

Master Thesis

Conversational Access to Cultural Heritage: Evaluating RAG on the Vienna History Wiki

Caitlynn Hans Setiabudi

Date of Birth: 16.04.2003

Student ID: h12422282

Subject Area: Information Business

Supervisor: Daniil Dobriy and Prof. Dr. Axel Polleres

Date of Submission: 22.09.2026

*Department of Information Systems & Operations Management, Vienna
University of Economics and Business, Welthandelsplatz 1, 1020 Vienna,
Austria*

Acknowledgements

First and foremost, I would like to thank my supervisors, Daniil Dobriy and Prof. Dr. Axel Polleres. Thank you for your patience, guidance, and support that you have given me throughout the writing of this thesis. I would also like to say thank you for always being there and making time for me, especially during my challenging times. Your guidance and feedback have not only helped me complete this thesis but have also helped me grow and develop in my career. For that, I am deeply grateful.

I would also like to thank Christian Erlinger for giving me an overview of the Vienna History Wiki and guidance before the expert interviews took place. Your input has helped me immensely in preparing the questions and understanding their context. My thanks also go to Katrin Kühnert and Christiane Fritze for taking the time to share their expertise and views on the Vienna History Wiki. Your insights directly influenced the use case and design decisions in this thesis.

I would also like to thank Bernhard Krabina, whose work on the knowledge graph of the history of Vienna laid much of the foundation on which this thesis builds.

To my parents, Jie Rita Handayani and Gunawan Setiabudi, thank you for your unconditional love, support, and belief in me. None of this would have been possible without you.

A special thank you to Julius Serbenta for lending me his laptop when I needed it most, and to Ana, Azina, Jan, and Huda for always supporting me and keeping me going along the way. Finally, thank you to Ivi for the constant companionship and comfort throughout this journey.

Contents

1	Introduction	11
1.1	Background and Motivation	11
1.2	Problem Statement	12
1.3	Research Objectives and Hypotheses	12
1.4	Research Questions	14
1.5	Contributions	14
1.6	Thesis Structure	15
2	Theoretical Background	16
2.1	Retrieval-Augmented Generation	16
2.2	Vector-Based Retrieval	17
2.2.1	Text Embedding	17
2.2.2	Similarity Search	18
2.2.3	Exact and Approximate Nearest Neighbour Search	20
2.2.4	Vector Store	21
2.3	Knowledge Graphs and Graph Traversal	22
2.3.1	RDF Representation and Verbalisation	22
2.3.2	Triple Stores and SPARQL	23
2.3.3	Graph Neighbourhood Expansion	23
2.4	Agentic Retrieval	24
2.5	Evaluation of Retrieval-Augmented Generation	25
2.5.1	Lexical Matching	25
2.5.2	LLM-Based Metrics	25
2.5.3	Claim-Level Evaluation	27
3	Related Work	28
3.1	Vector RAG Approaches	28
3.2	Graph RAG Approaches	29
3.3	Agentic Graph Reasoning Approaches	31
4	Use Case	32
4.1	Vienna History Wiki	33
4.2	Consultation with Domain Experts	33
4.2.1	Accuracy and Factual Grounding	34
4.2.2	Question Type Complexity	34
4.2.3	Sensitive Topics Handling	34
4.2.4	Data Completeness and Ontology Coverage	35
4.3	Implications for System Design	35

5	Research Methodology	35
5.1	Experimental Design	36
5.2	Compared Retrieval Architectures	37
5.3	Experimental Controls and Configuration Selection	38
5.4	Component and Model Selection	39
5.4.1	Embedding Model Selection	39
5.4.2	Retrieval Infrastructure	41
5.4.3	Generation and Judge Models	42
5.4.4	Prompt Requirements	43
5.5	Evaluation	44
5.5.1	Benchmark	44
5.5.2	Evaluation Protocol	46
5.5.3	Retrieval-Based Evaluation	46
5.5.4	RAG Assessment (RAGAs)	47
5.5.5	Manual Evaluation	47
6	Implementation	49
6.1	Document Vector RAG	50
6.1.1	Input Pre-Processing	50
6.1.2	Chunking	52
6.1.3	Embedding and Index Construction	53
6.1.4	End-to-End Pipeline	53
6.2	Knowledge Graph-Derived RAG	56
6.2.1	RDF Pre-Processing	56
6.2.2	QLever Graph Construction	57
6.2.3	RDF Triple Verbalisation	57
6.2.4	Triple Embedding and Qdrant Index Construction	59
6.2.5	Shared Runtime Components	60
6.2.6	End-to-End KG-Derived Vector RAG	63
6.2.7	End-to-End GraphRAG 1-Hop	64
6.2.8	End-to-End GraphRAG 2-Hop	67
6.3	Agentic Graph RAG	70
6.3.1	System Overview	70
6.3.2	Topic Entity Extraction and Linking	71
6.3.3	Relation Exploration	72
6.3.4	Entity Exploration	73
6.3.5	Path Bookkeeping, Reasoning, and Termination	74

7 Results	76
7.1 Retrieval-Based Evaluation Results	77
7.1.1 Results by Question Category	77
7.2 RAG Assessment (RAGAs) Results	79
7.2.1 Effect of the Generation Provider	80
7.2.2 Results by Question Category	82
7.3 Manual Evaluation Results	84
7.3.1 Results by Architecture and Question Type	85
7.3.2 Effect of the Generation Provider	90
8 Discussion	91
9 Conclusion	97
9.1 Limitations	98
9.2 Future Work	99
A Code and Data Availability	105
B Retrieval Configuration Ablation	105
B.1 Grid and Measurement	106
B.2 Selection Rule	106
B.3 Results	107
B.3.1 The Chosen Configuration	108
B.3.2 KG-Derived Vector RAG Configuration and Document	
Vector RAG configuration	109
B.3.3 Observations from the Grid	109
C Evaluation Scoring Details	110
D Pre-Processing and Data Preparation	111
D.1 Document Pre-Processing	111
D.2 Document Chunk Construction	112
D.3 RDF Pre-Processing Algorithm	114
D.4 RDF Triple Verbalisation Algorithm	114
E Retrieval Queries	116
E.1 One-Hop SPARQL Neighbourhood Query	116
E.2 Second-Hop SPARQL Neighbourhood Query	116
F Generation Prompt for KG-Based Architectures	116
G End-to-End Algorithm of KG-Derived Vector RAG	118

List of Figures

1	Example of Simplified RDF Triples	22
2	Overview of the Evaluated System	49
3	Example Qdrant Payload of a Document Chunk	53
4	End-to-End Pipeline of Document Vector RAG	54
5	Prompt Template used by Vector RAG Pipeline	54
6	Algorithm of Document Vector RAG	55
7	Document Vector RAG: Processing of the Example Question	55
8	End-to-End Pipeline of KG-Derived Vector RAG	63
9	KG-Derived Vector RAG: Top Six Retrieved Triples for the Example Question	64
10	End-to-End Pipeline of GraphRAG 1-Hop	65
11	GraphRAG 1-Hop: Processing of the Example Question	67
12	End-to-End Pipeline of GraphRAG 2-hop	67
13	Algorithm of Graph RAG 1 and 2 Hop	69
14	End-to-End Pipeline of Agentic Graph RAG	71
15	Algorithm of the Agentic Graph RAG(q)	75
16	Agentic Graph RAG: Processing of the Example Question	76
17	Document Chunks Pre-Processing	112
18	Document chunk construction	113
19	RDF corpus pre-processing	114
20	VerbaliseFile(input_nt, output_jsonl)	115
21	One-Hop SPARQL Neighbourhood Query Template	116
22	Second-Hop SPARQL Neighbourhood Query Template	116
23	ASK($query$, $top_k = 90$) - Verbalised Triple RAG	118

List of Tables

1	Mapping of Research Questions to Architecture Comparisons	37
2	Comparison of the Evaluated Retrieval Architectures	37
3	Retrieval Configurations of the Evaluated Architectures	38
4	Top 5 Candidate Embedding Models by Score on the Massive Text Embedding Benchmark	40
5	Example Benchmark Question and Required Evidence per Question Category	45
6	Criteria for Classifying Statements as True Positive, False Positive, or False Negative	48
7	Qdrant Payload Fields and Examples for the Triple Collection	60
8	Term Recall per Architecture and Generation Provider	77
9	Term Recall per Architecture and Question Category with Combined LLM Providers	78
10	Faithfulness, Answer Relevancy, Context Recall, and Answer Correctness per Architecture	79
11	Automated RAG Assessment Metrics per Architecture and Generators	81
12	Answer Correctness by Architecture, Category, and Provider (O = OpenAI, Q = Qwen)	82
13	Statement-Level Outcomes by Architecture and Question Category	84
14	Precision, Recall, and F1 per Architecture and Question Category Under Each Generation Provider	86
15	Ablation Grid of Retrieval Configurations Ordered by Context Recall	108
16	Decomposition Rules for Atomic Gold Claim	111
17	Differences Between the Original Think-on-Graph and this Thesis Adaptation	119

Abstract

Large Language Models (LLMs) offer a promising way to make large cultural heritage collections accessible through natural-language conversation. However, historical information needs to be presented accurately, transparently, and without unsupported speculation, since wrong or misleading answers can change readers’ understanding of people, events, places, and their relationships. Retrieval-Augmented Generation (RAG) supports this need by grounding generated answers in evidence retrieved from an external knowledge source.

This thesis compares five RAG architectures for question answering over the Vienna History Wiki, a cultural heritage knowledge base built on Semantic MediaWiki. The evaluated architectures consist of Document Vector RAG over unstructured wiki content, vector retrieval over verbalised RDF triples, Graph RAG using one hop and two hop expansion, and an agentic system that selects graph traversal steps dynamically.

The systems are evaluated on a 120-question benchmark across four categories, namely, single value, multi hop, aggregation, and description, which are designed to test the retrieval and reasoning capabilities of different complexities. The first evaluation is through manual statement-level classification of true positives, false positives, and false negatives, from which precision, recall, and F1 are calculated. Second, retrieval quality is assessed through automated term-recall analysis independently of answer generation. Third, RAG-specific answer quality is evaluated using the RAG Assessment (RAGAs) framework, covering faithfulness, answer relevance, context recall, and answer correctness, as a complementary automated layer.

The results show that the most suitable degree of graph integration depends on the relational structure of the question. Single-value questions are best answered by verbalised-triple architecture. For aggregation questions, graph-based retrieval outperforms vector-based retrieval. One-hop Graph RAG is the strongest static architecture when related facts are concentrated around one entity, while the agentic system achieves the highest performance. For multi-hop questions, the agentic system also performs strongest. For description questions, the agentic system achieves the highest performance, followed closely by Document Vector RAG, whose broader chunks can provide multiple relevant attributes together. Overall, the graph integration becomes more useful as relational complexity increases.

