful examination, the extracted key elements relevant to the deterministic graph are function name, input parameter (name, datatype, and description), Python, VectorScript, Category, Description, and return value and return description. Figure 12 shows the function “WallHeight” from the previous example, translated into the graph. It is essential to extract those elements as they are the key to understanding the function. The next step is identifying which elements must be individual nodes, relationships, or properties. It is important not to map every element as nodes as it would blast the graph without providing extra information; instead, it is necessary to cluster properties with nodes to describe them in more detail.



<elementId>	4:4042a6a9-1f8c-4b1b-afe0-247b45e2d90b:4627
<id>	4627
description	Procedure WallHeight returns the wall heights of the referenced wall object.
function_id	2796
name	WallHeight
python	(startHt, endHt) = vs.WallHeight(wallHd)
return	(0.0, 0.0)
vector_script	PROCEDURE WallHeight(wallHd:HANDLE; VAR startHt:REAL; VAR endHt:REAL);

Figure 12 - Function “WallHeight” in the BIM-API-Graph with the properties of the Function node (right)

So, in this case, elements describing the function will be added to the function node as properties. Therefore, the node will get the element's name, description, python, return, and VectorScript as properties. The node “Parameter” receives name, description, and datatype as properties. This separation enables better queries and adds new relationships to reveal hidden connections. For clustering purposes, the “Category” node is added. Another critical node is “Datatype,” which is the data type of the parameters as well as the return datatype of the functions. This enables relations between functions to be found by finding other functions using the datatype.

```
MATCH(c:Category)-[b:BELONGS_TO]-(f:Function)-[h:HAS_PARAMETER]-
(p:Parameter)-[r]-(d:Datatype)
WHERE type(r)="IS_DATATYPE" AND d.name="HANDLE" AND c.name="Objects - Wal
ls"
RETURN p, d
```

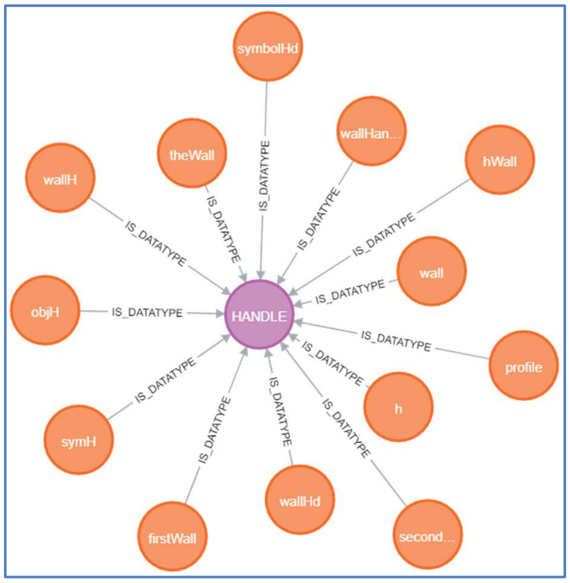


Figure 13 - Query Graph for parameters with special characteristics

With this kind of structure, it is possible to retrieve new data about the documentation from the graph. With the query in Figure 13, it is possible to find all the parameters with the datatype “HANDLE,” which belong to the Category “Objects – Wall.” This information makes it possible to find new relations between functions, helping to understand their usage better.

5.1.2 Construct graph nodes from JSON with a deterministic approach

The deterministic approach utilizes a JSON file to create nodes based on the key elements of the JSON schema. Therefore, the documentation is converted into a JSON file using a custom converter. The “*txtToJSON*”-converter uses regular expression patterns to extract the keys and values from the text file. This part needs to be manually adapted to meet the correct patterns. After the conversion, the JSON can be loaded, and the key-value pairs can be extracted and added to the graph. The key-value pairs are properties that Cypher queries can add to the graph. First, the functions, categories, and datatypes are added, and lastly, the parameter nodes.

5.1.3 Construct graph nodes from textual files using LLM-embeddings

Approach A

For this part, the text file from the documentation is loaded to the graph using the "*TextLoader*"[15] from LangChain. After that, the text is split into chunks using the "*CharacterTextSplitter*[16]" form LangChain. As a separator, the keyword for the function is used in this case, *"/ndef"*. Depending on the documentation size, the following step is quite time-intensive. The chunks are split into batches and embedded for the whole batch. After embedding, the chunks are added to the graph using the "*LLMGraphTransformer*"[17] from LangChain. Finally, a Vector index is created on the property "*textEmbedding*" from the Chunk node.

Approach B

To compare the results, another approach to generate embeddings with an LLM was used. This loads the textual document into a KG-Builder[18] pipeline from Neo4j, which then generates embeddings and uses an LLM to extract entities from the embeddings.

```
kg_builder = SimpleKGPipeline(
    llm=llm,
    driver=driver,
    text_splitter=FixedSizeSplitter(chunk_size=450, chunk_overlap=100),
    embedder=embedder,
    prompt_template=prompt_template,
    entities=node_labels,
    relations=rel_types,
    from_pdf=False
)
```

For this approach, a prompt template is necessary to guide the LLM. In the following section, a part from the prompt template is provided.

```
You are a BIM-API researcher tasked with extracting information from text
documentations
and structuring it in a property graph to inform further research Q&A.

Extract the entities (nodes) and specify their type from the following
Input text.
Also extract the relationships between these nodes. The relationship
direction goes from the start node to the end node.
```

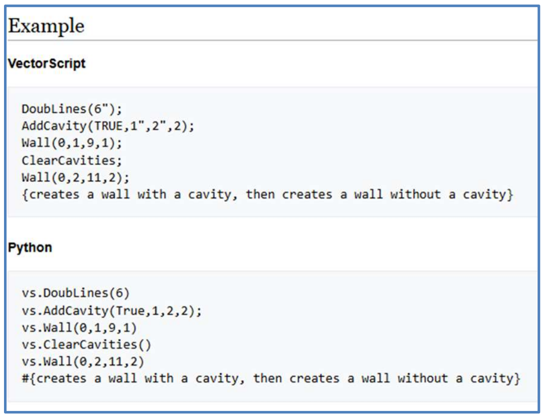
[15] TextLoader: https://python.langchain.com/api_reference/community/document_loaders/langchain_community.document_loaders.text.TextLoader.html
[16] CharacterTextSplitter: https://python.langchain.com/api_reference/text_splitters/character/langchain_text_splitters.character.CharacterTextSplitter.html
[17] LLMGraphTransformer: https://python.langchain.com/api_reference/experimental/graph_transformers/langchain_experimental.graph_transformers.llm.LLMGraphTransformer.html
[18] KG-Pipeline: https://neo4j.com/docs/neo4j-graphrag-python/current/user_guide_kg_builder.html#customizing-the-simplekgpipeline

```
Return the result as JSON using the following format:
{{
    "nodes": [
        {{"id": "0", "label": "the type of entity", "properties":
{{"name": "name of entity"}} }}
    ],
    "relationships": [
        {{"type": "TYPE_OF_RELATIONSHIP", "start_node_id": "0",
"end_node_id": "1", "properties": {{"details": "Description of the
relationship"}} }}
    ]
}}
```

Furthermore, certain entities and relations can be defined to guide the LLM regarding the final graph scheme. This approach also creates a Vector Index.

5.1.4 Construct graph nodes from examples of the web-documentation

For this section, the implementation examples are retrieved from the web documentation. Figure 14 shows an example section from a function from VectorWorks web page documentation[19]. Regular expressions identify specific function calls (e.g. prefixes with vs.) within these examples.


Example

VectorScript

```
DoubLines(6");
AddCavity(TRUE,1",2",2);
Wall(0,1,9,1);
ClearCavities;
Wall(0,2,11,2);
{creates a wall with a cavity, then creates a wall without a cavity}
```

Python

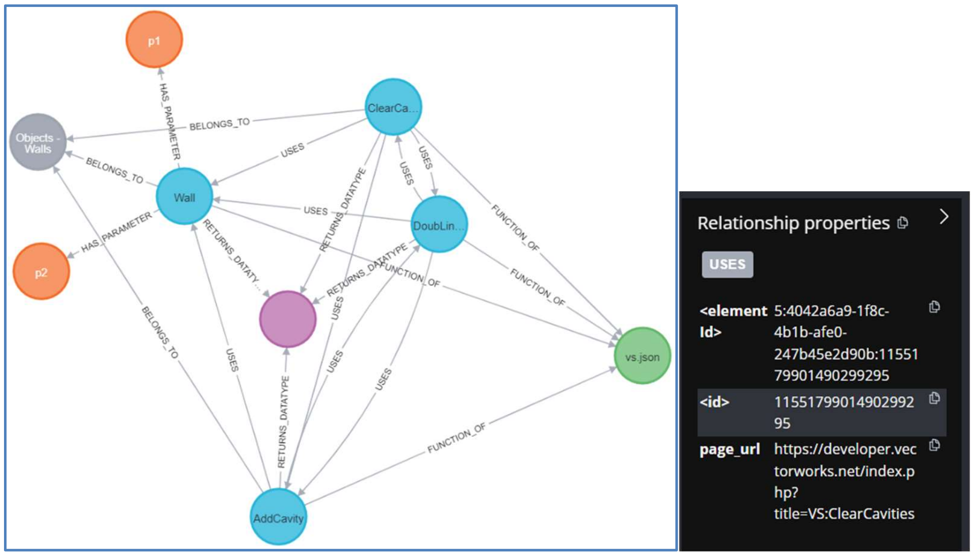
```
vs.DoubLines(6)
vs.AddCavity(True,1,2,2);
vs.Wall(0,1,9,1)
vs.ClearCavities()
vs.Wall(0,2,11,2)
#{creates a wall with a cavity, then creates a wall without a cavity}
```

Figure 14 - Example page showcasing the example section to be detected by the algorithm

Those functions are then saved and added to the graph using Cypher queries. The relation is the property with the page URL of the example added.

Figure 15 displays a section from the resulting graph.

[19] Vectorworks API: https://developer.vectorworks.net/index.php?title=VS:Function_Reference



Relationship properties

USES

<element Id>	5:4042a6a9-1f8c-4b1b-afe0-247b45e2d90b:1155179901490299295
<id>	1155179901490299295
page_url	https://developer.vectorworks.net/index.php?title=VS:ClearCavities

Figure 15 - Example from the resulting graph with the created USES-relations and the properties (right)

The figure shows that the function “AddCavity” uses the “Wall” function in an example code snipped, and in the properties, the page URL defines the location of the example.

5.2 Graph-based RAG Agent Development

5.2.1 Approach A

This section details the development process of the graph-based RAG Agent, focusing on its integration with the knowledge graph constructed in the previous section and its ability to effectively assist developers in querying BIM API documentation. The agent and the prompts are based on an example of Neo4j[20] and are explained in detail in the following chapter.

Choice of Graph system

For the graph database management system, Neo4j[21] was selected, as it offers efficient storage and retrieval of graph data with high performance and scalability. Essentially, this approach was also the local graph storage option, which enables the test of the approach to be local on the machine, enabling more extensive storage options. Furthermore, Neo4j utilizes the declarative query language Cypher, which simplifies the process of formulating complex queries.

[20] Example from Neo4j: https://github.com/neo4j-graphacademy/llm-knowledge-graph-construction/blob/main/llm-knowledge-graph/chatbot/solutions/agent.py
[21] Neo4j: https://neo4j.com/

Framework Selection

The agent system was developed using LangChain and Neo4j, which were chosen for their capabilities in handling semantic search and graph queries. LangChain facilitates the seamless integration of the LLM with the KG, while Neo4j Cypher provides a robust mechanism for querying and managing graph-based relationships.