1 Introduction

This chapter introduces the context and motivation for investigating the use of Retrieval-Augmented Generation (RAG) to support conversational access to the Vienna History Wiki. Firstly, this chapter will explain the opportunities and limitations of Large Language Models in answering factual questions in cultural heritage domains. Afterwards, this chapter presents the research purpose, hypotheses, and research questions that guide the empirical comparison of different retrieval architectures. Finally, the chapter summarises the main contributions of the study and provides an overview of the thesis structure.

1.1 Background and Motivation

Large Language Models (LLMs) have led to notable improvements in the capabilities of conversational systems and natural language interfaces, producing fluent and contextually appropriate responses to user queries. However, LLMs are also known to exhibit significant limitations. They hallucinate facts not present in their training data and cannot reliably access knowledge that post-dates their training cutoff [26]. These issues have become the main challenge in domains where factual correctness is the most important requirement.

Retrieval-Augmented Generation (RAG) has emerged as an approach to address these limitations. Instead of relying only on the model’s parametric knowledge, RAG systems retrieve relevant information from an external knowledge source at query time and use it as context for the language model. This grounds the generated response in verifiable external evidence. Most existing RAG systems use vector-based retrieval over unstructured text collections, using embedding similarity to find relevant documents or passages.

Many cultural heritage platforms represent their knowledge as structured and interconnected semantic graphs. The Vienna History Wiki is one example. This wiki is a Semantic MediaWiki-based knowledge base that models knowledge about Vienna’s history through semantic links between people, locations, buildings, and institutions [16]. The wiki’s content is exportable as RDF and queryable via SPARQL, making it a suitable basis for studying graph-based retrieval approaches to conversational access. Its domain is factually rich, multilingual, and structured enough to support systematic comparisons of retrieval strategies across question types of varying complexity.

1.2 Problem Statement

Vienna History Wiki contains rich, structured historical knowledge. However, complex questions are difficult to access for non-technical users because they may require SPARQL queries and knowledge of the underlying schema. This limits direct access to information that depends on combining facts, following relationships, or aggregating data. Therefore, natural language access requires a retrieval approach that can interpret user questions and retrieve the relevant evidence from the wiki.

RAG offers one possible solution, but retrieval architectures differ in how they access the underlying knowledge. Document-based Vector RAG retrieves unstructured texts through embedding similarity, while the knowledge-graph-based approach retrieves structured facts and follows the explicit relationships between entities [26]. This difference is important, especially for complex queries such as relational queries and aggregation queries. Although interest in Graph RAG is growing, it is still unclear when the use of graph structures provides a benefit to justify the additional complexity [1]. Therefore, this thesis compares five retrieval architectures, namely, document-based vector retrieval, verbalised-triples vector retrieval, one-hop and two-hop graph expansion, and agentic graph traversal. This comparison evaluates when increasing graph integration improves retrieval and answer quality to justify the additional complexity.

1.3 Research Objectives and Hypotheses

The primary objective of this thesis is to investigate how different degrees of graph integration affect the quality of natural-language question answering over the Vienna History Wiki, and to determine which retrieval strategies are most suitable for different types of information needs. To address this objective, five Retrieval-Augmented Generation (RAG) architectures are empirically compared, ranging from vector retrieval over unstructured text and verbalised RDF triples to fixed graph expansion and agentic graph traversal.

Previous studies suggest that the effectiveness of vector-based and graph-based retrieval depends on the type of information required by the question. Conventional RAG performs well on direct and fact-oriented questions, whereas graph-based retrieval provides greater benefits when answering questions that require relational or multi-step reasoning [11]. Graph-based retrieval connects entities explicitly and could retrieve connected evidence through the graph structure. This makes it suitable for questions that need to combine different facts or explore connections between those facts [26]. On the other

hand, document-based retrieval provides a broader information context in each retrieval unit, which benefits direct and descriptive questions when the information needed is already available in the retrieved text. Therefore, according to these complementary characteristics, the benefit expected from graph integration depends on the question category.

Hypothesis 1. Graph RAG is expected to achieve a higher claim-level F1 than vector-based RAG for multi-hop and aggregation questions, but a lower claim-level F1 for single-value and description questions.

The retrieval granularity also affects the types and the amount of facts that are presented to the large language models (LLMs) to generate the answer. A bigger retrieval unit keeps more of its surrounding context while also including information that is redundant and irrelevant. On the other hand, a smaller retrieval unit provides more focused evidence, with the risk that it may not cover all the information needed to provide a complete answer [10]. Furthermore, previous works also show that reducing irrelevant retrieval context causes the information given to the model to be more focused, even though extreme filtration can also remove useful evidence [35]. These findings show that there is a trade-off between broader information coverage from document chunks and the more focused evidence provided by individual RDF-derived facts.

Hypothesis 2. Retrieval over verbalised RDF triples is expected to result in higher claim-level precision and lower claim-level recall than retrieval over document chunks.

The agentic retrieval differs from fixed neighbourhood expansion because the traversal decision can be constantly adjusted to the question and the evidence found in the previous retrieval stage. Think-on-Graph performs iterative relation and entity selection, keeping an explicit reasoning path and continuously evaluating whether the gathered evidence is enough to answer the question [32]. The Graph RAG research also differentiates adaptive retrieval from non-adaptive retrieval and highlights its potential for questions requiring connected evidence across multiple reasoning steps [26]. Therefore, selective traversal is expected to provide an advantage over fixed graph expansion, especially when the required evidence depends on intermediate entities or relationships.

Hypothesis 3. Graph RAG enhanced with an agentic LLM layer is expected to achieve a higher overall claim-level F1 than Graph RAG with fixed traversal, especially for multi-hop and aggregation questions.

Lastly, the answer quality of RAG systems also depends on the language model used to generate the answer, since models differ in how effectively they use retrieved evidence and in their sensitivity to irrelevant context

[30]. Therefore, this thesis compares two different models, namely GPT-5.4 and Qwen 3.5 which is the hosted version of open-weight model Qwen3.5-397B-A17B. Previous work shows that the choice of generator model can significantly affect RAG performance, even though it has not yet directly compared GPT-5.4 and Qwen 3.5 [30]. Given the stronger general capabilities expected from GPT-5.4 on public benchmarks, it is hypothesised to achieve higher answer quality than Qwen 3.5 in the evaluated RAG settings.

Hypothesis 4. LLM GPT-5.4 is expected to achieve higher overall answer quality than Qwen 3.5 Plus across the evaluated retrieval architectures and question categories, as measured by claim-level precision, recall, and F1.

1.4 Research Questions

This thesis addresses the following main research question: "To what extent does increasing graph integration, from vector retrieval over unstructured and structured data, through fixed one-hop and two-hop SPARQL neighbourhood expansion, to agentic graph traversal, improve answer quality for queries over the Vienna History Wiki?"

This is decomposed into the following sub-questions:

- Research Question 1: Does the relative claim-level F1 of Vector and Graph RAG depend on question category, and in which direction?
- Research Question 2: How does the granularity of the retrieval unit, document chunks versus verbalised RDF triples, affect the balance between claim-level precision and recall?
- Research Question 3: Does adding an agentic LLM layer to Graph RAG improve overall claim-level F1 compared with Graph RAG using fixed traversal?
- Research Question 4: How does the choice of generation model affect claim-level precision, recall, and F1 across the evaluated retrieval architectures?

1.5 Contributions

This thesis makes several contributions to conversational access over structured cultural heritage knowledge. First, this thesis provides a reproducible end-to-end pipeline for transforming a cultural heritage wiki represented through Semantic MediaWiki into a form suitable for retrieval-augmented generation. The pipeline supports retrieval over both unstructured document content

and structured RDF representation, which can be adapted to other cultural heritage wikis with similar data structures.

Second, this thesis provides a benchmark that is manually constructed, consisting of 120 question and answer pairs across four categories, namely single-value, description, aggregation, and multi-hop. This benchmark was developed based on the Vienna History Wiki and cross-checked with domain experts to ensure that the questions reflect the relevant information needs and that the required answers are correct. This benchmark provides a structured basis for evaluating different retrieval architectures on various types of questions in the cultural heritage context.

Third, this thesis systematically implements and evaluates five retrieval architectures that represent different levels of graph integration. These architectures include the conventional Document Vector RAG, vector retrieval on verbalised RDF triples, one hop and two hop Graph RAG, and Graph RAG with an agentic LLM layer. All of the architectures are evaluated under a common experimental setting using term recall, automated RAG Assessment (RAGAs), and manual evaluation. This allows the effects of retrieval representation, graph expansion, and agentic traversal to be compared while distinguishing retrieval quality from final-answer correctness. By highlighting the strengths and limitations of each approach, these findings can support future implementation decisions and help the Vienna History Wiki select or combine retrieval strategies according to their information-access needs.

Finally, this thesis develops an agentic system for the Vienna History Wiki that performs iterative traversal over its RDF structure. The system adapts agentic graph reasoning to the heterogeneous Semantic MediaWiki representation through entity linking, selective graph exploration, path preservation, and sufficiency checking. This broadens the range of retrieval architectures available for conversational access to the Vienna History Wiki beyond standard vector-based approaches.

1.6 Thesis Structure

The remainder of this thesis is organised as follows. Section 2 introduces the technical concepts required throughout the thesis, including text embeddings, similarity search, approximate nearest-neighbour indexing, and vector stores. Section 3 reviews related work on Vector RAG, Graph RAG, and agentic graph retrieval. Section 4 presents the Vienna History Wiki use case and summarises the findings from the domain-expert consultation that informed the system requirements.

Section 5 describes the research methodology, including the compared retrieval architectures, components, model selection, benchmark design, and

evaluation framework. Section 6 explains the implementation of the Document-based Vector RAG, KG-derived and static Graph RAG approaches, and the agentic graph-traversal system. Section 7 presents the experimental results across retrieval quality, automated RAG evaluation, and manual evaluation. Section 8 discusses the findings in relation to the research questions, hypotheses, and cultural heritage use case. Finally, Section 9 presents the conclusions, limitations, and directions for future work. Additional retrieval-configuration experiments and implementation details are provided in the Appendix.

2 Theoretical Background

This section introduces technical concepts that are needed to understand retrieval architectures that are evaluated in this thesis. First, it explains the general RAG pipeline, followed by the main concepts underlying vector-based retrieval, knowledge graphs and graph traversal, and agentic graph retrieval.

2.1 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) is an approach that combines external information retrieval with language model generation. The purpose is to provide relevant information from the external knowledge source to the language model instead of only relying on information stored in the model’s parameters. As a result, the answer produced can be based on the retrieved evidence while allowing the system to access domain-specific or updated information [26].