LangChain is a framework for developing LLM applications and implementing an interface for embeddings, vector stores, and other related technologies (LangChain, 2025a). They are offering open-source components to build stateful agents. Alternatives like LlamaIndex[22] are also possible but are less specialized in conversational agent and tool orchestration than LangChain.

Choice of LLM

The approach for this study is mainly based on an open-source LLM, in this case, the model from Ollama “llama:3.2,” which is a small 3B size model optimized for agentic retrieval, tool use, and summarization tasks (Ollama, 2025a). It is a lightweight model that is easy to run on local machines. As the documentation of API can contain many Lines of code, integrating an open-source model can reduce costs tremendously.

The API documentation of Vectorworks contains 43,536 lines of code, which then has an average of 15 tokens per line, resulting in about 653,040 tokens in total.

For testing of the approach, it was mainly the “lama3.2” model, but to compare the results, the approach with the questions was also tested using the OpenAI model “GPT-4o-mini”. Together with the “llama3.2” model for the embedding provider, the “mxbai-embed-large” model was used, which is the state-of-the-art model from “mixed-bread.ai” (Ollama, 2025b).

Choice of Question

Considering the question schema explained in Chapter 4.2.1 for this study, 36 questions were developed with different difficulty levels. The questions are designed to test whether the LLM can identify the correct entities and relationships. Some questions ask for the same answer in different ways, testing which wording works the best or

[22] LlamaIndex: https://www.llamaindex.ai/

whether it has an influence. In Table 3, the questions for the different levels are listed. They aim to reflect developers' typical questions.

Table 3 – Questions for the different levels

Level 1	Level 2	Level 3
List all node names with the label Category.	Explain to me the function Add3DPt.	How can I create a wall?
List all nodes with the label Category.	Explain to me the function Add3DPoint.	How can i add a roof?
List all nodes belonging to Category Worksheets.	Explain to me the function AddSurface.	Which functions return the datatype REAL?
List all nodes belonging to the Category: Layers.	Explain to me the function CreateWallFeature.	How can I add a vertex?
List all nodes belonging to Textures.	Explain to me the function BeginRoof.	What parameters has the function AddHole?
List all node names with the label Parameter.	Explain to me the function BeginSweep.	How can I create a Layout?
List all nodes belonging to Category Textures.	Explain to me the function AddVertex3D.	How can I add a symbol to a wall?
List all node names with the label Function.	Explain to me the function AddCavity.	How can I add a symbol to a wall?
Find all the parameters using the datatype INTEGER.	Explain to me the CreateLay-out.	How can I add a new story level?
Find me all Parameters with the datatype STRING.	Explain to me the function BuildResourceList.	How can I create a Mesh?
List me the parameters from the function Abs.	Explain to me the function Message.	How can I add a Sweep?
List all functions returning HANDLE.	Explain to me the function AddChoice.	How can I add a point?

Graph Query - Tool adaptation

The “GraphCypherQAChain[23]” from Neo4j was used for the graph query tools, as it provides solid results and is straightforward to integrate and adapt (LangChain, 2025b).

```
cypher_chain = GraphCypherQAChain.from_llm(
    llm=llm,
    graph=graph,
    cypher_prompt=cypher_generation_prompt,
    verbose=True,
    allow_dangerous_requests=True,
    enhanced_schema=True,
    validate_cypher=True,
    top_k=20,
    return_direct=True,
)
```

The parameters are set to receive the best query results; the parameters enhanced_schema allow the chain to use the graph schema to build the Cypher query; with validate_cypher, the query will be validated whether it is runnable. With top_k, the number of returned elements is limited, in this case 20. With the return_direct set to true, the result will be directly returned without the LLM formulating an answer. This is used as the agents need the pure result to develop an overall answer.

To improve the result translation back to the agent, the run_cypher function was modified, and an extract key value was added after the run of the cypher_chain. As the Ollama-based Agent had difficulties interpreting the result directly from the cypher_chain, the extract_key_values function returns extracted the function names from the graph query tool to the agent in the cased used with the Ollama model.

```
def run_cypher(q):
    # Perform the query and return the result
    print("run_cypher")
    print(f"Received query: {q}")
    # Run the cypher_chain
    result = cypher_chain.invoke(q)
    def extract_key_values(data, key_patterns):
        extracted_values = []
        if isinstance(data, dict):
            for key, value in data.items():
                # Match key against key_patterns
                if key in key_patterns:
                    extracted_values.append(value)
                elif isinstance(value, (dict, list)):
```

[23] GraphCypherQAChain: https://python.langchain.com/api_reference/community/chains/langchain_community.chains.graph_qa.cypher.GraphCypherQAChain.html

```
                    extracted_values.extend(extract_key_values(value,
key_patterns))
        elif isinstance(data, list):
            for item in data:
                extracted_values.extend(extract_key_values(item,
key_patterns))
        return extracted_values

    # Define key patterns to search for
    key_patterns = ['p.name', 'c.name', 'n.name', 'name', 'f.name']

    # Extract values matching any of the key patterns
    extracted_values = extract_key_values(result.get('result', []),
key_patterns)

    return extracted_values
```

The key pattern extractions enable the function to return only the wanted function, parameter, or datatype names. The key patterns can be adapted manually.

Connected Example Query - Tool adaptation

The Connected Example Query Tool is similar to the Graph Query Tool; it also uses the GraphCypherQAChain, but the cypher_prompt is slightly different, and the key extraction is different. The prompt integrates an example of the query to be built by the chain, which guides the tool in which relationships and nodes need to be extracted.

```
 Examples:
# Find functions connected to a first function (by name case insensitive)
using the [USES] relation
MATCH (f:Function)
MATCH (f)-[r1:USES]->(f2:Function)-[r2:USES]->(f3:Function)
RETURN f.name AS FunctionName, f2 AS FunctionUsed, f3 AS FunctionUsed2
```

The run_uses function is modified to extract the function names of the connected functions, which then will be returned to the agent from the tool.

```
def run_uses(q):
    print("run_uses", q)
    result = cypher_chain.invoke(q)

    functions_used = []
    for item in result.get('result', []):
        function_used = item.get('FunctionUsed')
        if function_used and 'name' in function_used:
            functions_used.append(function_used['name'])

    return functions_used
```

Semantic Search - Tool adaptation

This tool initializes the Neo4jVector[24] instance, which facilitates retrieval by connecting an existing vector index in a Ne4j graph. A custom Cypher query defines the retrieval and determines the structure of the search results. Then, the similarity_search method is performed based on the user input (user_input) and the number of closest matches to return (k). The method transforms the user_input into a vector using the embedding provider, compares it against the stored vectors in the Neo4j index, and retrieves the top k matches.

```
retrieval_query = f"""
RETURN node {{.text}} AS text, score, {{documentId: "id", vectorScore:
score}} AS metadata
"""
retrieval_example = Neo4jVector.from_existing_index(
    embedding_provider,
    graph=graph,
    index_name="vectorIndex",
    embedding_node_property="textEmbedding",
    text_node_property=["text"],
    retrieval_query=retrieval_query
)
query_result = retrieval_example.similarity_search(user_input, k=3)
return query_result
```

This tool uses Neo4j's ability to store and query textual and vector-based data, making it highly suitable for hybrid search. Leveraging a pre-existing vector index ensures fast retrieval of semantically similar data. With the retrieval_query, the format and metadata of the search result can be controlled flexibly.

Challenges and Limitations

Adapting the agent to handle diverse user queries required extensive customization of the toolset. Ensuring smooth interaction between the LLM agent and the graph query tools posed initial difficulties, particularly returning the query result to the agent in a format the agent understands.

5.2.2 Approach B

In approach B, a different adaptation for the semantic search tool and the graph query tool was examined. A Vector Retriever from Neo4j was chosen for the semantic search tool, which enables a GraphRAG retrieval based on the vector index.

[24] Neo4jVector: https://api.python.langchain.com/en/latest/vectorstores/langchain_community.vectorstores.neo4j_vector.Neo4jVector.html

Graph Query Tool

For the Graph Query tool, a VectorCypherRetriever[25] was adapted using a retrieval query. The VectorCypherRetriever is initialized with the name of the vector index, a Cypher query for additional retrieval, and an optional embedder for text-to-vector conversion. The search process starts with a vector similarity search to find nodes similar to the query; then, for each node retrieved, the retrieval_query will be run to fetch related information, enriching the context for the query. The combination of vector similarity search with graph traversal enables gathering more comprehensive information. The retrieval_query first matches the chunk, which is the start point of the graph traversal, and then the chunk-connected entities are found with the "[:FROM_CHUNK]" relationship. The ")-[relList:!FROM_CHUNK]-{1,2}(nb)" finds relationships between the entity and its neighbors (nb), specified with {1,2} including relationships up to two hops away from the entity. The query returns a list of the connected entities and relations.

```
graph_retriever = VectorCypherRetriever(
    driver=driver,
    index_name="text_embeddings",
    embedder=embedder,
    retrieval_query="""
    MATCH (chunk)<-[:FROM_CHUNK]-(entity)-[relList:!FROM_CHUNK]-{1,2}(nb)
    UNWIND relList AS rel
    WITH collect(DISTINCT chunk) AS chunks, collect(DISTINCT rel) AS
rels, collect(DISTINCT entity.name) AS visited_node_names
    RETURN apoc.text.join([c IN chunks | c.text], '\n') +
        apoc.text.join([r IN rels |
        startNode(r).name+' - '+type(r)+' '+r.details+' ->
'+endNode(r).name], '\n') AS info,
        visited_node_names
    """)
```

Semantic Search Tool

The semantic search tool uses the VectorRetriever[26], which performs a vector similarity search based on the vector index name and an embedder, converting the query text into vector embeddings. The return properties define which node properties will be retrieved.

```
vector_retriever = VectorRetriever(
    driver=driver,
    index_name="text_embeddings",
    embedder=embedder,
    return_properties=["text"])
```

[25] VectorCypherRetriever: https://neo4j.com/docs/neo4j-graphrag-python/current/user_guide_rag.html#vector-cypher-retriever
[26] VectorRetriever: https://neo4j.com/docs/neo4j-graphrag-python/current/user_guide_rag.html#vector-retriever

6 Results and Analysis

This chapter evaluates the effectiveness of Approaches A and B for constructing a BIM-API-graph and assesses the graph-based RAG agent's performance. The analysis focuses on graph quality, agent accuracy, and code suggestion reliability.

First, the structure of the generated BIM-API-Graph is analyzed, comparing the outputs of Approach A and B. Second, the responses generated by the graph-based RAG Agent are evaluated, and problem cases will be identified. This includes comparing answers from two LLMs: Ollama's open-source "llama:3.2-latest" and OpenAI's "gpt-4o-mini." Additionally, the accuracy of the agent-generated code implementations is examined. The next step is the analysis of the results produced by Approach B; interesting here is whether the different GraphRAG-module can produce more solid results using the modified graph structure. Finally, Approaches A and B will be compared to the overall performance and usability. Figure 16 visualizes the description of the result analysis.