The RAG pipeline generally consists of an offline preparation stage and a query processing stage. In the preparation stage, the source information is processed into retrievable units, such as document chunks or facts from a knowledge graph, and stored in a searchable index. When a query is submitted to the language model, the user query is used to retrieve relevant units from that index. Then, the evidence found is combined with the original question and given to the language model for final answer generation.

RAG architectures can generally be divided into vector-based and graph-based approaches. Vector RAG represents questions and retrieval units as embeddings and retrieves information based on semantic similarity alone. On the other hand, Graph RAG uses the explicit relations between entities to retrieve connected evidence from a knowledge graph. Agentic Graph RAG expands the Graph RAG by adding an agentic LLM layer that iteratively selects relations or entities that need to be explored while also deciding when the evidence is sufficient. Across these approaches, the retrieval unit

and available context capacity affect the balance between focused evidence, broader information coverage, and irrelevant context.

2.2 Vector-Based Retrieval

Vector-Based retrieval identifies information based on meaning, not only on direct word similarity. Both query and retrievable units are represented as numerical vectors in the same embedding space. The retrieval process is done by comparing the query vector with the stored vectors using a similarity or distance measure. Therefore, this approach is based on four components covered in the subsections below. The text embedding subsection explains how the embedding model represents words, sentences, or documents as vectors. The similarity search subsection introduces the similarity measure that is used to rank candidates against the query. The neighbour search subsection explains the exact and approximate nearest-neighbour search which keeps this comparison feasible as the collection grows. Finally, the vector store subsection describes the vector store where the vectors and their metadata are stored and queried. All of these components determine which units are returned as the retrieved context, which will be used by the generator to answer the question.

2.2.1 Text Embedding

An embedding represents an information unit as a dense numerical vector in a continuous vector space. In text retrieval, the objective is to place semantically related texts close to each other so that similarity in the vector space can be used as an approximation of semantic relevance[21]. Compared with traditional representations such as one-hot encoding that represents words as sparse vectors with a single one and zeros that are unable to capture semantic relationships, neural embedding models learn to project text into a low-dimensional space where semantic similarity can be measured mathematically. Therefore, text embeddings become the foundation for similarity-based retrieval in many RAG systems.

According to the granularity level, text embedding is separated into word embeddings and Sentence or Document Embeddings. Word embeddings represent individual words as dense vectors in a continuous vector space, where semantically similar words are placed close to each other. Early neural methods such as Word2Vec and GloVe represent each word using a fixed-length vector. Both are based on the distributional hypothesis, which assumes that words appearing in similar contexts have similar meanings. Nevertheless, they learn these representations differently. Word2Vec is a prediction-based model

that learns from the local context of words, while GloVe is a count-based model that uses corpus-wide word co-occurrence statistics [2]. However, these methods only obtain a single global representation for each word, which makes them less suited than context-dependent representations for gathering sequence-level semantics [19]. Arora et al. confirm this empirically by reporting that non-contextual embeddings perform poorly at disambiguating words that are used in several different ways [3].

Sentence and document embeddings represent entire sentences or documents as single vectors that capture their overall meaning. One example is Doc2Vec which modifies Word2Vec to produce vector representations for full documents. Other examples includes the Universal Sentence Encoder (USE) which generates fixed-length vectors that are suitable for semantic similarity tasks, and Sentence-BERT (SBERT) which adapts the BERT architecture with a siamese network to generate sentence embeddings for efficient semantic search and similarity. These models are useful for RAG pipelines because queries and retrieved contexts can be represented in the same vector space. Modern sentence embedders are usually trained using objectives such as contrastive loss over large collections of text pairs. The multilingual E5 models, for example, uses contrastive pre-training with an InfoNCE loss and in-batch negatives, followed by supervised fine-tuning [34].

Documents are commonly divided into smaller chunks before embedding, since larger chunks preserve more context, while smaller ones keep the retrieved passage focused. Therefore, the document is split into shorter chunks before it gets embedded. One of the most important hyperparameters in RAG pipeline is the size of the chunk [10]. Chunks that are too short do not contain enough context, while chunks that are too long introduce a compression problem instead. One paper divided Wikipedia articles into disjoint 100-word chunks in the original RAG implementation [17]. Another configuration overlaps consecutive chunks to preserve information that spans their boundaries, using 1,000-token chunks with a 100-token overlap [35] or 1,000-character chunks with a 200-character overlap [8]. Furthermore, Gao et al. note that there is no single optimal setting for every RAG system. The choice of chunk size involves a trade-off between preserving sufficient context and reducing irrelevant information. Therefore, it should be adapted to the retrieval task [10].

2.2.2 Similarity Search

After text is represented as embeddings, the similarity between embeddings needs to be calculated to determine which information is most relevant to the query. Vector similarity search compares the query embedding with stored

embeddings using a specific similarity measure and ranks the candidates based on their similarity scores [8]. The chosen similarity measure decides how the similarity is computed and can affect retrieval accuracy and efficiency. This subsection introduces the similarity measures relevant to the retrieval methods used in this thesis.

Cosine Similarity This measures the directional alignment between two vectors and is widely used in semantic retrieval and RAG systems [35]. Cosine similarity between two vectors $\mathbf{x}$ and $\mathbf{y}$ is defined as the normalised dot product:

$$\text{sim}_{\cos}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} = \frac{\sum_{i=1}^d x_i y_i}{\sqrt{\sum_{i=1}^d x_i^2} \sqrt{\sum_{i=1}^d y_i^2}} \quad (1)$$

Here, $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ are the two embedding vectors being compared (e.g., a query embedding and a document embedding), and d is the dimensionality of the embedding space, for example, the number of numerical components in each vector. The scalars x_i and y_i show the i -th components of $\mathbf{x}$ and $\mathbf{y}$ respectively. The numerator $\sum_{i=1}^d x_i y_i$ is the dot product of the two vectors, measuring the extent to which they point in the same direction. The denominator terms $\|\mathbf{x}\| = \sqrt{\sum_{i=1}^d x_i^2}$ and $\|\mathbf{y}\| = \sqrt{\sum_{i=1}^d y_i^2}$ are the Euclidean (L2) norms of $\mathbf{x}$ and $\mathbf{y}$, representing their respective magnitudes. This measures the angle between the two vectors, reaching 1 when they point in exactly the same direction and -1 when they point in opposite directions [8]. Since the measure depends only on orientation, two embeddings can be scored as similar regardless of how long the underlying vectors are [8].

Inner Product (Dot Product) This compares two vectors without normalising their magnitudes. For unit-normalised vectors, ranking by inner product is equivalent to ranking by cosine similarity. The inner product (or dot product) of two vectors is [8]:

$$\text{sim.}(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^d x_i y_i \quad (2)$$

Here, $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ are the two vectors being compared, x_i and y_i are their i -th components, and d is the dimensionality of the vector space. The result $\text{sim.}(\mathbf{x}, \mathbf{y}) \in \mathbb{R}$ is an unbounded scalar that increases with both directional alignment and vector magnitude, so a larger value indicates greater similarity or relevance.

Euclidean Distance (L2 Distance) This measures the geometric distance between vectors and assigns greater similarity to vectors with a smaller distance. The Euclidean distance between vectors is the L^2 norm of their difference [8]:

$$d_{L2}(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_2 = \sqrt{\sum_{i=1}^d (x_i - y_i)^2} \quad (3)$$

Here, $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ are the two vectors whose geometric separation is being measured, x_i and y_i are their i -th components, and d is the dimensionality of the vector space. The term $(x_i - y_i)^2$ shows the squared displacement along the i -th axis, and adding these squared differences in all dimensions results in the squared distance. The result $d_{L2}(\mathbf{x}, \mathbf{y}) \geq 0$ is a non-negative scalar measuring absolute geometric separation, where a smaller value indicates more similar vectors.

2.2.3 Exact and Approximate Nearest Neighbour Search

Vector index organise embeddings to support efficient retrieval while also affecting the trade-off between retrieval accuracy, latency, and scalability. Exact k -nearest-neighbour (k NN) search compares a query vector with every vector in the collection and return k most similar results based on the chosen similarity measure. Even though this method guarantees the exact nearest neighbours, the process requires $O(N \cdot D)$ comparisons for N vectors with dimensionality of D , so the computational cost increases with the size of the corpus. Approximate Nearest-neighbour (ANN) search reduces this computational cost by only checking part of the vector space. Instead of guaranteeing the exact nearest neighbours, ANN methods are used to find candidate with high similarity scores more efficiently. This approach covers several methods, including graph-based, hash-based, clustering-based, tree-based, and quantisation-based methods.

The first category consists of graph-based ANN methods. These methods represent vectors as nodes connected to nearby vectors through proximity edges, allowing retrieval to navigate towards increasingly similar candidates without comparing the query with every vector in the collection. Hierarchical Navigable Small World (HNSW) extends this approach by organising vectors into several proximity graph layers [23]. Each layer consists of a subset of nodes that are connected to neighbouring vectors based on certain similarity or distance measure, for example, cosine similarity, or Euclidean distance. During retrieval, the algorithm starts at an entry point in the highest layer and moves greedily to the neighbour that is most similar with the query. This

process continues to the lower level layer to base layer where a detailed local search is performed to determine the final nearest-neighbour candidates.

Second, hash-based ANN methods such as locality-sensitive hashing (LSH) map similar vectors to the same or nearby buckets with high probability. They are good for sparse and high-dimensional collections that are rarely updated, but miss a larger share of the true nearest neighbours [21]. Another category is tree-based method such as KD-tree and ball-tree, which recursively partition the vector space and only visit regions that may contain the nearest neighbours. Their advantage decreases in high dimensions, where the dimensionality problem leaves few regions that can be skipped [21].

Last example is cluster-based, for example, inverted file (IVF). IVF is an ANN method that divides the embedding space using k -means clustering, where centroids are computed by minimising the overall within-cluster variance and each vector is assigned to its nearest centroid [8]. During query processing, IVF only evaluate cluster that has the closest centroid with the query vector, which is called *nprobe* clusters. Adjusting the number of searched clusters provides a controllable trade-off that can be controlled between retrieval latency and recall [8]. A larger *nprobe* value usually increases the retrieval accuracy, while a smaller value increases the search speed.

2.2.4 Vector Store

A vector store provides a storage and retrieval layer for embedding-based search. It stores vector representations with their metadata or related payload, while also providing an interface to retrieve the most similar vector to the query embedding. In the RAG pipeline, the vector store connects the indexing and the retrieval stage. Embeddings that are produced during pre-processing are stored in the database, and at query time, the query embedding is compared with the stored vectors using a similarity measure. This makes it possible for semantic retrieval based on vector similarity and not only exact keyword matching. Vector stores commonly support some additional functionalities such as approximate nearest-neighbour indexing, metadata filtering, and persistence for vector representations. Some examples include Qdrant, Weaviate, Milvus, and Pinecone while libraries such as FAISS provide vector similarity search that is efficient without offering whole database functions like a dedicated vector store.

With the vector store in place, the four components form a complete retrieval path. The corpus is embedded and indexed once in the pre-processing stage. At query time, the same model encodes the question, and the store is queried for the entries most similar to it. Then, the returned entries are combined into a context sent to the generator to produce the final generated

answer.

2.3 Knowledge Graphs and Graph Traversal

Unlike vector-based retrieval that represents information as embeddings and retrieves based on semantic similarity, knowledge-graph-based retrieval represents information as entities and relationships that are explicitly connected. The difference in structure affects the way information is stored and retrieved. Knowledge-graph-based retrieval uses a graph database or triple store to access relations, to find neighbouring entities, and traverse the graph. Consequently, this KG-based retrieval takes not only the relevant facts but also facts that become relevant through relationships with other entities. There are three concepts that are needed to describe this retrieval type. The RDF representation and verbalisation subsection introduces the RDF triple as the unit of representation and the verbalisation step that makes the graph readable by embedding and language models. The triple stores subsection introduces triple stores and SPARQL, through which the graph stays queryable at retrieval time. The last subsection describes neighbourhood expansion, which is how connected evidence is collected by following relations outward from a starting entity.

2.3.1 RDF Representation and Verbalisation

In the Resource Description Framework (RDF), every statement is a triple in the form of

$$(s, p, o), \tag{4}$$

where s is the subject, p is the predicate, and o is the object. The subject identifies the entity being described, the predicate specifies the property or relationship, and the object represents either another entity or a literal value.

For example, a simplified RDF representation may contain the following facts:

```
Karl Seitz | birthPlace | Vienna  
Karl Seitz | occupation | Politician
```

Figure 1: Example of Simplified RDF Triples

When an object identifies other entities, several triples can be connected together to form directed labelled graph. The entity identifier acts as a graph node, while predicate represents the labelled edge. Literal values such as dates, names, or numbers usually act as terminal values and not as entities that can be traversed further. Consequently, object that is in the form of

entity can direct to other facts that are connected, while literals usually only give factual scores without forming a new graph neighbourhood.

Although RDF stores these relationships explicitly, text embedding models and language models work on text instead of on structured triples. Therefore, graph elements usually are verbalised, which means converted into text, before they can be embedded, retrieved, or passed to a language model as context. In its simplest form, a triple is verbalised by changing every URI with labels that are easily readable, then combines the subject, predicate, and object into one line. As an example, the first triple in the Figure [1](#) then becomes *Karl Seitz birthPlace Vienna*.

2.3.2 Triple Stores and SPARQL

A triple store is a database designed for storing and querying data represented using the Resource Description Framework (RDF) efficiently. A triple store keeps the original structure of the graph so that a retrieval system can access the explicit relations, identify neighbouring entities, and traverse relationships. Triple stores are particularly important for Graph RAG architectures because the RDF graph must remain accessible during retrieval. They are usually accessed using SPARQL Protocol. SPARQL supports structured graph patterns for entity retrieval, result filtration, relation connection, aggregation, and graph traversal. Some examples of triple stores include Apache Jena Fuseki, OpenLink Virtuoso, GraphDB, and QLever.

2.3.3 Graph Neighbourhood Expansion

Graph neighbourhood expansion takes information that is connected with one or several selected entities through graph relations. In a knowledge graph, entities are nodes, and labelled relations are the edges between them. The immediate neighbourhood of an entity covers edges that go in or out of it with entities and literal values that are reached through those edges.

A *one-hop* expansion takes information that is connected directly with the chosen entity. The *two-hop* traversal continues from the entities that were found in the first expansion. In general, traversal depth shows the number of graph relations that can be followed from the starting point. For example, a question like “Where was the spouse of person A born” is answered by traversing the relation from person A to their spouse and subsequently traversing a second relation from the spouse to their birthplace. Therefore, the required evidence is not always similar to the original question but becomes a relevant through the intermediate relationship.

A deeper traversal depth increases the region of graph that is reachable

from the initial entities, but it does not guarantee that the retrieved evidence becomes more relevant. Every expansion can add neighbouring facts, and the total of candidates can increase significantly as more entities are visited. It is due to this, graph retrieval produces a trade-off between *breadth*, which is the number of initial or neighbouring candidates considered, and *depth*, the number of relational steps followed.

With the neighbourhood expansion in place, the three components form a second retrieval path. At query time, one or more starting entities are found, and the triple store is queried through SPARQL for the relations and neighbouring entities around them. The retrieved triples are verbalised and assembled into the context passed to the language model for final answer generator.

2.4 Agentic Retrieval

The static retrieval architecture follows a procedure that has been set before the question is processed, including the number of candidates retrieved, traversal depth, and expansion rules. On the other hand, agentic retrieval allows a language model to decide how the retrieval process should proceed based on the question and the evidence that has been found in the previous steps.

In agentic graph retrieval, the retrieval process is performed iteratively as a beam search. Beam search is a best-first search that keeps only the w highest-scoring candidates at each depth, where w is the beam width. Starting from one or several entities, the language model evaluates candidate relations, chooses relevant neighbouring entities, and retrieves additional evidence from the graph. After each iteration, the language model evaluates whether the evidence that has been collected is enough to answer the question. If the evidence is not sufficient, the system performs further graph retrieval. If the evidence is sufficient, traversal stops, and the final answer is generated.

There are several operations that differentiate between agentic retrieval and static graph expansion. *Relation selection* identifies relations that are relevant to the question, while *entity selection* identifies the neighbouring entities should remain under consideration. *Path construction* preserves the order of relations used to find the evidence, while *sufficiency checking* decides whether further traversal is still needed.

Therefore, agentic retrieval can allocate the retrieval process selectively instead of expanding every neighbourhood using the same set of rules. The graph queries themselves are issued by a fixed controller, so the agency lies in the language model acting as the search heuristic and stopping criterion. A procedural loop would replace both with predefined rules. Agentic retrieval is

useful when only a small part of the graph is relevant to the question or when the required evidence must be connected through several relations. However, relation selection, entity selection, and sufficiency checking require additional model calls. This increases the computational cost and makes the retrieval process dependent on the behaviour of the chosen model.

2.5 Evaluation of Retrieval-Augmented Generation

There are multiple ways to evaluate a RAG system. The approaches differ in what they compare, such as the generated answer, the retrieved context, or referenced answer, and in how the comparison is performed, such as string matching, a language model acting as a judge, or manual evaluation. These following approaches are the most relevant for this thesis.

2.5.1 Lexical Matching

This approach is the simplest approach that checks whether the expected terms from a reference answer appear in a text, for example, in the retrieved context or in the generated answer. Since the comparison is based on string matching, it is deterministic, transparent, and inexpensive to compute. Nevertheless, the same fact can be expressed with different wording. As a result, the exact matching may miss correct synonyms, paraphrases, or alternative formats unless these variations are taken into account.

2.5.2 LLM-Based Metrics

Frameworks such as RAG Assessment (RAGAs) use a language model as a judge that decomposes texts into individual claims and decides for each claim whether it is supported by another text [9]. This allows for the evaluation to be done in a large dataset without manual intervention. There are several of these metrics that are commonly used. [27].

Faithfulness measures how well the generated answer is supported by the retrieved evidence. To calculate the score, the judge first decomposes the generated answer into individual factual claims and then checks each claim against the retrieved context. A claim is counted as supported when it can be inferred from that context.

$$\text{Faithfulness} = \frac{|\text{claims in the answer supported by the retrieved context}|}{|\text{claims in the answer}|}, \quad (5)$$

The numerator counts the claims in the generated answer that the judge considers supported by the retrieved context, and the denominator counts all claims identified in the answer. The gold answer is not used. A score of 1 means that all the generated claims are supported by the retrieved contexts.

Context Recall measures whether the retrieved context contains enough information to answer the question. The gold answer is used as a proxy for the required evidence. The gold answer is first broken down into individual claims, and then each claim is checked to determine whether the retrieved context contains information similar to the gold answer.

$$\text{ContextRecall} = \frac{|\text{claims in the reference supported by the retrieved context}|}{|\text{claims in the reference}|}. \quad (6)$$

The numerator counts the claims of the gold answer found in the retrieved context, and the denominator counts all claims in the gold answer. The score ranges from 0 to 1, and a score of 1 means that the retrieved context contains enough information for the generator to answer adequately according to the gold answer.

Answer Correctness compares the generated answer with the gold answer directly and combines statement-level factual correctness with semantic similarity. For the factual component, the judge decomposes the generated and gold answers into individual statements and classifies them as true positives (TP), false positives (FP), or false negatives (FN). A TP is a generated statement supported by the gold answer, an FP is a generated statement not supported by the gold answer, and an FN is a required statement from the gold answer that is missing from the generated response.

$$F_1^{\text{factual}} = \frac{|TP|}{|TP| + \frac{1}{2}(|FP| + |FN|)}, \quad (7)$$

where $|TP|$, $|FP|$, and $|FN|$ are the numbers of true-positive, false-positive, and false-negative statements, respectively. A higher F_1^{factual} indicates greater factual agreement and completeness with respect to the gold answer. The semantic component is the cosine similarity between the embeddings of the generated answer and the gold answer, which gives a high score to semantically equivalent wording. Both components are combined with the default RAGAs weighting:

$$\text{AnswerCorrectness} = 0.75 \cdot F_1^{\text{factual}} + 0.25 \cdot \cos(E(\text{answer}), E(\text{gold})), \quad (8)$$

where F_1^{factual} is the statement-level factual score, $E(\cdot)$ denotes the embedding model used to vectorise the text, answer is the generated response, gold is the gold answer, and $\cos(\cdot, \cdot)$ denotes cosine similarity between their embeddings. The weighting gives greater importance to factual correctness while still accounting for semantic equivalence.

Answer Relevancy measures whether the answer addresses the question that was asked, without looking at the gold answer at all. The judge is asked three independent times to invent the question that the generated answer would be answering and to flag whether the answer is noncommittal. Noncommittal means the answer is vague or evasive in the manner of *I don't know*. Each reconstructed question is embedded, and the score is the mean cosine similarity to the embedding of the original question, turned to zero when every reconstruction flags the answer as noncommittal. The final score is calculated as:

$$\text{AnswerRelevancy} = \frac{1}{3} \sum_{i=1}^3 \cos(E(q), E(\hat{q}_i)) \cdot g, \quad (9)$$

where q is the original question, $\hat{q}_i$ the i -th reconstructed question, and g is zero when all three reconstructions flag the answer as noncommittal and one otherwise. A precise, on-topic answer lets the judge to reconstruct something close to the original question, while an off-topic answer reconstructs a different question.