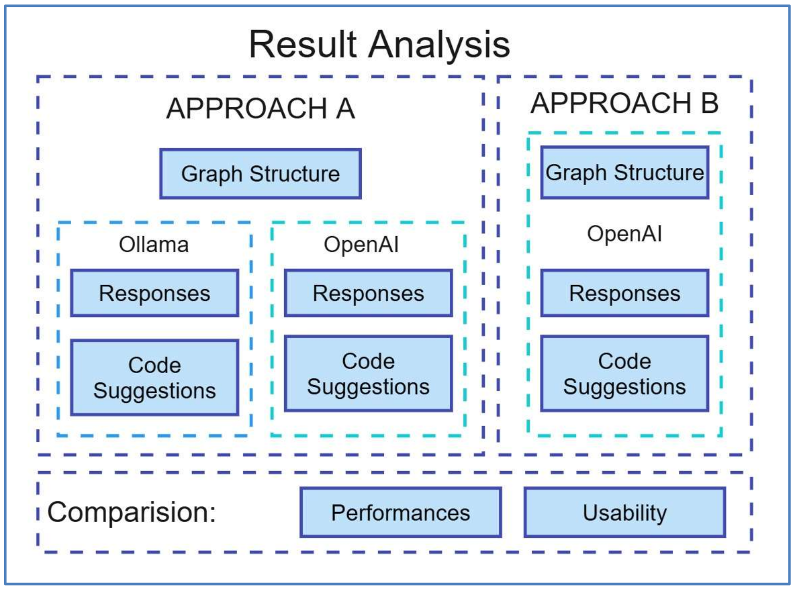


Figure 16 – Overview Result and Analysis

6.1 Approach A

This section describes the results obtained from Approach A, starting with an evaluation of the generated BIM-API-Graph, followed by an assessment of the retrieval performance of the graph-based RAG agent. The approach uses two different LLMs, the open-source model “llama:3.2-latest” and the OpenAI model “gpt-4o-mini”, to evaluate their effectiveness in handling different question types. These questions, designed to test the agent’s capabilities across multiple levels of complexity, were validated by cross-referencing the results with the API documentation. Additionally, agent-generated code implementations were tested for executability and accuracy. The findings are summarized and discussed to highlight the strengths and limitations of the approach in meeting the research objectives.

6.1.1 Graph Structure

The evaluation of the graph structure aims to assess the effectiveness of Approach A in accurately representing BIM API documentation. This section analyses the quantity and quality of nodes and relationships generated, highlighting key characteristics and potential limitations.

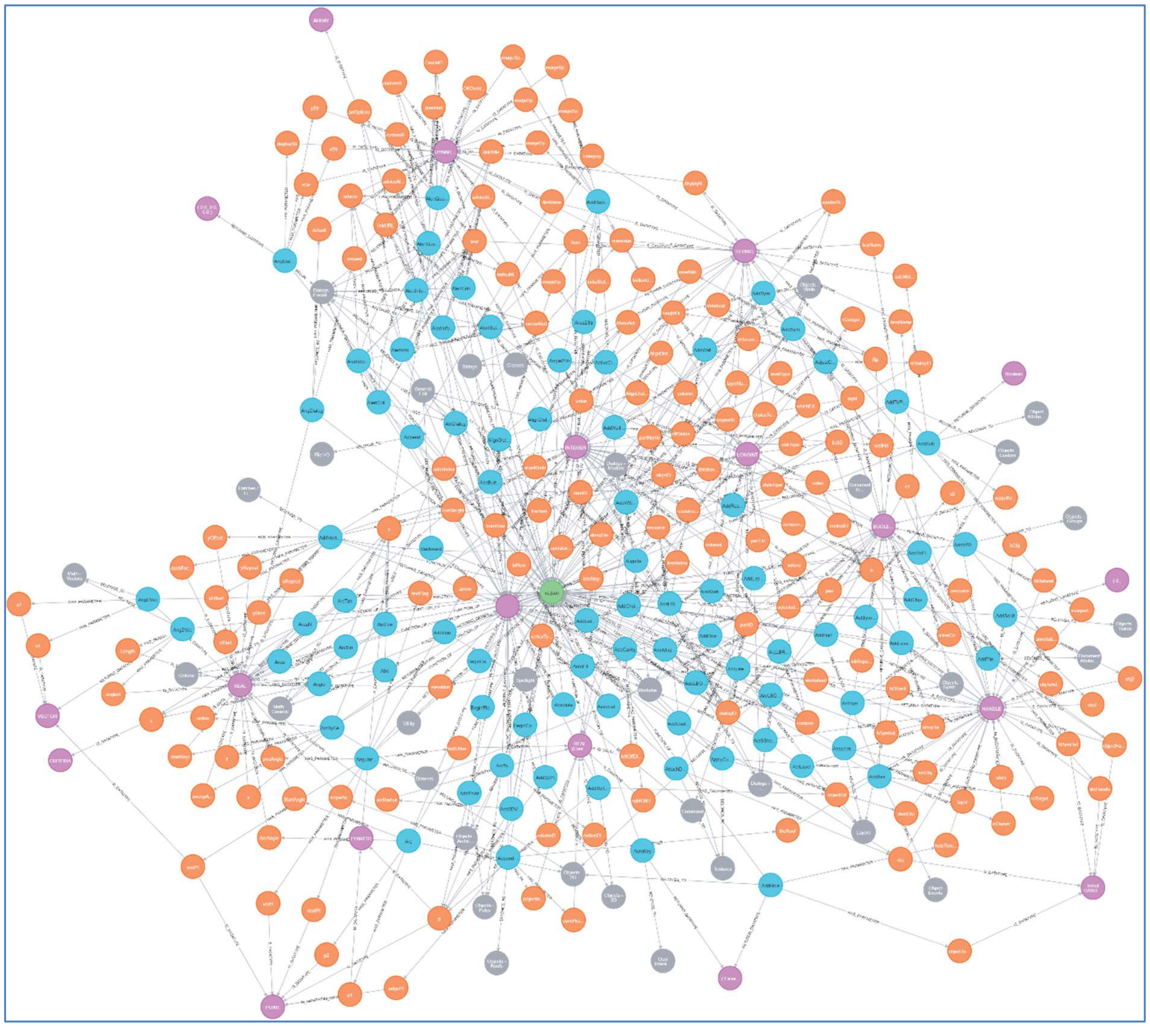


Figure 17 – Section from the Graph generated from Approach A

Figure 17 illustrates a section of the BIM-API-Graph generated from Approach A, comprising a total of 7,650 nodes and 20,860 relationships. The different types of nodes and relationships used in the graph are summarized in

In addition, the relationship “HAS_PARAMETER” links functions to their respective parameters, providing a detailed insight into the dependencies between parameters and functions. With 6,194 recorded connections, this relationship accounts for a significant part of the diagram. The graph also contains 1,622 parameter nodes and 2,866 function nodes from the deterministic approach. The JSON file from the documentation counts 2,866 functions.

Table 4 and provide a clear comparison between the deterministic approach, the LLM-embedded approach, and the webpage example approach.

The “USES-Relationship” establishes connections between functions, enabling the discovery of relations between them, as described in chapters 4.1.4 and 5.1.4. This feature proved particularly useful in uncovering functional relationships, with 1,366 connections generated under this relationship type.

In addition, the relationship “HAS_PARAMETER” links functions to their respective parameters, providing a detailed insight into the dependencies between parameters and functions. With 6,194 recorded connections, this relationship accounts for a significant part of the diagram. The graph also contains 1,622 parameter nodes and 2,866 function nodes from the deterministic approach. The JSON file from the documentation counts 2,866 functions.

Table 4 – Generated Nodes and Relationships from Approach A

	Deterministic Approach	**LLM-embedded Approach**	**Webpage Examples Approach**
Node types	Datatype Function Parameter Category Document	Chunk Document	Function
Relationships	FUNCTION_OF HAS_PARAMETER BELONGS_TO	PART_OF	USES

	IS_DATATYPE RE- TURNS_DATATYPE		
Property keys	id name description function_id python vector_script return datatype	Id Text TextEmbedding	page_url

Furthermore, 112 datatype nodes were added, connected to parameters by 1,821 “IS_DATATYPE” relations and to functions by 2,865 “RETURNS_DATATYPE” relations.

While the deterministic approach provided a reliable framework for extracting well-defined elements, some challenges emerged. The datatype extraction, explained in chapters 4.1.2 and 5.1.2, faced challenges with multi-line return values, resulting in inconsistent node creation. For instance, the "Boolean" return type for the "Centroid" function was correctly identified, but the additional return parameters were misclassified as input parameters. This indicates a need for improved parsing techniques to handle multi-line returns effectively. In Table 5, the original function from the documentation is shown (vs.txt); on the other side, the extracted function (vs.json), the sequence is marked in blue. This is not only an issue from the “txtToJSON.py” parser, but it as well happened for the automated approach, where the entities are extracted from the LLM directly from the vs.txt file. This is described in chapter 6.2.1 and visualized in Figure 25.

Table 5 – Comparison ws2GetToolInfo-Function in .txt with .json

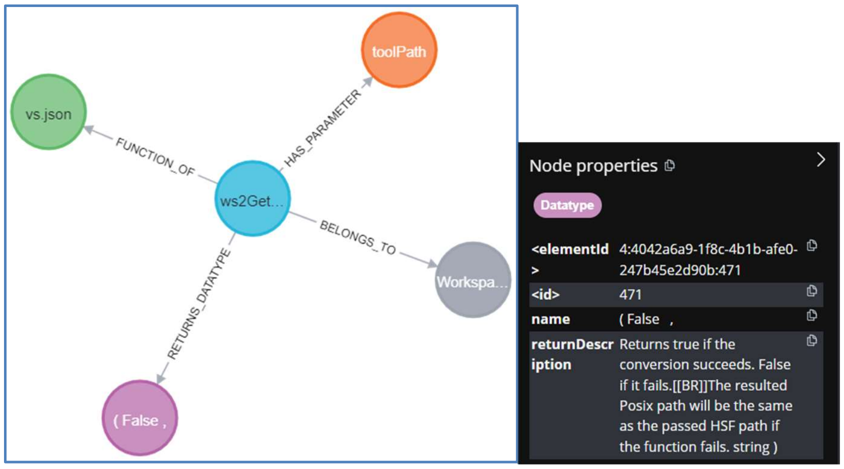
vs.txt	vs.json

`def **ws2GetToolInfo**(`
`        **toolPath**   # DYNARRAY[] of CHAR -`
`        ):`
`    '''`
`        **Python**: (BOOLEAN, outDisplayName, outShortcutKey, outShortcutKeyModifier, outResourceID) =`
	`{`
	`    "**id**": 2822,`
	`    "**FunctionName**": "ws2GetToolInfo",`
	`    "**InputParameters**": [`
	`        {`
	`            "name": "toolPath",`
	`            "datatype": "DYNARRAY[] of CHAR",`
	`            "description": ""`

```
vs.ws2GetToolInfo(toolPath)
       VectorScript: FUNCTION
ws2GetToolInfo(toolPath:DYNARRAY
of CHAR; VAR
outDisplayName:DYNARRAY of CHAR;
VAR outShortcutKey:CHAR; VAR
outShortcutKeyModifier:INTEGER;
VAR outResourceID:INTEGER) :
BOOLEAN;
       Category: Workspaces
       Workspace advanced APIs.
Return the tool information at
the specified index of the parent
tool at the specified path. See
'ws2GetToolsCnt'.
    '''
    pass
    return ( False   , #
             'string',
             'a'     ,
             0       ,
             0        )
```

```
        }
    ],
    "Return": "( False   ,",
    "ReturnDescription": "string,
a     , 0        , 0        )",
    "description": "Workspace
advanced APIs. Return the tool
information at the specified
index of the parent tool at the
specified path. See
'ws2GetToolsCnt'.",
    "Python": "(BOOLEAN,
outDisplayName, outShortcutKey,
outShortcutKeyModifier,
outResourceID) =
vs.ws2GetToolInfo(toolPath)",
    "VectorScript": "FUNCTION
ws2GetToolInfo(toolPath:DYNARRAY
of CHAR; VAR
outDisplayName:DYNARRAY of CHAR;
VAR outShortcutKey:CHAR; VAR
outShortcutKeyModifier:INTEGER;
VAR outResourceID:INTEGER) :
BOOLEAN;",
    "Category": "Workspaces"
},
```

The function is displayed in the graph in the following Figure 18.



<elementId>	4:4042a6a9-1f8c-4b1b-afe0-247b45e2d90b:471
<id>	471
name	(False ,
returnDescription	Returns true if the conversion succeeds. False if it fails.[[BR]]The resulted Posix path will be the same as the passed HSF path if the function fails. string)

Figure 18 - ws2GetToolInfo Functions with the properties of the return datatype

In Figure 19, all the functions are displayed connected to the false return type.

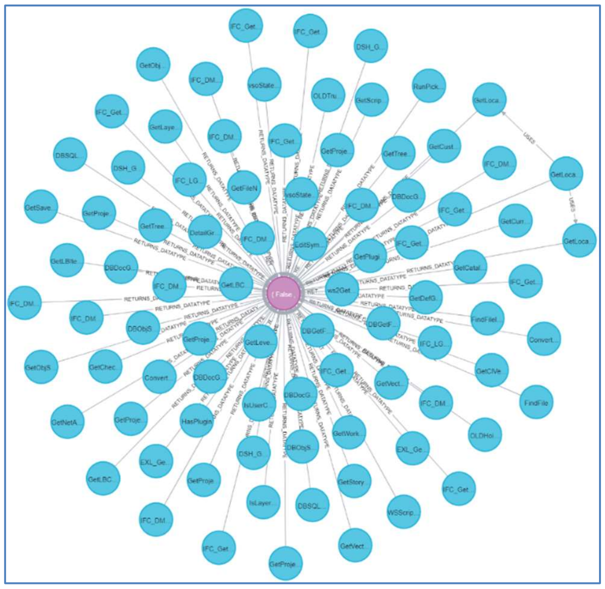


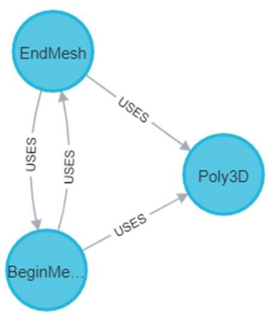
Figure 19 - Functions connected to the datatype "(False ,"

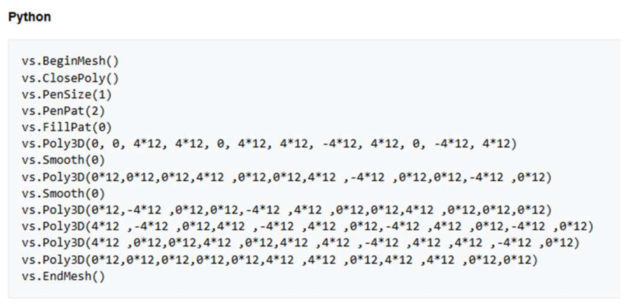
Another unexpected but beneficial outcome emerged from merging the connected example function with the graph: generating 18 additional function nodes not originally documented in the Python API documentation of Vectorworks. These newly identified functions expand the graph's scope and highlight the potential for discovering undocumented functionalities through this method.

The following functions were added to the graph:

- Message, Poly, Concat, InsertChoice, GetSelChoice
- Writeln, WriteLn, Read, ReadLn, RemoveGeoref
- message, Handle, EndText, BeginText, Poly3D
- GetPointAndParameter, GetVSVar, SetVSVar

In the following Figure 20, the function “Poly3D” is visualized as connected to two other functions of the graph, and the link to the web page example where the use of the functions becomes directly visible.





Python

```
vs.BeginMesh()
vs.ClosePoly()
vs.PenSize(1)
vs.PenPat(2)
vs.FillPat(0)
vs.Poly3D(0, 0, 4*12, 4*12, 0, 4*12, 4*12, -4*12, 4*12, 0, -4*12, 4*12)
vs.Smooth(0)
vs.Poly3D(0*12,0*12,0*12,4*12 ,0*12,0*12,4*12 ,-4*12 ,0*12,0*12,-4*12 ,0*12)
vs.Smooth(0)
vs.Poly3D(0*12,-4*12 ,0*12,0*12,-4*12 ,4*12 ,0*12,0*12,4*12 ,0*12,0*12,0*12)
vs.Poly3D(4*12 ,-4*12 ,0*12,4*12 ,-4*12 ,4*12 ,0*12,-4*12 ,4*12 ,0*12,-4*12 ,0*12)
vs.Poly3D(4*12 ,0*12,0*12,4*12 ,0*12,4*12 ,4*12 ,-4*12 ,4*12 ,4*12 ,-4*12 ,0*12)
vs.Poly3D(0*12,0*12,0*12,0*12,0*12,4*12 ,4*12 ,0*12,4*12 ,4*12 ,0*12,0*12)
vs.EndMesh()
```

Figure 20 - VS:Poly3D in the graph (left) and in the example (right)

6.1.2 Agent Performance

This section examines the retrieval performance from the graph-based RAG agent in Approach A, which was tested using two LLMs, “llama:3.2” and “gpt-4o-mini.” The agent's performance was evaluated by asking questions of varying complexity, validating the answers against the BIM documentation, and checking the executability of the code generated by the agent. The agent answered 36 questions across three complexity levels: single-hop, multi-hop, and ambiguous queries. Accuracy varied significantly between complexity levels:

Responses from Ollama-based RAG Agent

The implementation effort for the Ollama model consisted primarily of defining the prompt and structuring the expected results so that the LLM could interpret them effectively. The model achieved its best performance when answering questions with text responses. However, the frequency of coding implementations in the responses was lower than initially expected. Despite this, the Ollama model adhered closely to the agent's prompt description and invoked the required tools as instructed.

Table 6 summarizes the overall results and provides an accuracy rating for both text and code responses.

Text accuracy is presented as:

- percentage of questions correctly answered out of all questions asked
- percentage of questions correctly answered from the subset of questions where an answer was provided.

The same applies to the code accuracy:

- percentage of correct code suggestions out of all questions asked
- percentage of correct code suggestions for answers that included a code example, as not all answers included code implementations.

Table 6 - Result from graph-based Agent RAG – Ollama "lama3.2"

Level	Questions Asked	Questions Answered	Correct Text Answers	Answer includes Code example	Correct Code Suggestions	Working Python Code Example	Text Accuracy (Asked vs. Correct)	Text Accuracy (Answered vs. Correct)	Code Accuracy (Asked vs. Correct)	Code Accuracy (Code vs. Correct)
1	12	8	6	0	0	0	0,500	0,750	0,000	
2	12	8	8	6	5	5	0,667	1,000	0,417	0,833
3	12	9	9	8	7	7	0,750	1,000	0,583	0,875
	36	**25**	**23**	**14**	**12**	**12**				

From the results presented, it becomes clear that the questions from level 3 are answered more frequently correctly than those from level 1. This discrepancy can be attributed to the agent's struggles in generating accurate and valuable Cypher queries, particularly for Level 1 questions. The questions from levels 2 and 3 are more often correct as the agent retrieves the results from the semantic search tool here.

The following examples illustrate how the agent responded to questions and why it failed. The questions from Level 1 were answered correctly when the Graph Query produced an accurate Cypher query and returned valid results. However, errors occurred when the agent failed to detect the correct entities. This happened for the question: "List all nodes belonging to Textures." Where the following query was returned:

```
MATCH (f:Function {name: "Textures"})
OPTIONAL MATCH (f)-
[r:FUNCTION_OF|BELONGS_TO|RETURNS_DATATYPE|HAS_PARAMETER|USES]-
(d:Document)
RETURN f, d, r
```

The problem is that "Textures" is not a function but a Category, leading to an empty result. Instead of acknowledging the issue, the LLM fabricated an answer, highlighting its limitation in entity recognition.

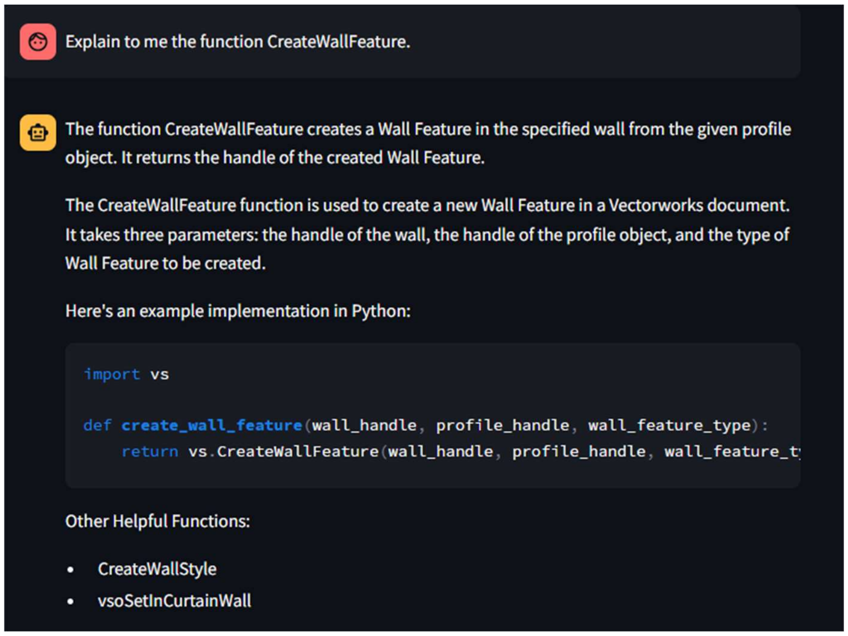
Furthermore, the agent occasionally failed entirely for questions requiring more complex Cypher queries. In such cases, the Ollama model's tool-handling capabilities proved insufficient. The toolchain frequently entered an endless reasoning loop, concluding that it was impossible to return a valid result. This limitation demonstrates the need for improved error-handling mechanisms and enhanced query-generation logic, as the OpenAI-based agent does not fail to answer the same questions.

Code Suggestion from Ollama-based RAG Agent

The code suggestion provided by the Ollama-based RAG Agent closely adhered to the examples from the documentation. Even though they are not as complex as they could be, they were mostly correct when added to the answer. The answers from the agent were correct when they stuck to the prompt description and generated the correct graph query with the correct result. For example, for the question, “Explain to me the function CreateWallFeature.” The Semantic Search tool and the Graph Query tool were used. The Graph Query tool generated the correct Cypher query and returned the result to the agent.

```
MATCH (f:Function {name: "CreateWallFeature"})
RETURN f
```

The agent then generates the answer shown in Figure 21.



Explain to me the function CreateWallFeature.

The function CreateWallFeature creates a Wall Feature in the specified wall from the given profile object. It returns the handle of the created Wall Feature.

The CreateWallFeature function is used to create a new Wall Feature in a Vectorworks document. It takes three parameters: the handle of the wall, the handle of the profile object, and the type of Wall Feature to be created.