2.5.3 Claim-Level Evaluation

Instead of assigning one score for the whole answer, the answer and its reference answer can both be decomposed into atomic claims that are compared individually. RAGChecker uses this principle to define the claim-level precision as the proportion of claims in the answer that are supported by the reference, and recall as the proportion of reference claims that appear in the answer [30]. With true positive (TP), false positive (FP), and false negative (FN) counted at the claim level, these use the standard formula for information retrieval which can be seen below [24].

$$P = \frac{TP}{TP + FP}, \quad R = \frac{TP}{TP + FN}, \quad F_1 = \frac{2PR}{P + R}. \quad (10)$$

Precision decreases when an answer contains incorrect or unrequested claims. On the other hand, recall decreases when the required claims are missing with F1 balances both. This separates the incorrect or additional information

from missing information, which a single score cannot show. The claims and their labels can be determined either by a judge model or manually by human evaluators.

3 Related Work

This section reviews previous research on three retrieval approaches that are evaluated in this thesis, namely, Vector-Based RAG, Graph RAG, and Agentic graph retrieval. The discussion focuses on retrieval mechanisms, implementation, empirical findings, and limitations of each approach.

3.1 Vector RAG Approaches

Vector-Based RAG combines a language model with a retriever that identifies relevant text based on embedding similarity. Lewis et al. introduce Retrieval-Augmented Generation as an architecture that combines parametric memory in a sequence-to-sequence model that was previously trained with non-parametric memory that is accessed through dense retrieval [17]. This retrieve-then-generate design is also known as the "Retrieve-Read" framework [10].

The effectiveness of Vector RAG depends on design choices such as chunk size, overlap, and retrieval width [35]. Retrieving too little information could give the model not enough evidence, while retrieving too much information could increase token usage and also input redundant or irrelevant context [35]. Context-compression methods attempt to address this problem by filtering or compressing retrieved passages before generation. For example, COCOM compresses multiple retrieved passages into a small number of context embeddings and reports substantially faster answer generation while preserving competitive answer quality [28].

Vector RAG has also been applied to cultural heritage collections. Kelly et al. present FolkRAG for digital collections of the American Folklife Center at the Library of Congress, whose catalog records, finding aids, and digital objects are not uniformly linked [15]. They replaced FAISS with DeepLake because FAISS does not inherently store metadata with the vectors, even though that metadata is needed for retrieval. HyDE, which embeds an LLM-generated hypothetical answer with the query, increases the retrieval accuracy substantially when compared with traditional pipelines. On the other hand, HyDER, which adds several hypothetical documents and re-ranking, only provides insignificant improvement. On the full dataset, a 250 character chunk is also found to consistently outperform the 1,000 character chunk. According

to the researchers, this is because a bigger chunk size incorporates irrelevant surrounding information, which reduces the semantic precision [15]. Liu et al. address hallucination in ancient Chinese poetry with ACP-RAG. This RAG retrieves from over 12.5 million question-answer pairs by combining semantic retrieval, re-scoring with a trained rank model, keyword retrieval, keyword matching, and LLM-based context filtering [20]. With Qwen1.5-7B as the generator, the accuracy of responses increases from 49.2% without retrieval to 89.0%.

Despite their success, vector RAG systems have notable limitations. First, Vector RAG treats text passages as independent retrieval units. As a result, it may fail to capture the structured relational knowledge that cannot be represented through semantic similarity alone [26]. Previous evaluations show that RAG performs well on single-hop, detail-oriented factual queries but is less effective on multi-hop and reasoning-intensive questions [11]. Second, concatenating retrieved snippets into the prompt produces excessively long contexts, leading to redundancy and the "lost in the middle" problem [26]. These limitations have motivated research on retrieval over explicitly structured knowledge representations.

3.2 Graph RAG Approaches

Graph RAG uses a graph structure to retrieve relational evidence that is represented in the form of nodes, edges, triples, paths, communities, or subgraphs. Peng et al. describe Graph RAG as a three-stages process, namely building a graph rag, retrieving graph elements, and generating answers using these elements [26]. The Graph RAG architecture differs in the graph elements that are retrieved, how the retrieved evidence is represented for the language model, and what the construction of the graph itself requires.

One of the approaches converts natural language questions into a structured query that can be executed directly on the database or knowledge graph. A language model that has gone through pre-training or fine-tuning can produce queries in languages such as SQL, Cypher, GraphQL, or SPARQL [12]. This approach can provide accurate access to the structured information only when the underlying schema is known and the generated query is valid, which requires the model to identify the correct entities and predicates and to match the vocabulary of the target graph. In cultural heritage applications, structured queries are not always produced by the language model. Riali et al. convert object records from the Latin American collections of the Museo delle Culture di Milano (MUDEC) into a CIDOC CRM (Conceptual Reference Model) knowledge graph stored in Neo4j. Cypher queries retrieve an object's origin, dating, and related historical context, which then be turned into a

narrative by an LLM [29]. They found that grounding generation in the ontology lowers the rate of invented content but does not remove the need for expert review since factual errors still appear in the generated narratives [29]. Barla and de Berardinis pair each competency question with a hand-written parameterised SPARQL template, so the query is fixed in advance instead of produced by the model at retrieval time, which rules out invented retrieval steps [4].

Another family keeps embedding-based retrieval but lets graph structure shape what is retrieved. G-Retriever applies embedding similarity to graph elements directly, scoring nodes and edges against the query and extracting a connected subgraph from the highest-scoring ones [13]. GNN-Ret still retrieves text passages, but links them into a graph through shared keywords and document structure and uses a graph neural network to propagate relevance along those links before ranking [18]. These approaches show that graph structure can be used either as the retrieval unit itself or as a signal over conventional text retrieval.

Different Graph RAG approaches focus on broad, corpus-level questions instead of a small number of query-specific facts. One of the Graph RAG architectures that is introduced uses a language model to build the entity knowledge graph from the source document, which then applies a community detection procedure to identify groups of entities that are relevant [7]. The summary for each community is made first. When the query is processed, each community summary can produce partial responses, which are later combined into a final answer. This architecture is suitable for query-focused summarisation and questions that need an overall picture regarding themes that are scattered across a large corpus.

The Graph RAG systems also differ in how the retrieved graph data is presented to the generator. Peng et al. explain that when language models are used as generators, graph data cannot be combined directly with text and must first be converted into a compatible format [26]. They distinguish graph languages from graph embeddings and group graph languages into five types, namely adjacency table or edge table, natural language, code-like forms, syntax trees, and node sequences [26]. In the systems they review, KG-GPT linearises the triples of a retrieved subgraph and concatenates them as input, while other approaches fill a predefined natural-language template for each edge type or use a language model to rewrite an edge table into a natural-language description [26].

Graph retrieval and conventional semantic retrieval can also be combined because both of them provide evidence that is complementary. Text retrieval provides broader contextual coverage, while graph retrieval makes the relationships between entities explicit. In the evaluation using the O-RAN

specification, graph-based configuration increased factual correctness and context relevance compared with conventional Vector RAG [1]. Han et al. also found that conventional RAG is more suitable for direct questions and single-hop scenarios, while Graph RAG gives bigger benefit for relational and reasoning-intensive questions [11]. The same study also found that routing between different retrieval approaches or combining evidence from those approaches can increase the performance, even though this strategy creates a trade-off between retrieval effectiveness and computational efficiency [11].

A similar trade-off is also found in cultural heritage storytelling. Barla and de Berardinis compare three retrieval strategies on a knowledge graph of the 1985 Live Aid concert. KG-RAG retrieves only the triples that answer a SPARQL template, Hybrid-RAG adds curated text chunks to these triples, and Graph-RAG explores the neighbourhood around the seed entities, usually up to one or two hops [4]. In most of the configurations, KG-RAG achieved the highest support, which is a sentence-level measure of grounding in the retrieved evidence. On the other hand, Hybrid-RAG achieved the highest evidence coverage. Lastly, Graph-RAG achieved the highest global cohesion but substantially lower support, which the writers consider to be an artefact of triple-matching metrics. They conclude that the retrieval strategy should be chosen according to the requirements of the application instead of treating one method as the best [4]. These findings show that graph retrieval benefits depend on the information needed by the questions.

Graph RAG also requires additional construction and operational costs. Building a graph from unstructured documents may require the extraction and resolution of entities, relation extraction, graph storage, and indexing. Its effectiveness also depends on graph quality because an entity or relation that is missing can limit the available evidence, while a graph element that is wrong or duplicated can add noise [11]. Approaches that are based on queries, like LogicRAG, try to reduce this limitation by building an adaptive reasoning structure and removing redundant information that is retrieved [5]. Across these approaches, the effectiveness of Graph RAG depends on the type of questions, graph quality, retrieval strategy, and the representation through which graph evidence is supplied to the language model.

3.3 Agentic Graph Reasoning Approaches

Previous research describes two ways in which LLMs can be integrated with knowledge graphs [32]. In the loose-coupling "LLM $\oplus$ KG" concept, exemplified by knowledge base question answering via LLM-generated SPARQL queries, the LLM acts only as a translator that converts the question into a machine executable query but does not participate in the graph reasoning

itself. Because of this, its success depends heavily on the completeness and quality of the KG, and a single missing relation can cause retrieval to fail [32]. In the tight coupling "LLM $\otimes$ KG" concept, the model and the graph work together throughout the reasoning process. When a relation is missing from the graph, the LLM can fill the gap by exploring related triples and concluding based on what it already knows [32].

Think-on-Graph (TOG) implements this concept through an LLM-guided search over the knowledge graph. Starting from the topic entities of the question, the LLM repeatedly explores and prunes the adjacent relations and entities up to N reasoning paths, and then judges whether these paths are sufficient to answer the question or the maximum depth is reached [32]. This paper also summarises the resulting advantages in three points [32]. First, ToG supports deep reasoning by extracting diverse multi-hop reasoning paths from the knowledge graph and using them as the basis for the LLM's reasoning process. Second, it also supports responsible reasoning because the explicit and editable reasoning paths make the process more explainable and allow the provenance of model outputs to be traced and corrected. Third, ToG offers flexibility and efficiency as a training-free, plug-and-play framework that can be applied across different LLMs and knowledge-graphs, while knowledge can be updated directly in the graph instead of costly retraining of the LLM [32].

A related agentic approach is Graph Chain-of-Thought, which performs graph reasoning iteratively through three stages, namely, language-model reasoning, language-model-graph interaction, and execution of the requested graph operation. The resulting information is returned to the model, and the process continues until an answer is reached [14].

Think-on-Graph 2.0 (ToG 2) extends this graph traversal with document retrieval. In other words, now the graph entities link to the documents that mention them, while the retrieved passages provide entity context that guides the next graph step, until the LLM has collected enough evidence to answer. Similar to the first version, ToG 2 requires no additional training, and with GPT-3.5 it achieves state-of-the-art results on six of seven knowledge-intensive datasets [22].

4 Use Case

This section presents the Vienna History Wiki as a use case for evaluating the retrieval architectures developed within this thesis. This section first describes the characteristics of the platform, the historical and technical characteristics, and how structured and unstructured knowledge are represented through Semantic MediaWiki. In addition, this section summarises the results obtained

from the domain-expert consultation and describes how these results were utilised to define the requirements and design considerations for the RAG systems evaluated within this thesis.

4.1 Vienna History Wiki

The Vienna History Wiki (*Wien Geschichte Wiki*) is a digital cultural heritage platform that is managed by the City of Vienna. This platform originated from the digitalisation and continued development of Felix Czeike’s six-volume *Historisches Lexikon Wien* and was made available to the public in 2014. Instead of working just as a digital copy of the encyclopedia, this platform allows historical information to be updated and developed. While traditional wikis allow any member of the public to edit them freely, the Vienna History Wiki is supported by an editorial team consisting of various departments of the City of Vienna and partner institutions. Contributions from the public are reviewed before being accepted, while participating institutions can actively add articles, images, and revisions.

The platform is built using Semantic MediaWiki and combines unstructured encyclopedic articles with structured semantic properties [16]. It contains information about a wide range of historical entities, including people, places, buildings, organisations, events, and monuments. The articles provide descriptive text, while semantic properties represent attributes and relationships between entities explicitly. The structured content can be exported in RDF format, which means that the same source of knowledge can be represented as a knowledge graph as well as a collection of articles. The platform also provides an RDF dump that can be reused for research. This combination allows for document-based retrieval and knowledge-graph-based retrieval to be evaluated using the same knowledge source.

4.2 Consultation with Domain Experts

Consultation with domain experts took place in order to determine the actual needs and expectations concerning access to the Vienna History Wiki. This consultation addressed how the system is supposed to work, types of questions that are the hardest to answer through the current interface, and some further requirements. These findings from the consultation are grouped into several categories, including accuracy and factual grounding, question type complexity, transparency of retrieved information, and the importance of objectivity.

4.2.1 Accuracy and Factual Grounding

The domain experts emphasised that the Vienna History Wiki is considered a scientific source. Therefore, any AI-generated response must be grounded in verifiable content. One participant noted, "to give correct information 100% basically is like the main goal," which shows a requirement for a system that does not hallucinate or speculate when data is incomplete.

4.2.2 Question Type Complexity

The consultation identified several types of questions that users most often need answered but are difficult to obtain through the current Vienna History Wiki interface. These are consistent with the four question categories used in the evaluation benchmark of this thesis:

- Single-value queries (e.g., "When did Gustav Klimt die?")
- Multi-hop queries (e.g., "Where is the architect of the Karl-Marx-Hof buried?")
- Aggregation queries (e.g., "How many chapels does the Stephansdom have, and what are their names?")
- Description queries (e.g., "Who was Hanns Blaschke?")

Domain experts indicated that aggregation and multi-hop queries are among the most difficult questions to answer because they either need aggregate information across multiple entries or follow chains of relationships between entities. Although Semantic MediaWiki's structured query tool can facilitate answering such queries, it remains difficult to operate in reality, especially for users who lack sufficient knowledge about structured querying.

4.2.3 Sensitive Topics Handling

Domain experts raised concerns about the potential for AI systems to misrepresent or oversimplify sensitive historical topics, particularly those related to Austria's National Socialist period. They emphasized the need for neutral, evidence-based language and cautioned against presenting incomplete or glorified narratives. They suggested that this concern can be addressed through closed-world generation, where the answers must rely on the evidence retrieved from the Vienna History Wiki without speculation. For example, when answering a question about a person's involvement during the National Socialist period, the system reports only the roles, dates, and affiliations available in the retrieved context without additional suggestions.

4.2.4 Data Completeness and Ontology Coverage

The consultation also showed that the completeness and structure of the Vienna History Wiki vary across subject areas. Categories such as street names and biographies of famous individuals are well-represented, while events and miscellaneous historical objects may be less consistently structured. Some certain concepts, such as "politician", may appear in an article without being formally encoded in the ontology. These differences show that relevant information can appear in unstructured articles but not in the RDF representation, or the other way around.

4.3 Implications for System Design

The use case and consultation with domain experts resulted in three main requirements for the system design. Firstly, answers must be based on the evidence retrieved from the Vienna History Wiki and cannot contain unsupported speculation. Secondly, the retrieval system must support several information needs, ranging from direct factual and descriptive questions to aggregation and multi-hop questions. Thirdly, the system comparison must take into account the differences between unstructured article content and structured RDF representation within the platform.

Overall, each requirement affects a different part of the experimental design. The need for a grounding answer is reflected in the closed-world prompt that will be used for the architectures and the metric used to evaluate it. The range of information needs is reflected in the benchmark, which is divided evenly into four question categories. The difference between the articles and RDF representations is reflected in the source-equivalence condition on benchmark construction and in the choice of architectures. The research methodology chapter describes each of these in detail.

5 Research Methodology

This chapter explains the methodology used to answer the research questions introduced in the first chapter. Five retrieval architectures are evaluated with the Vienna History Wiki as the only source of knowledge, under two large language model providers, and across four question categories, while all other experimental parameters are held constant. This chapter describes the experimental design, the compared architectures, the experimental controls and chosen components, and the three evaluation layers.

5.1 Experimental Design

The primary objective of the experiment is to investigate how different degrees of graph integration affect natural-language question answering over the Vienna History Wiki and whether the suitability of a retrieval architecture depends on the type of information requested.

Three factors are varied. The first factor is the retrieval architecture. Five systems are compared, namely, Document Vector RAG, KG-Derived Vector RAG, GraphRAG 1-hop, GraphRAG 2-hop, and Agentic Graph Retrieval. All five architectures are described in the subsequent section. These architectures are chosen because they differ in how knowledge is fed into the retrieval process. This comparison does not assume that integrating more graph structure means better performance.

The second factor is the question category. The benchmark differentiates the questions into single-value, description, aggregation, and multi-hop questions. These categories represent different information structures instead of a strict ordering of task difficulty. With this variation, it can be tested whether an architecture that performs best on one information need stays the best for another information need. The last factor is the generation provider. Every architecture runs under GPT-5.4 and Qwen 3.5 Plus. The benchmark, retrieval configuration, context budget, and evaluation procedure stay the same under these two providers. Therefore, the effect that comes from the retrieval can be separated from the generator effect.

Each research question is related to one part of the design. Research question one compares the vector-based and graph-based architectures within each question category, using the strongest-performing architecture from each family. Research question two compares the Document Vector RAG with the KG-Derived Vector RAG, which use the same embedding-based retrieval mechanism but differ in retrieval unit, specifically document chunks against the atomic RDF-derived facts. This comparison isolates the granularity of the retrieval unit. Research question three compares Agentic Graph Retrieval with Graph RAG 1-hop and 2-hop that use a predetermined expansion limit for each query. As a result, the selective exploration can be separated from the set depth of exploration. Research question four compares two generation providers. Claim-level F1 from the manual statement-level evaluation is used as the main outcome for all comparisons. Research questions two and four further report the precision and recall separately, because the differences that are expected relate to the balance between information that is not supported and missing information, not to overall answer quality.

The methodological mapping can be summarised as follows:

Table 1: Mapping of Research Questions to Architecture Comparisons

Research Question	Primary Comparison	Primary Outcome
RQ1	Vector vs. Graph Architectures $\times$ four question categories	Claim-level F_1
RQ2	Document Vector RAG vs. KG-Derived Vector RAG	Precision and Recall
RQ3	GraphRAG 1-hop/2-hop vs. Agentic Graph Retrieval	Claim-level F_1
RQ4	GPT-5.4 vs. Qwen 3.5 Plus	Precision, Recall, and F_1

5.2 Compared Retrieval Architectures

There are five retrieval architectures that are compared and ordered by increasing graph integration. The architectures differ in their knowledge representation, retrieval unit, graph traversal mechanism, and retrieval termination condition. Table 2 summarises these differences. The implementation of each architecture will be explained in details in Section 6.

Table 2: Comparison of the Evaluated Retrieval Architectures

Architecture	Knowledge representation	Retrieval unit	Retrieval and termination
Document Vector RAG	Unstructured wiki articles	Document chunks	Retrieves the top- k embedded chunks by vector similarity and then stops.
KG-Derived Vector RAG	Verbalised RDF representation	Verbalised triples	Retrieves the top- k embedded triples by vector similarity without graph traversal.
GraphRAG 1-Hop	RDF knowledge graph	Seed triples and direct neighbours	Retrieves vector-based seeds, performs one SPARQL expansion hop, and then stops.
GraphRAG 2-Hop	RDF knowledge graph	Seed triples and two-hop neighbourhoods	Retrieves vector-based seeds and performs two successive SPARQL expansion hops.
Agentic Graph RAG	RDF knowledge graph	Selected relations, entities, and paths	Uses an LLM to select traversal steps iteratively until evidence is sufficient or the maximum depth is reached.

Even though KG-Derived Vector RAG uses information that is based on

a knowledge graph, this architecture is still categorised as vector retrieval architecture since it does not perform any traversal of the graph relation. The two static Graph RAG architectures apply predetermined expansion depths, while the Agentic Graph RAG selects traversal steps dynamically according to the question and the evidence retrieved.

5.3 Experimental Controls and Configuration Selection

Since the research questions compare the differences in performance with retrieval architecture, retrieval-unit granularity, traversal strategy, or generation provider, other experimental factors must be kept as consistent as possible. Firstly, all the systems use the same knowledge source, which is the Vienna History Wiki. Each benchmark question is included only when the information required to answer it is available in both the article representation and the RDF representation. Without this condition, an apparent architectural advantage could arise not because of better retrieval architectures, but because one source representation has information that is not available in the other representation.

Second, the same embedding model and similarity measure are used for all systems that need semantic vector retrieval. This limits the possibility that the difference between vector-based architectures is caused by the differences in embedding space or similarity function instead of retrieval representation or graph expansion that is analysed. Third, all the systems use the same context budget of 6,000 tokens. This constraint is important when comparing flat retrieval with graph expansion because neighbourhood traversal could produce a larger number of evidence items. Without the shared context budget, graph-based systems can obtain an evidence-volume advantage that is not controlled.