Here's an example implementation in Python:

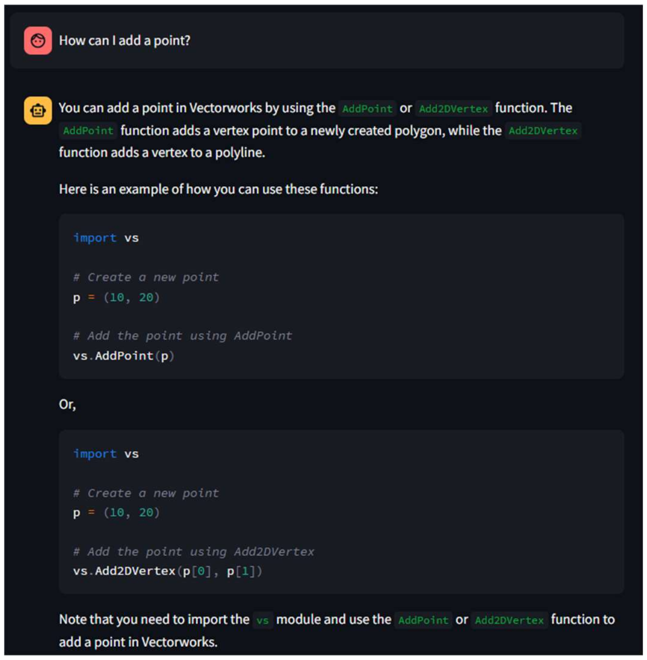
```
import vs

def create_wall_feature(wall_handle, profile_handle, wall_feature_type):
    return vs.CreateWallFeature(wall_handle, profile_handle, wall_feature_t
```

Other Helpful Functions:

- CreateWallStyle
- vsoSetInCurtainWall

Figure 21 – Example response from the Ollama-based Agent

The coding suggestions for more abstract questions from Level 3, including example values, were more complex. “How can I add a point?” was answered by offering a coding suggestion with example values. This shows the general ability of the agent to provide based on the graph-data coding examples.



How can I add a point?

You can add a point in Vectorworks by using the `AddPoint` or `Add2DVertex` function. The `AddPoint` function adds a vertex point to a newly created polygon, while the `Add2DVertex` function adds a vertex to a polyline.

Here is an example of how you can use these functions:

```
import vs

# Create a new point
p = (10, 20)

# Add the point using AddPoint
vs.AddPoint(p)
```

Or,

```
import vs

# Create a new point
p = (10, 20)

# Add the point using Add2DVertex
vs.Add2DVertex(p[0], p[1])
```

Note that you need to import the `vs` module and use the `AddPoint` or `Add2DVertex` function to add a point in Vectorworks.

Figure 22 – Response from the agent to a Level 3 question

Responses from OpenAI-based RAG Agent

Now the responses from the graph-based RAG Agent using the OpenAI-LLM “gpt-4o-mini” are analyzed. In the following table, the overall results are shown. It evaluates the accuracy of the text and code answers. The text accuracy is separated into the percentage of questions correctly answered from all asked questions and the percentage of questions correctly answered from the questions answered at all. The same applies to code accuracy, divided into the percentage of correct code suggestions from all asked questions and the percentage of correct code suggestions when the answer includes a code example, as not all answers have code suggestions.

Table 7 – Results from graph-based RAG Agent - OpenAI “gpt-4o-mini”

Level	Questions Asked	Questions Answered	Correct Text Answers	Answer includes Code example	Correct Code Suggestions	Working Python Code Example	Text Accuracy (Asked vs. Correct)	Text Accuracy (Answered vs. Correct)	Code Accuracy (Asked vs. Correct)	Code Accuracy (Code vs. Correct)
1	12	12	9	4	4	1	0,750	0,750	0,333	1,000
2	12	12	12	12	12	12	1,000	1,000	1,000	1,000
3	12	12	11	10	9	8	0,917	0,917	0,750	0,900
	36	**36**	**30**	**25**	**24**	**21**				

Table 7 shows that the model performed the highest for questions from Level 2. Overall, the agent answered all questions with a high percentage of correct code suggestions.

Starting with questions on level 1, here, more solid queries and answers were produced by the model “GPT-4o-mini” than by Ollama. However, it struggled with the same question as the Ollama model: “List all nodes belonging to Textures.”

For this question, GPT-4o-mini generated the following query:

```
MATCH (f:Function)
WHERE toLower(f.name) CONTAINS 'textures'
RETURN f
```

This query failed because "Textures" is a Category, not a Function, leading to incorrect results. A similar issue arose: “Find all the parameters using the datatype INTEGER.”

for which the following query was generated:

```
MATCH (f:Function)-[:RETURNS_DATATYPE]->(d:Datatype)
WHERE toLower(d.name) = toLower("INTEGER")
RETURN f, d
```

Although syntactically correct, the query did not align with the question's intent, resulting in an incorrect answer.

Overall, “GPT-4o-mini” showed greater abilities in generating Cypher queries based on prompts and entities provided. It handled the inconstancies in the names by using this kind of queries:

Question: “List all functions returning HANDLE.”

Generated Query:

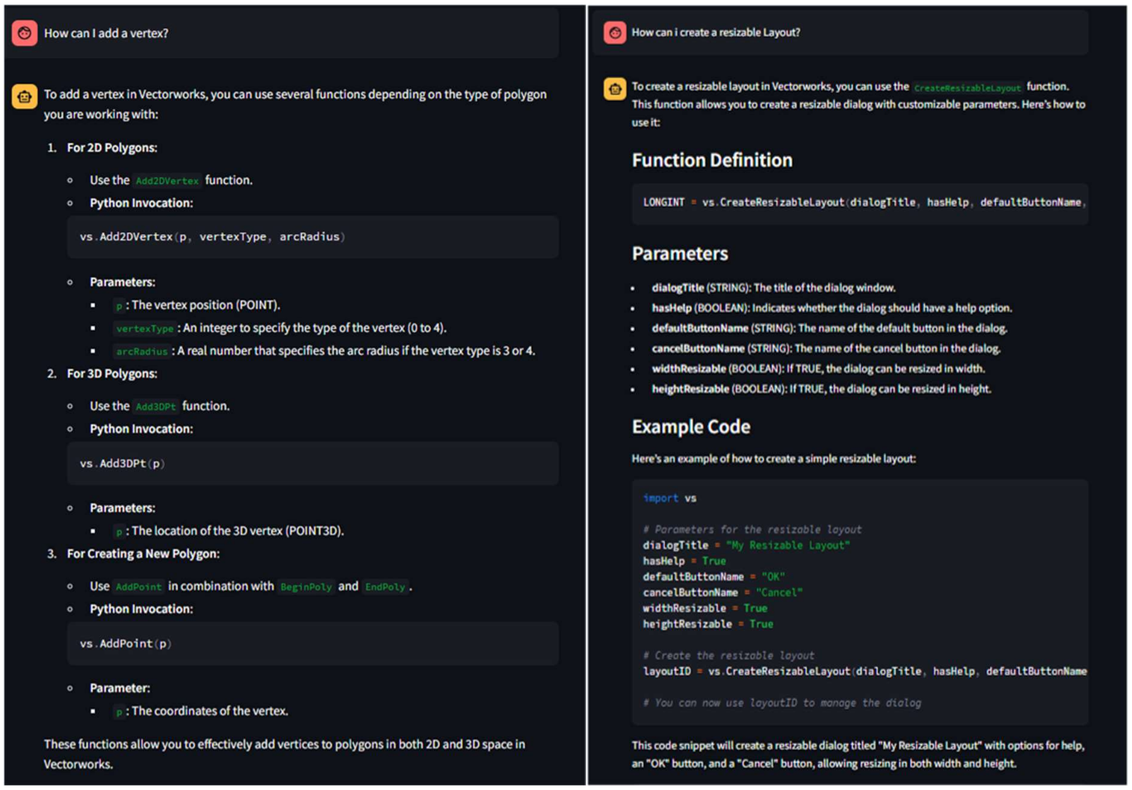
```
MATCH (f:Function)
WHERE toLower(f.return) = 'handle'
RETURN f
```

Continuing with the questions from level 2, where the agent showed great results in the semantic search with the vector index, combined with sufficient Graph Queries. The agent mainly used the “semantic-search tool,” and for more complex questions where Ollama previously failed, it retrieved the nodes from the graph using the Graph Query tool. The agent used the semantic-search tool alone in 8 of 12 questions and together with the graph-query tool in 4 of 12 questions.

At level 3, questions were not asked about a specific function but rather about understanding the connections between the functions, and the results were not as good as those from level 2, but still on a high level.

Code Suggestions from OpenAI-based RAG Agent

The agent returns high-level coding suggestions with complex examples and a good explanation of functions and parameters. The answers were more detailed than those provided by the Ollama model, illustrated in Figure 23.



How can I add a vertex?

To add a vertex in Vectorworks, you can use several functions depending on the type of polygon you are working with:

1. **For 2D Polygons:**
   - Use the `Add2DVertex` function.
   - **Python Invocation:**

     ```
     vs.Add2DVertex(p, vertexType, arcRadius)
     ```

   - **Parameters:**
     - `p` : The vertex position (POINT).
     - `vertexType` : An integer to specify the type of the vertex (0 to 4).
     - `arcRadius` : A real number that specifies the arc radius if the vertex type is 3 or 4.
2. **For 3D Polygons:**
   - Use the `Add3DPt` function.
   - **Python Invocation:**

     ```
     vs.Add3DPt(p)
     ```

   - **Parameters:**
     - `p` : The location of the 3D vertex (POINT3D).
3. **For Creating a New Polygon:**
   - Use `AddPoint` in combination with `BeginPoly` and `EndPoly` .
   - **Python Invocation:**

     ```
     vs.AddPoint(p)
     ```

   - **Parameter:**
     - `p` : The coordinates of the vertex.

These functions allow you to effectively add vertices to polygons in both 2D and 3D space in Vectorworks.

How can i create a resizable Layout?

To create a resizable layout in Vectorworks, you can use the `CreateResizableLayout` function. This function allows you to create a resizable dialog with customizable parameters. Here's how to use it:

Function Definition

```
LONGINT = vs.CreateResizableLayout(dialogTitle, hasHelp, defaultButtonName,
```

Parameters

- **dialogTitle** (STRING): The title of the dialog window.
- **hasHelp** (BOOLEAN): Indicates whether the dialog should have a help option.
- **defaultButtonName** (STRING): The name of the default button in the dialog.
- **cancelButtonName** (STRING): The name of the cancel button in the dialog.
- **widthResizable** (BOOLEAN): If TRUE, the dialog can be resized in width.
- **heightResizable** (BOOLEAN): If TRUE, the dialog can be resized in height.

Example Code

Here's an example of how to create a simple resizable layout:

```
import vs

# Parameters for the resizable layout
dialogTitle = "My Resizable Layout"
hasHelp = True
defaultButtonName = "OK"
cancelButtonName = "Cancel"
widthResizable = True
heightResizable = True

# Create the resizable layout
layoutID = vs.CreateResizableLayout(dialogTitle, hasHelp, defaultButtonName

# You can now use layoutID to manage the dialog
```

This code snippet will create a resizable dialog titled "My Resizable Layout" with options for help, an "OK" button, and a "Cancel" button, allowing resizing in both width and height.

Figure 23 - Coding suggestions at Level 3 questions provided by OpenAI-based agent

Overall Comparison Approach A: Ollama with OpenAI

The Ollama-based agent stuck to the agent prompt and used the tools accordingly but failed to retrieve correct answers due to incorrect queries. Meanwhile, the GPT-based agent used the semantic search tool, combined with the graph query tool, to retrieve, in the majority of cases, the correct answers.