Lastly, final answer-generation conditions are kept consistent. The two LLM providers receive evidence that is produced by the retrieval architecture with the same grounding constraints and answer-style. The final retrieval configurations can be seen in the table below.

Table 3: Retrieval Configurations of the Evaluated Architectures

Architecture	Retrieval Configuration
Document Vector RAG	Text chunks, $k = 15$
KG-Derived Vector RAG	Verbalised triples, $k = 90$, no expansion
GraphRAG 1-hop	$k = 40$ seeds, hop-1 limit = 5
GraphRAG 2-hop	$k = 20$ seeds, hop-1 limit = 5, hop-2 limit = 5
Agentic Graph Retrieval	Beam width = 20, maximum depth = 3, up to 5 seed entities

These are the configurations used in the final comparative experiment, which were established through a preliminary ablation study under the common context constraint. Candidate configurations were compared according to retrieval coverage while excluding settings that produced context sizes that were too close to or above the shared token budget. The complete configuration grid can be found in the Appendix [B](#) section.

5.4 Component and Model Selection

The architectures that are compared in this thesis use the same infrastructures which make the differences in the results can be attributed to the retrieval strategy instead of the components. Therefore, four decisions are made once and held constant. The embedding model defines how texts and verbalised triples represented for every vector-based retrieval step. The retrieval infrastructure covers the similarity score, index structure, vector store, and triple store, which all decide how the units are retrieved and returned. The generation and the judge models decide how the answer is generated and evaluated. The prompt requirements define the generation behaviour that all the architectures must follow. The selection criteria used are the suitability of the component with the content of the Vienna History Wiki, which is mainly in German, retrieval or generation performance, availability for local deployment, and reproducibility.

5.4.1 Embedding Model Selection

The embedding model is chosen based on four main requirements. First, the model needs to support German language since most of the Vienna History Wiki content uses German language while also being suitable for English-language queries. Second, the model needs to stay computationally feasible within the existing hardware environment. Third, high retrieval performance is the highest priority in choosing embedding model because the main role for embedding model in RAG architectures is to identify relevant evidence. Lastly, preference was given to models that could be used locally and reproducibly without dependence on a restricted embedding service.

Candidate models were identified using the Massive Text Embedding Benchmark (MTEB), which evaluates embedding models across retrieval and other semantic representation tasks through its Hugging Face leaderboard [1](#). The initial screening considered the language support, memory requirements, model size, embedding dimensionality, maximum input length, accessibility,

¹MTEB Leaderboard, <https://huggingface.co/spaces/mteb/leaderboard>, accessed 16 February 2026

and performance on retrieval-related benchmark tasks. The technical filtering was based on the computational resources available for this study. Therefore, models with memory requirements below approximately 2.5 GB, fewer than 0.6 billion parameters, and embedding dimensionality below 3,072 were prioritised. Retrieval performance was treated as the primary MTEB criterion, while reranking and Semantic Textual Similarity (STS) were used as supporting indicators. Five models remained after this filtering.

Table 4: Top 5 Candidate Embedding Models by Score on the Massive Text Embedding Benchmark

Model	Bitext	Class.	Clust.	Rerank	Retrieval	STS
KaLM-embedding-multilingual-mini-v1	64.77	56.40	44.64	51.07	52.68	60.32
granite-embedding-278m-multilingual	58.48	54.09	41.41	59.89	52.20	67.27
bilingual-embedding-small	69.48	57.45	43.13	57.49	51.68	66.96
bangla-sentence-transformer	67.99	37.56	37.75	73.50	51.39	39.73
multilingual-e5-small	69.40	56.97	41.30	60.43	50.91	72.08

The comparison shows that the strongest candidates achieved relatively similar retrieval scores. Kalm-embedding-multilingual-mini-v1 obtained the highest retrieval score of 52.68, followed closely by Granite-Embedding-278M-Multilingual with 52.20. The difference between the two is only 0.48 points, while Granite is considerably smaller with 278 million parameters and 768-dimensional embeddings. Since the full corpus of document chunks and verbalised triples has to be embedded on the available hardware, the smaller model was preferred at an almost identical retrieval performance. Multilingual-E5-Small achieves higher reranking and STS scores, but a lower retrieval score, which is the primary criterion of this selection, while the remaining candidates do not provide sufficient overall advantage to justify their selection. The model is also suitable to the bilingual setting of this thesis since it supports both German and English languages. This is important since most the retrieved content of Vienna History Wiki is in German, while the benchmark questions and generated answers are in English. Furthermore, its compatibility with SentenceTransformers ecosystem allows the same embedding procedure to be applied consistently throughout all vector-based retrieval architectures. In addition, the model contains 278 million parameters and produces 768-dimensional embeddings. Therefore, Granite-Embedding-278M-Multilingual is used as the shared embedding model, and is used for Document Vector RAG, KG-Derived Vector RAG, and the vector-based seed retrieval components of the static GraphRAG architectures.

5.4.2 Retrieval Infrastructure

Cosine similarity is chosen as the common vector-similarity measure for this study. This decision is suitable with the normalised representations that are produced by the embedding model. Furthermore, since the embeddings are already unit length, it can be computed as a dot product between normalised vectors. This reduces the query latency and is natively supported by modern vector databases. Additionally, it is also the measure most commonly used in semantic retrieval. The alternatives considered are not really aligned with the purpose of this thesis. Euclidean distance produces the same ranking as cosine similarity for vectors that are unit normalised, Manhattan distance applies the comparison per dimension that is not directly relevant to the directional comparison used for dense semantic embedding, while Jaccard similarity is defined for set-based or binary representations. As a result, cosine similarity is chosen.

Hierarchical Navigable Small World (HNSW) is used as the index structure for all static architectures. A comparative study reports that there is no single configuration that universally performs better than others. HNSW, IVF, and ScaNN are able to reach aggregate result that are similar in the retrieval metrics and generation quality, but they differ in the latency and configuration that performs the best for certain metric [8]. Therefore, HNSW is chosen not because it outperforms other alternatives, but it is because of three main reasons. First, graph-based approximate nearest neighbour method offers a good balance between retrieval accuracy and efficiency for high-dimensional vector search [21]. Second, HNSW is one of the most supported method in different vector database which beneficial for reproducibility and integration. Third, HNSW does not need separate tuning effort as much as IVF or post-processing such as product quantisation, which is suitable with transformer-based embedding spaces and dataset with the size of Vienna History Wiki.

Qdrant is used as the vector-storage and retrieval backend. It implements HNSW as its primary indexing strategy with a built-in cosine support. It also runs entirely on local infrastructure through Docker without cloud access or usage costs and exposes a REST API and a Python client that integrate with the rest of the pipeline [2]. Furthermore, every vector point brings JSON payload so the answer can be filtered based on documents, namespace, or date range without the need to re-embed the knowledge base.

QLever is used as the triple store. Graph-based architectures need persistent access to RDF representation for entities, predicates, literals, and

²<https://qdrant.tech/documentation/overview/> and <https://qdrant.tech/documentation/quickstart/>, accessed 21 February 2026.

neighbourhoods retrieval through SPARQL continuously. As a result, the selection criteria are support for the RDF and SPARQL standards, efficient read-oriented querying, scalability to the processed graph, local deployment, and reproducibility. Apache Jena Fuseki, OpenLink Virtuoso, GraphDB Free, and QLever were considered. QLever was chosen because it is designed for efficient SPARQL querying over large RDF datasets and provides a freely deployable implementation that is suitable for reproducible research.

5.4.3 Generation and Judge Models

The language model that can be used to produce answers is divided into two groups. The first group consists of proprietary models, whose weights are not released and which can only be used through the provider’s API, such as the OpenAI models. The second group consists of open-weight models, whose parameters are publicly released and which can be deployed locally. Open-weight models are released in various sizes. The Qwen 3.5 family, for example, includes small models such as Qwen3.5-0.8B, Qwen 3.5-2B, and Qwen 3.5-4B, medium models including Qwen3.5-27B, Qwen3.5-122B-A10B, and Qwen 3.5-397B-A17B³.

This thesis evaluates the answer generation with one model from each group to the five pipelines, namely `gpt-5.4` from OpenAI as a proprietary model and `qwen3.5-plus` from Alibaba as the hosted version of an open-weight model. Qwen 3.5-Plus is the hosted version that corresponds to Qwen3.5-397B-A17B, with additional production features such as a 1 million context length by default. Therefore, Qwen3.5-Plus represents the open-weight model while being accessed through an API in the same way as GPT-5.4 Both models were released within a similar time frame. Hence, the comparison compares two groups without a large difference in model maturity. On public benchmarks, GPT-5.4 scores higher than Qwen3.5-397B-A17B overall⁴. For this reason, GPT-5.4 is expected to achieve a higher answer quality, as stated in the hypothesis 4. For both models, extended reasoning is disabled to ensure that none of the generators receives additional reasoning steps.

There are three main reasons as to why the small open-weight model is not used. First, API access makes the experiments feasible and repeatable without local GPU infrastructure. The hardware available for this thesis could only run very small open-weight models locally, which could have limited the comparison to a model far smaller than GPT-5.4 Such a model would also

³Qwen3.5 collection on Hugging Face, <https://huggingface.co/collections/Qwen/qwen35>, accessed 21 March 2026.

⁴LLM Stats, <https://llm-stats.com/leaderboards/llm-leaderboard?orgs=openai,qwen>, accessed 21 March 2026.

differ considerably in capability. Hence, any difference from GPT-5.4 would mix the effect of the model group with the effect of model size and result in an unfair comparison. Second, the Agentic Graph RAG uses the language model not only to generate the final answer but also to select relations, score candidate entities, and decide whether the retrieved evidence is sufficient at every depth. Therefore, a weaker model would affect the traversal itself and make it harder to attribute differences to the retrieval strategy. Third, the main research question concerns the retrieval architectures. A local deployment would add further variables, such as quantisation, the serving framework, and hardware-dependent limits, which could affect the answers independently of retrieval. Each pipeline exposes an `LLM_PROVIDER` setting which allows the same prompts, retrieval configuration, benchmark, and evaluation procedure to be repeated under either model without changing the pipeline itself.

A different model is used for the evaluation. GPT-4o-mini is used as the judge model for the RAGAs evaluation regardless of the generation model that produced the evaluated answer. Keeping the judge constant and separated from both of the generators make the scoring result comparable across systems and providers while also preventing one generator to measure its own output.

5.4.4 Prompt Requirements

The experts point out that the answer needs to be based on what the Vienna History Wiki actually contains and must not fill the gaps with speculation. Therefore, the generator cannot be left to answer freely. Consequently, there are four requirements apply to the system instruction prompt for every architecture to ensure that all five systems generate under the same conditions. The first requirement is role assignment. The generator is given a persona as a research assistant for Vienna History Wiki. This gives the generator a task that needs to be done and pushes the model to produce answers that are correct and factual, based on the retrieved content and not based on the general knowledge base.