6.2 Approach B

This section describes the results obtained from Approach B, starting with an evaluation of the generated BIM-API-Graph, followed by an assessment of the retrieval performance of the graph-based RAG agent. The approach uses the OpenAI model “gpt-4o-mini” for graph generation and retrieval. This approach evaluates a different method and their effectiveness in handling different question types. These questions, designed to test the agent's capabilities across multiple levels of complexity, were validated by cross-referencing the results with the API documentation. Additionally, agent-generated code implementations were tested for executability and accuracy. The findings are summarized and discussed to highlight the strengths and limitations of the approach in meeting the research objectives.

6.2.1 Graph Structure

Approach B provided a method to connect the graph more, show in Figure 24. Still, as the splitting was not defined with a CharacterTextSplitter, no separator was specified, resulting in functions being connected to more than one chunk node.

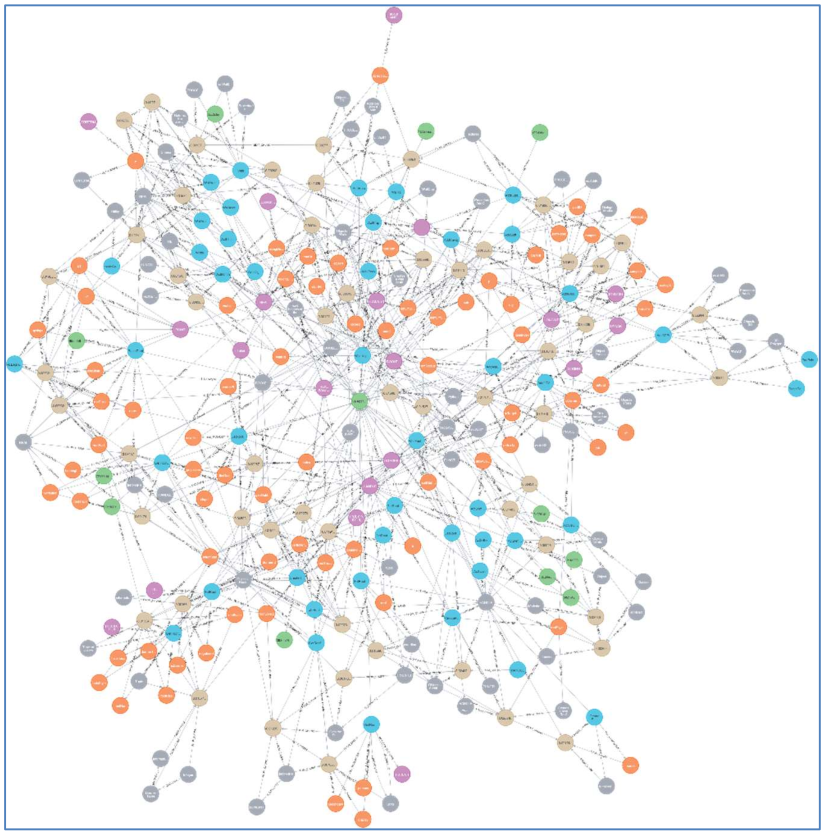


Figure 24 – Graph from Approach B

Most of the datatypes were recognized correctly, but some, especially the multi-line return types, were not. This can be seen in the following Figure 25, which visualizes the "Centroid" function. The return type "Boolean" was identified correctly, but the other two parameters were added as input parameters to the function.

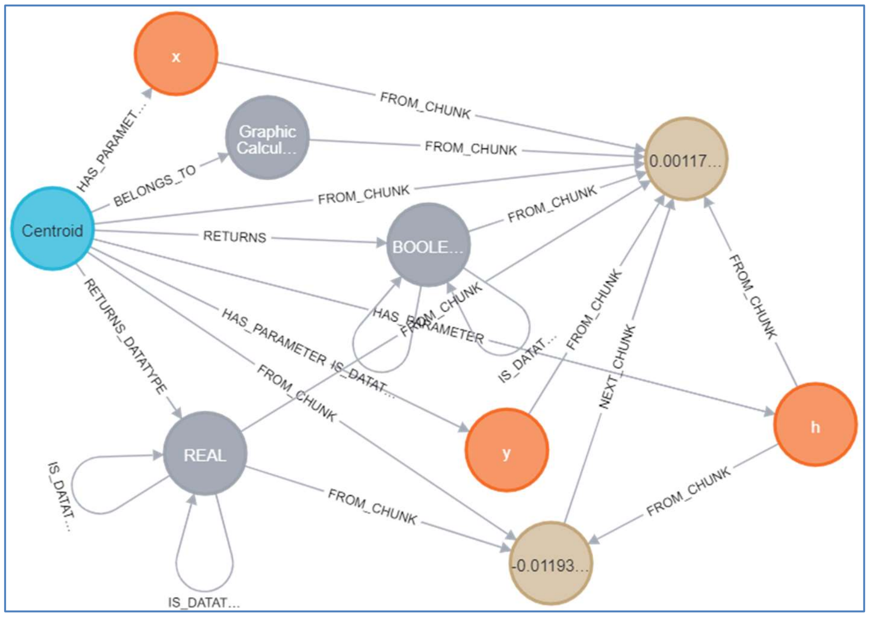


Figure 25 - Centroid Function in the BIM-API-Graph-B

In comparison, this is the function in the API:

```
def Centroid(
       h  # HANDLE -
       ):
    '''
       Python: (BOOLEAN, x, y) = vs.Centroid(h)
       VectorScript: FUNCTION Centroid(h:HANDLE; VAR x:REAL; VAR y:REAL)
: BOOLEAN;

       Category: Graphic Calculation
       Returns the centroid of the object. Returns false if an
unsupported object type is supplied.
    '''
    pass
    return ( False   , #
             0.0     ,
             0.0      )
```

6.2.2 Agent Performance

The evaluation of the RAG agent focused on its ability to retrieve and present accurate information using the knowledge graph. This included answering queries, demonstrating multi-hop reasoning capabilities, and generating coding suggestions based on API documentation.

During the evaluation, it was found that the retrieval process for the entire graph was not feasible due to the token limitations in the “gpt-4o-mini” model. In particular, the system encountered the following error:

```
neo4j_graphrag.exceptions.LLMGenerationError: Error code: 429 - {'error':
{'message': 'Request too large for gpt-4o-mini in organization org-
iNj2mullHIZkHPT4TEyp85J4 on tokens per min (TPM): Limit 200000, Requested
308538. The input or output tokens must be reduced in order to run
successfully. Visit https://platform.openai.com/account/rate-limits to
learn more.', 'type': 'tokens', 'param': None, 'code':
'rate_limit_exceeded'}}
```

The evaluation was performed on a smaller, representative graph segment to address this limitation. This segment comprised 500 lines selected from the API document to ensure the most relevant queries were covered. The selected rows were embedded using OpenAI's “text-embedding-ada-002” model, and the graph was built using the kg_builder method described in chapter 4.1.3.

Responses

The retrieval performance was analyzed based on the selected API lines. Table 8 evaluates the accuracy of the text and code answers. The text accuracy is separated into the percentage of questions correctly answered from all asked questions and the percentage of questions correctly answered from the questions answered at all. The same applies to code accuracy, divided into the percentage of correct code suggestions from all asked questions and the percentage of correct code suggestions when the answer includes a code example, as not all answers have code suggestions. This highlights the agent's ability to retrieve relevant information, its effectiveness in generating executable code, and areas where performance can be improved.

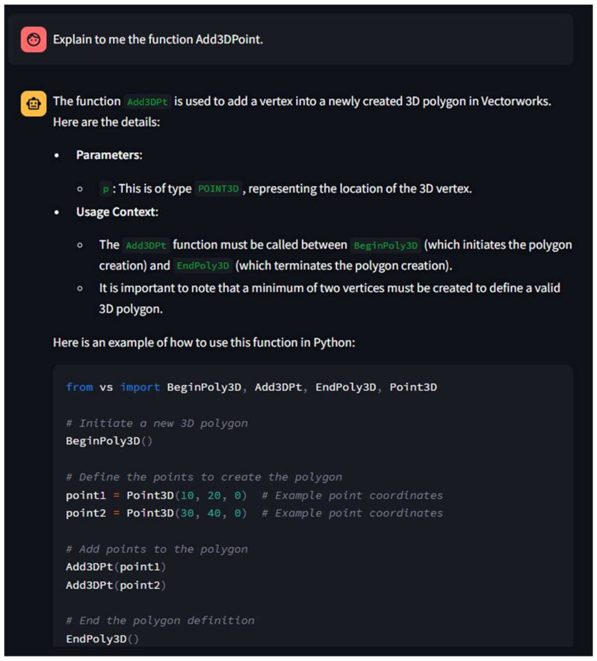
Table 8 – Results Approach B

Level	Questions Asked	Questions Answered	Correct Text Answers	Answer includes Code example	Correct Code Suggestions	Working Python Code Example	Text Accuracy (Asked vs. Correct)	Text Accuracy (Answered vs. Correct)	Code Accuracy (Asked vs. Correct)	Code Accuracy (Code vs. Correct)
1	11	11	10	0	0	0	0,909	0,909	0,000	
2	5	5	5	4	2	2	1,000	1,000	0,400	0,500
3	6	6	4	4	2	2	0,667	0,667	0,333	0,500
	22	**22**	**19**	**8**	**5**	**4**				

The graph-based agent accurately answered level 1 questions and only gave an incorrect answer in one case. The agent relied primarily on the Graph Query tool, which was used in seven cases, while the Semantic Search tool was used in four cases. Compared to Approach A, the results of the GraphRAG retrieval showed a significant improvement in terms of accuracy and relevance. This result confirms the usability of the GraphRAG tool for entity retrieval using the multi-hop approach.

For level 2 questions, the agent successfully provided accurate text answers. However, 50% of the generated code suggestions were not functional. As shown in Figure 26, although the text response was mostly correct, it contained a reference to a non-existent function, Point3D, in the Vectorworks API. This highlights the need to define nodes using a deterministic approach further.

The answers to the level 3 questions were partially correct. The textual answers were often as expected, and when they were correct, the agent provided more sophisticated and detailed answers. However, the code suggestions were not of the expected quality. Although the modified retrieval tool successfully identified relevant nodes, it did not consistently generate executable code. This indicates that additional efforts are required to improve the robustness of the agent's code generation capabilities.



Explain to me the function Add3DPoint.

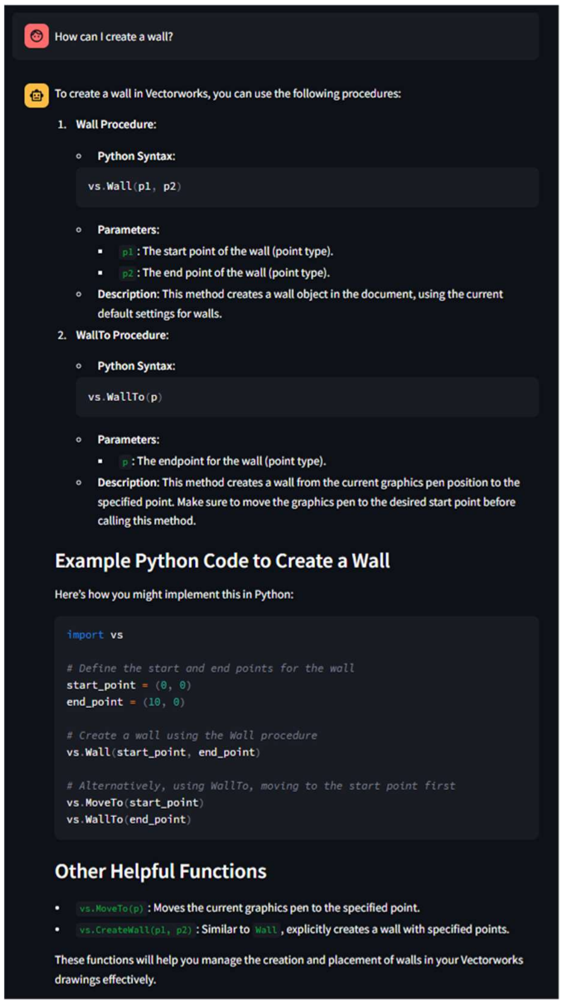
The function `Add3DPt` is used to add a vertex into a newly created 3D polygon in Vectorworks. Here are the details:

- **Parameters:**
  - `p` : This is of type `POINT3D` , representing the location of the 3D vertex.
- **Usage Context:**
  - The `Add3DPt` function must be called between `BeginPoly3D` (which initiates the polygon creation) and `EndPoly3D` (which terminates the polygon creation).
  - It is important to note that a minimum of two vertices must be created to define a valid 3D polygon.

Here is an example of how to use this function in Python:

```
from vs import BeginPoly3D, Add3DPt, EndPoly3D, Point3D

# Initiate a new 3D polygon
BeginPoly3D()

# Define the points to create the polygon
point1 = Point3D(10, 20, 0)  # Example point coordinates
point2 = Point3D(30, 40, 0)  # Example point coordinates

# Add points to the polygon
Add3DPt(point1)
Add3DPt(point2)

# End the polygon definition
EndPoly3D()
```

Figure 26 - Question and Answer - Level 2



How can I create a wall?

To create a wall in Vectorworks, you can use the following procedures:

1. **Wall Procedure:**
   - **Python Syntax:**
     ```
     vs.Wall(p1, p2)
     ```
   - **Parameters:**
     - `p1` : The start point of the wall (point type).
     - `p2` : The end point of the wall (point type).
   - **Description:** This method creates a wall object in the document, using the current default settings for walls.
2. **WallTo Procedure:**
   - **Python Syntax:**
     ```
     vs.WallTo(p)
     ```
   - **Parameters:**
     - `p` : The endpoint for the wall (point type).
   - **Description:** This method creates a wall from the current graphics pen position to the specified point. Make sure to move the graphics pen to the desired start point before calling this method.

Example Python Code to Create a Wall

Here's how you might implement this in Python:

```
import vs

# Define the start and end points for the wall
start_point = (0, 0)
end_point = (10, 0)

# Create a wall using the Wall procedure
vs.Wall(start_point, end_point)

# Alternatively, using WallTo, moving to the start point first
vs.MoveTo(start_point)
vs.WallTo(end_point)
```

Other Helpful Functions

- `vs.MoveTo(p)` : Moves the current graphics pen to the specified point.
- `vs.CreateWall(p1, p2)` : Similar to `Wall` , explicitly creates a wall with specified points.

These functions will help you manage the creation and placement of walls in your Vectorworks drawings effectively.

Figure 27 - Question and Answer - Level 3

The token limitation imposed by “gpt-4o-mini” required an adaptation of the evaluation methodology to a smaller data set. This adaptation emphasizes the need for scalability in future graph retrieval systems. A smaller graph segment effectively demonstrated the agent's capabilities, especially when answering many queries. While the accuracy of the text search was consistently high, the correctness of the code suggestions varied, indicating room for improvement in creating robust implementation examples.

6.3 Comparison of Approaches

This evaluation compares the performance of the graph-based RAG agents with the different approaches and levels of query complexity. The following tables summarize the results and focus on the text accuracy and the quality of the code suggestions.

Table 9 below compares the correct text answers (Asked vs. Correct) to identify the overall best-performing model.

Table 9 – Compares Text Accuracy

Correct Text Answers	Approach A – Ollama	Approach A – OpenAI	Approach B – OpenAI
Level 1	0,500	0,750	0,909
Level 2	0,667	1,000	0,800
Level 3	0,750	0,917	0,667

The following Table 10 compares the correct code suggestions answers to identify the overall best-performing model regarding code. The correct code suggestions are evaluated against the total answer, including code implementation.

Table 10 – Compares Code Accuracy

Correct Code Suggestions	Approach A – Ollama	Approach A – OpenAI	Approach B – OpenAI
Level 1		1,000	
Level 2	0,833	1,000	0,500
Level 3	0,875	0,900	0,500

Approach A

The OpenAI model outperformed the other configurations in both text accuracy and code suggestions. It showed robust performance across all query levels, achieving 100% text accuracy for level 2 questions and high-quality code generation. The Ollama model performed well on text accuracy for higher-level queries, but its ability to suggest code lagged behind OpenAI. The deterministic graph structure proved to be a critical factor in the success of Approach A, as it ensures reliable relationships and well-defined nodes.

Approach B

Showed promise in terms of flexibility and adaptability, especially when processing unstructured data. However, the results for both text accuracy and code suggestions were inconsistent, especially for more complex level 3 queries. The LLM-generated graph lacked the precision and structural reliability of the deterministic graph in Approach A, leading to challenges in finding accurate and executable code.

Summary

From the results, it is reasonable to conclude that the graph's structure plays a crucial role. The deterministic graph in approach A provided a more stable basis for querying and code generation than the LLM-generated graph in approach B. Furthermore, the LLM selection was shown to be essential. OpenAI's model outperformed Ollama in both text and code accuracy, suggesting that the choice of LLM significantly affects the agent's performance. The lower performance of Approach B in code generation indicates the need for improved graph construction and the integration of richer code examples and extract with the automated approach, as well as node properties with code implementations for each function node similar to Approach A.

Approach A's deterministic graph is better suited for large amounts of data as its structure ensures reliability and avoids redundant relationships. Approach B, which relies on LLMs for graph generation, can scale better with unstructured or diverse APIs but requires optimization for larger data sets. A careful combination of both Approaches can potentially reveal further capabilities.

Approach B offers higher adaptability for different APIs as it relies on LLMs for graph creation, which makes it flexible for different data formats and structures. Approach A requires manual preprocessing and schema creation, which limits adaptability but ensures higher precision.

In summary, Approach A proved to be the best-performing method, especially when combined with the OpenAI LLM. Its robust graph structure provided higher accuracy and better code suggestions, making it ideal for precision and reliability scenarios. Meanwhile, Approach B showed potential for handling unstructured data and adapting to different APIs but requires further refinement of graph construction and query methods.

7 Discussion

Based on the experimental results in the previous chapter in this chapter the answering of the research questions defined in Chapter 1.2. will be discussed in this chapter.

What kind of knowledge graph schema design can effectively represent BIM API documentation?

An effective knowledge graph schema for representing BIM API documents must maintain a balance between structural reliability, semantic depth, and practical usability. To accomplish this, it is important to represent specific nodes with the most essential properties from the BIM API. Such important nodes are functions, parameters, and data types. Functions nodes are primary nodes with properties such as function name, description, return type, and code implementation details. Parameter nodes for function parameters capture properties such as parameter names, data types, and descriptions. Separate nodes for data types allow relationships between functions that share input or output types. The second most important element of the BIM-API-Graph are the relationships between those nodes, as they reveal the dependency within the entities. Relationships like [HAS_PARAMETER], [RETURNS_DATATYPE], and [USES] are essential, offering developers a way to navigate and understand complex documentation. The HAS_PARAMETER relation links function nodes to parameter nodes, clearly defining input requirements. RETURNS_DATATYPE relation connects functions to their return types, facilitating queries about output characteristics. A significant enhancement comes from integrating implementation examples through [USES] relationships. These connections illustrate connections between functions from more complex use cases and coding practices, making the graph more practical and relevant for developers. By linking examples to specific functions and their dependencies, the knowledge graph explains how functions work in isolation and interact in typical workflows, which is crucial for problem-solving and code generation.

The hybrid generating combines deterministic key-value node relationships with semantically enriched embeddings and practical usage links. The deterministic approach ensures reliability and provides a consistent foundation. Semantically enriched enables advanced reasoning and context-aware querying. This combination effectively balances reliability, depth, and usability for BIM API documentation. Additionally, the ability to support multi-hop queries through this schema allows the tracing of complex relationships between functions, parameters, and data types. This way facilitates insights beyond the textual documentation and enables dynamic queries.

What are the comparative advantages of Knowledge Graphs over raw textual data when utilized as a knowledge base for an LLM-based RAG Agent in the context of BIM API queries?

Knowledge graphs (KGs) offer several advantages over raw text data when used as a knowledge base for a graph-based RAG Agent in the context of BIM API queries. These advantages result from their ability to structure, connect, and enrich data in a way impossible with raw text.

Raw textual BIM API often lacks structure, and it is challenging to identify relationships and dependencies in the data. Whereas Knowledge Graphs organize data into well-defined nodes (e.g., functions, parameters, datatypes) and relationships (e.g., [HAS_PARAMETER], [USES], [RETURNS_DATATYPE]), enabling straightforward navigation and retrieval.

Another advantage of knowledge graphs over raw text is that text requires LLMs to infer connections implicitly, which can lead to errors or hallucinations, whereas KG supports explicit multi-hop queries that allow the agent to track and retrieve related information, such as chains of dependent functions required for a specific workflow.

Furthermore, the study showcased improved coding suggestions from the LLM-based RAG agent when using the node properties, such as the Python property, to better understand the functions' implementation.

What challenges emerge while transforming BIM API documentation into a Knowledge Graph, and how can these be addressed?

During the transformation of BIM API documentation, different challenges emerged. Extracting relevant elements from API documentation, notably when the documentation lacks consistency, includes multi-line elements (e.g., complex return values or nested parameters), or uses varying formats across APIs. Solutions to address these issues were to develop custom parsers with regular expressions tailored to the structure of the API documentation. Another important step is the preprocessing of the documentation to convert it into machine-readable formats (e.g., JSON), ensuring uniformity. Furthermore, LLMs can support identifying and parsing unstructured elements dynamically.

Another issue occurred when transforming the documentation with an LLM into a KG using the LLM to detect entities, as the identifying entities (e.g., functions, parameters, datatypes) and relationships (e.g., [HAS_PARAMETER], [USES]) accurately can be difficult, especially when the documentation uses inconsistent naming conventions or descriptions. Here, the solution is to use predefined labels and relationships to guide the LLM on which elements are relevant entities to detect.

A further challenge is to enrich the graph with context, such as real-world usage examples while avoiding irrelevant or redundant relationships. The solution appears to scrape and integrate implementation examples from developer forums, API websites, or documentation snippets. Using the [USES] relationships to explicitly link functions to practical coding scenarios, enabling context-aware querying.

Summarize

In summary, the study showed that an effective knowledge graph schema for BIM API documentation must balance structural reliability, semantic depth, and usability by organizing data into nodes (e.g., functions, parameters, data types) and relationships (e.g., [HAS_PARAMETER], [USES], [RETURNS_DATATYPE]). Knowledge graphs outperform raw text data by enabling explicit multi-hop queries, reducing inference errors, and supporting enriched, dynamic insights that improve the functionality of LLM-based RAG agents. Challenges in transforming BIM API documentation into knowledge graphs, such as inconsistent formatting and enriching graphs with relevant context, are addressed through custom parsers, LLM-guided entity recognition, and integration of real-world coding examples.

8 Conclusion and Future Work

This study aimed to improve the interpretability of API documentation for BIM (Building Information Modeling) authoring software using graph-based approaches. To accomplish this, the usability of API documentation was significantly improved by introducing a hybrid graph generation methodology. The results presented in the previous chapter show that this approach effectively bridges the gap between complex technical documentation and practical usability by providing a structured tool for navigating API functionalities.

The hybrid graph generation method organizes documentation more effectively and facilitates improved code comprehension and integration for developers. The study's findings underscore the potential for graph-based representations to address longstanding challenges in API documentation usability, making it a valuable tool for enhancing the developer experience.

Future Work

Expanding the scope of this methodology to encompass a variety of APIs would provide quantitative insights into the types of API documentation that benefit most from graph-based representations. Comparative analyses across APIs from different domains can further validate and refine this approach.

It would be beneficial to incorporate more sophisticated examples beyond those available in web-based documentation to improve the relevance and complexity of LLM-generated code suggestions. Sourcing examples from developer forums and open-source projects could enrich the repository of coding samples.

Adding detailed coding examples as separate "Example Nodes" and linking them to corresponding function entities within the graph could further increase the usefulness of the documentation. This integration would provide developers with concrete implementation guidance, thereby enabling more complex and contextually relevant code suggestions from LLMs.

Future work could also focus on automating the graph generation process to handle large-scale API documentation efficiently. Investigating scalable solutions for real-time graph updates in response to API changes will be crucial.

Finally, conducting usability testing with a broad audience of developers would provide valuable feedback on the practical effectiveness of the proposed approach. Iterative refinements based on user feedback could ensure the solution remains robust and user-centered.

Through these extensions, the hybrid graph generation methodology could evolve into a comprehensive framework for API documentation, serving as a vital resource for developers and enhancing the overall productivity of software development workflows.

References

Abedu, S., Khatoonabadi, S., & Shihab, E. (2024, December 5). *Synergizing LLMs and Knowledge Graphs: A Novel Approach to Software Repository-Related Question Answering*. http://arxiv.org/pdf/2412.03815

Abioye, S. O., Oyedele, L. O., Akanbi, L., Ajayi, A., Davila Delgado, J. M., Bilal, M., Akinade, O. O., & Ahmed, A. (2021). Artificial intelligence in the construction industry: A review of present status, opportunities and future challenges. *Journal of Building Engineering*, *44*, 103299. https://doi.org/10.1016/j.jobe.2021.103299

Amatriain, X. (2024, January 24). *Prompt Design and Engineering: Introduction and Advanced Methods*. http://arxiv.org/pdf/2401.14423

Antropic. (2025, January 14). *Claude Sonnet 3.5 by Anthropic*. https://www.anthropic.com/

Barrasa, J., & Webber, J. (2023). *Building knowledge graphs: A practitioner's guide* (First edition). O'reilly. https://permalink.obvsg.at/AC17313964

Bloch, T., Borrmann, A [André], & Pauwels, P. (2023). Graph-based learning for automated code checking – Exploring the application of graph neural networks for design review. *Advanced Engineering Informatics*, *58*, 102137. https://doi.org/10.1016/j.aei.2023.102137

Bronzini, M. (2024). *Glitter or gold? Deriving structured insights from sustainability reports via large language models*. Springer Berlin Heidelberg.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., . . . Amodei, D. (2020, May 28). *Language Models are Few-Shot Learners*. http://arxiv.org/pdf/2005.14165

Chekalina, V., Razzigaev, A., Goncharova, E., & Kuznetsov, A. (2024, November 18). *Addressing Hallucinations in Language Models with Knowledge Graph Embeddings as an Additional Modality*. http://arxiv.org/pdf/2411.11531

Clark, K., Luong, M.-T., Le V, Q., & Manning, C. D. (2020, March 23). *ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators*. http://arxiv.org/pdf/2003.10555

Devlin, J., Chang, M.-W., Lee, K [Kenton], & Toutanova, K. (2018, October 11). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. http://arxiv.org/pdf/1810.04805

Du, C., Esser, S., Nousias, S [Stavros], & Borrmann, A [André]. (2024, August 15). *Text2BIM: Generating Building Models Using a Large Language Model-based Multi-Agent Framework*. http://arxiv.org/pdf/2408.08054

Du, C., Nousias, S [S.], & Borrmann, A [A.] (2024). Towards a copilot in BIM authoring tool using large language model based agent for intelligent human-machine interaction. In *Proc. of the 31th Int. Conference on Intelligent Computing in Engineering (EG-ICE)*. https://mediatum.ub.tum.de/node?id=1743921

Esser, S., Vilgertshofer, S., & Borrmann, A [André] (2023). Version control for asynchronous BIM collaboration: Model merging through graph analysis and transformation. *Automation in Construction*, *155*, 105063. https://doi.org/10.1016/j.autcon.2023.105063

Fernandes, D., Garg, S., Nikkel, M., & Guven, G. (2024). A GPT-Powered Assistant for Real-Time Interaction with Building Information Models. *Buildings*, *14*(8), 2499. https://doi.org/10.3390/buildings14082499

Ibrahim, N., Aboulela, S., Ibrahim, A., & Kashef, R. (2024). A survey on augmenting knowledge graphs (KGs) with large language models (LLMs): Models, evaluation metrics, benchmarks, and challenges. *Discover Artificial Intelligence*, *4*(1), 1–28. https://doi.org/10.1007/s44163-024-00175-8

Jeon, K., & Lee, G. (2025). Hybrid large language model approach for prompt and sensitive defect management: A comparative analysis of hybrid, non-hybrid, and GraphRAG approaches. *Advanced Engineering Informatics*, *64*, 103076. https://doi.org/10.1016/j.aei.2024.103076

Kau, A., He, X., Nambissan, A., Astudillo, A., Yin, H., & Aryani, A. (2024, July 9). *Combining Knowledge Graphs and Large Language Models*. http://arxiv.org/pdf/2407.06564

LangChain. (2025a). *Introduction | 🦜🔗 LangChain*. https://python.langchain.com/docs/introduction/

LangChain. (2025b). *Neo4j | 🦜🔗 LangChain*. https://python.langchain.com/docs/integrations/providers/neo4j/#graphcypherqachain

Liu, Y [Yinhan], Ott, M., Goyal, N., Du Jingfei, Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., & Stoyanov, V. (2019, July 26). *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. http://arxiv.org/pdf/1907.11692

Mallick, S. B., & Korevec, K. (2024, November 12). The next chapter of the Gemini era for developers. https://developers.googleblog.com/en/the-next-chapter-of-the-gemini-era-for-developers/

Needham, M., & Hodler, A. E. (2019). *Graph algorithms: Practical examples in Apache Spark and Neo4j* (First edition). O'Reilly Media.

Ollama. (2025a). *llama3.2:3b*. https://ollama.com/library/llama3.2:3b

Ollama. (2025b). *mxbai-embed-large*. https://ollama.com/library/mxbai-embed-large

OpenAI. (2025, January 14). *GPT-4o*. https://openai.com/index/hello-gpt-4o/

Pan, S. (2024, May 10). *Unifying Large Language Models and Knowledge Graphs: A Roadmap | IEEE Journals & Magazine | IEEE Xplore*.

Pauwels, P., Costin, A., & Rasmussen, M. H. (2022). Knowledge Graphs and Linked Data for the Built Environment. *Industry 4.0 for the Built Environment*, *20*, 157–183. https://doi.org/10.1007/978-3-030-82430-3_7

Peng, B., Zhu, Y., Liu, Y [Yongchao], Bo, X., Shi, H., Hong, C., Zhang, Y., & Tang, S. (2024, August 15). *Graph Retrieval-Augmented Generation: A Survey*. http://arxiv.org/pdf/2408.08921

Pfitzner, F., Braun, A., & Borrmann, A [André] (2024). From data to knowledge: Construction process analysis through continuous image capturing, object detection, and knowledge graph creation. *Automation in Construction*, *164*, 105451. https://doi.org/10.1016/j.autcon.2024.105451

Pusch, L., & Conrad, T. O. F. (2024, September 6). *Combining LLMs and Knowledge Graphs to Reduce Hallucinations in Question Answering*. http://arxiv.org/pdf/2409.04181

Raffel, C., Shazeer, N., Roberts, A., Lee, K [Katherine], Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2019, October 23). *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. http://arxiv.org/pdf/1910.10683

Saka, A., Taiwo, R., Saka, N., Salami, B. A., Ajayi, S., Akande, K., & Kazemi, H. (2024). GPT models in construction industry: Opportunities, limitations, and a use case validation. *Developments in the Built Environment*, *17*, 100300. https://doi.org/10.1016/j.dibe.2023.100300

Sun, C., Han, J., Deng, W., Wang, X., Qin, Z., & Gould, S. (2023, October 19). *3D-GPT: Procedural 3D Modeling with Large Language Models*. http://arxiv.org/pdf/2310.12945

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., & Lample, G. (2023, February 27). *LLaMA: Open and Efficient Foundation Language Models*. http://arxiv.org/pdf/2302.13971

Trajanoska, M., Stojanov, R., & Trajanov, D. (2023, May 8). *Enhancing Knowledge Graph Construction Using Large Language Models*. http://arxiv.org/pdf/2305.04676

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017, June 12). *Attention Is All You Need*. http://arxiv.org/pdf/1706.03762

W3C. (2025a, January 20). *SPARQL 1.2 Query Language*. https://www.w3.org/TR/sparql12-query/

W3C. (2025b, January 22). *RDF 1.2 Concepts and Abstract Syntax*. https://www.w3.org/TR/rdf12-concepts/

W3Schools. (2025a, January 23). *What is JSON*. https://www.w3schools.com/whatis/whatis_json.asp

W3Schools. (2025b, January 23). *XML Introduction*. https://www.w3schools.com/XML/xml_whatis.asp

Wang, Z [Zijian], Ying, H., Sacks, R., & Borrmann, A [André] (2023). CBIM: A Graph-based Approach to Enhance Interoperability Using Semantic Enrichment. In

Proc. of the 30th Int. Conference on Intelligent Computing in Engineering (EG-ICE). https://mediatum.ub.tum.de/node?id=1719836

Xu, Z., Jain, S., & Kankanhalli, M. (2024, January 22). *Hallucination is Inevitable: An Innate Limitation of Large Language Models*. http://arxiv.org/pdf/2401.11817

Yao, L., Peng, J., Mao, C., & Luo, Y. (2023, August 26). *Exploring Large Language Models for Knowledge Graph Completion*. http://arxiv.org/pdf/2308.13916

Zeng, A., Liu, X., Du Zhengxiao, Wang, Z [Zihan], Lai, H., Ding, M., Yang, Z., Xu, Y., Zheng, W., Xia, X., Tam, W. L., Ma, Z., Xue, Y., Zhai, J., Chen, W., Zhang, P., Dong, Y., & Tang, J. (2022, October 5). *GLM-130B: An Open Bilingual Pre-trained Model*. http://arxiv.org/pdf/2210.02414

Zheng, J., & Fischer, M. (2023, April 19). *BIM-GPT: a Prompt-Based Virtual Assistant Framework for BIM Information Retrieval*. http://arxiv.org/pdf/2304.09333

Zhu, J., Wu, P., & Lei, X. (2023). IFC-graph for facilitating building information access and query. *Automation in Construction*, *148*, 104778. https://doi.org/10.1016/j.autcon.2023.104778

Appendix

Responses from the graph-based RAG Agent (provided as XLS)

Code implementation (provided as ZIP-folder)