The second part is context limitation. The model is explicitly instructed to only answer based on the provided context. If the answer is not located in the retrieved context, the model is instructed to answer using the fixed fallback, "I don't know based on the indexed data". This constraint allows the hallucination to be measured since every factual claim that is unsupported by the retrieved context can be identified as a grounding failure.

Thirdly, there is the multilingual handling. Vienna History Wiki is a German-based knowledge source, while the interface system operates in English language. Consequently, every prompt instructs the model to keep

the nouns in the original German language, including the name of a person (for example, Karl Seitz), street name (for example, Mariahilfer Strasse), and the name of a building (for example, Stephansdom). As for other contents, they are translated into English language.

The final part is the answering style. The model is instructed to answer briefly and only provide the specific facts that are asked. As for the questions that require aggregation or counting, the model needs to mention every item explicitly with its final count. As for the descriptive questions, the model must limit the answer to the date, job, affiliation, and role that are asked and appear in the context without adding additional context beyond what is asked for and available.

It is important to note that some additional architecture-specific instructions are introduced only where the retrieved representation requires them. The knowledge-graph-based architectures receive additional instructions that address properties of the verbalised triple format instead of the task of answering the question, and the Agentic Graph RAG requires separate instructions for relation and entity selection. The grounding constraint, multilingual handling, and answer style stay the same across the five architectures.

5.5 Evaluation

In this study, the evaluation of five RAG systems will be done at three levels of evaluation, namely, retrieval-based metrics, LLM-judged answer metrics, and a manual statement-level assessment against the gold answers. It is important to apply the multi-level evaluation because judging a RAG system by only its retrieval component can be misleading as retriever relevance scores have been shown to correlate weakly with the performance of the full system [31]. The manual statement-level evaluation is used as the main measurement point of the answer quality in this thesis. Retrieval quality assessment in the form of term recall and automated LLM evaluation of RAG Assessment metrics are used as additional layers that is used to isolate the failure location, namely whether the evidence needed reached the generator and whether the generated answer is grounded to the evidence. Therefore, this chapter introduces the benchmark and explains each evaluation layer used in this study.

5.5.1 Benchmark

All systems are evaluated against a single, human-created benchmark of questions. This benchmark consists of 120 question and answer pairs over the Vienna History Wiki which are split evenly across the four question categories. One fixed benchmark is used for every architecture, provider, and metric layer

so that any differences in the reported results are due to the system that is tested.

The four question categories are differentiated based on the different needs of retrieval. The **single_value** questions ask for one attribute from one entity, and the answer is usually already contained in one fact. The **description** questions are still focused on one entity, but ask for several facts that form a complete answer. The **aggregation** questions require the system to collect several pieces of information while also ensuring that these information is complete. Lastly, **multi_hop** questions depend on the relational connection of facts between one or more entities or intermediate entities, so the next evidence may seem relevant after the previous relation is found. As a result, every category tests different retrieval aspects, namely direct lookup, coverage on several facts, completeness of a collection, and exploration of connected evidence. The benchmark contains 30 questions of each category so that the architecture evaluation is conducted on equally sized samples.

Table 5: Example Benchmark Question and Required Evidence per Question Category

Category	Question	Required evidence
single_value	When did Gustav Klimt die?	The entity and death date
multi_hop	When was the architect of Karl-Marx-Hof born?	The building, its architect, and the architect’s birth date
aggregation	How many chapels does the Stephansdom have?	The entity, the relation, and every member of the set
description	Who was Anton Bruckner? state his occupation, awards, birth and death dates, and burial place	Occupation, awards, birth and death dates, burial place

Each item in the benchmark includes the question text, category, a written **gold_answer**, and a list of **gold_evidence_terms**. Gold evidence terms are the entities, predicates, and literal values that are required to answer the question. For the single-value question in Table 5, the gold answer states that Gustav Klimt died on 6 February 1918, and the gold evidence terms are *Gustav Klimt* and *1918-02-06*. The gold answer is the reference for the LLM judge and the manual evaluation. The distinction between generation and retrieval allows the framework to identify failure modes that are related to retrieval and generation. The gold evidence terms are used to check the retrieved context before getting into the generator, while the gold answers are compared against what the model generated.

The creation of the benchmark relied on the input from the expert consultation. These were expanded into additional benchmark items through manual construction from the Vienna History Wiki. Each question was manually checked against the source and included only when the information required to answer it was available in both the article-based and RDF-based representations.

5.5.2 Evaluation Protocol

The evaluation is implemented in three stages, starting with *collect*, which retrieves and generates the answer once for each question and system configuration. Then *evaluate* that computes the deterministic retrieval-grounding metrics from the stored contexts. Lastly, the *judge* which calculates the RAGAs metrics using an LLM judge. Separating the process into stages avoids the redundant repeated retrieval and generation which allows the deterministic metrics to be recomputed without additional model calls.

5.5.3 Retrieval-Based Evaluation

This first layer focuses on answering one question "Did the system retrieve the evidence needed to answer the question regardless of how the generator used the evidence?" It uses the lexical matching approach described in Section 2.5. Since the same fact can appear in different wordings, exact string matching can miss valid synonyms or alternative expressions. It is because of this, every gold evidence term is extended into several variations before the matching. From here, a table consisting of synonyms of specific words is used to map the predicate to its alternative words that have the same meaning. As an example, `birthDate` can also be matched with *born*, *geboren*, or *Geburtsdatum*. The date in the ISO format is also expanded into several calendar writing alternatives in both English and German. The numeric number is also matched with and without a trailing decimal point. A term counts as found when at least one of its valid variations is found in the retrieved context.

Term Recall is the fraction of a question's gold evidence terms found in the retrieved context stored during the *collect* stage.

$$\text{TermRecall} = \frac{|\text{gold terms found}|}{|\text{gold terms}|}, \quad (11)$$

This matching process uses simple string matching and not cosine similarity, BM25, or any other semantic matching. This is used to keep the metric as deterministic and transparent as possible. The numerator represents the total number of gold terms found in the retrieved context, while the denominator

shows the total number of gold evidence terms expected for that particular question. The score ranges between 0 and 1. The higher the score, the more evidence is found in the retrieved context that is sent to the generator, while a lower score means that more of the required evidence is missing from the context.

5.5.4 RAG Assessment (RAGAs)

Faithfulness, Context Recall, Answer Correctness, and Answer Relevancy are calculated using the RAGAs framework without modification, following the definitions in Section 2.5. The judge is `gpt-4o-mini` and the embedding model is `text-embedding-3-small`, both fixed across every system, provider, and run.

Faithfulness is computed against the retrieved contexts collected in the *collect* stage and measures how well the generated claims are supported by the retrieved evidence. Context Recall uses the gold answer as a proxy for the required evidence. Therefore, context recall complements the term recall because both of them have the same purpose with different methods. Answer Correctness gives greater importance to factual correctness while still accounting for semantic equivalence. Hence, this metric complements the manual TP, FP, and FN evaluation described in the following subsection.

Answer Relevancy is computed by asking the judge three independent times to invent the question that the generated answer would be answering and to flag whether the answer is noncommittal. The score is turned to zero when every reconstruction flags the answer as noncommittal. This is relevant because the closed-world systems are permitted to answer *Not available in the indexed data*, which automatically receives zero relevancy by construction.

Although it has shown strong agreement with human judgement, particularly for Faithfulness, its claim decomposition and scoring still depend on an LLM evaluator and may therefore be affected by model-specific variability [9]. In addition, aggregate scores can obscure mixtures of correct, incorrect, and missing information within the same response [30]. For this reason, the primary evaluation in this thesis is conducted manually at the statement level against the gold answers, while RAGAs is used to support and interpret the manual results.

5.5.5 Manual Evaluation

The manual statement-level evaluation evaluate each generated answer directly with the gold answer without using the LLM as an evaluator. First, the gold answer is decomposed into atomic gold claims that represent facts that

have to exist in the answer. List members are treated individually, full titles and dates are evaluated as complete claims, and grave-plot information is decomposed into *Gruppe*, *Reihe*, and *Nummer*. For aggregation questions, each required list member is scored separately, while the overall count is treated as an additional claim only when it is explicitly included in the gold answer. The complete decomposition rules are provided in Appendix C. Second, the generated answer is decomposed into atomic statements, which are judged against the gold claims without any external reference and labelled as true positive (TP), false positive (FP), and false negative (FN) according to Table 6. From these three labels, the sum is used to measure the precision, recall, and F1 defined in Section 2.5. With this, the evaluation not only shows whether an overall answer is correct, but it also differentiates between correctly stated information, unsupported information, and missing information.

Table 6: Criteria for Classifying Statements as True Positive, False Positive, or False Negative

Outcome	Criterion	Notes
True positive (TP)	An atomic statement states a required gold claim correctly.	Correct paraphrases, aliases, and German-English equivalents. A required count is a TP only when the gold explicitly asks for it.
False positive (FP)	An atomic statement provides something the gold answer does not support.	Hallucinations and wrong entities, properties, or values, together with competing or additional entities, extra list members, an unsupported day or month added to a correct year, an incorrect bridge entity, and a wrong count. Statements are judged strictly against the gold.
False negative (FN)	A required gold claim is absent from the answer.	The formal expression of an incomplete answer. Includes each missing list member. A refusal (<i>Not available in the indexed data</i>) turns every gold claim into an FN and produces no FP unless it also states a wrong fact.
Unscored	A non-factual procedural or meta-commentary statement that does not assert an answer fact.	Contributes to no count: procedural or meta commentary (e.g. “based on the retrieved context”)

There are no true negatives in this evaluation since true negative would be a statement that the system correctly did not make, and that set is unbounded in open-ended answer generation. Hence, accuracy is not defined and the evaluation combines the TP, FP, and FN to measure precision, recall, and F1.

As an example, the multi-hop question "In which grave plot is the architect of the Akademie der bildenden Künste buried?" has the gold answer "Gruppe 14A, Nummer 20", which is decomposed into two gold claims, the *Gruppe* and the *Nummer*. GraphRAG 1-hop with Qwen answered "Nummer 20". This results to 1 true positive for the number, but also 1 false negative for the Gruppe. Therefore, the answer reaches a precision of 1, but only a recall of 0.5, resulting in an F1 of 0.667.

6 Implementation

This chapter explains how the five retrieval architectures introduced in previous sections are implemented. This chapter focuses on how each pipeline was built and how information moves through it. The explanation is divided into three parts, namely the baseline Document Vector RAG, the three knowledge-graph-based RAGs, and the Agentic Graph RAG.

Figure 2: Overview of the Evaluated System

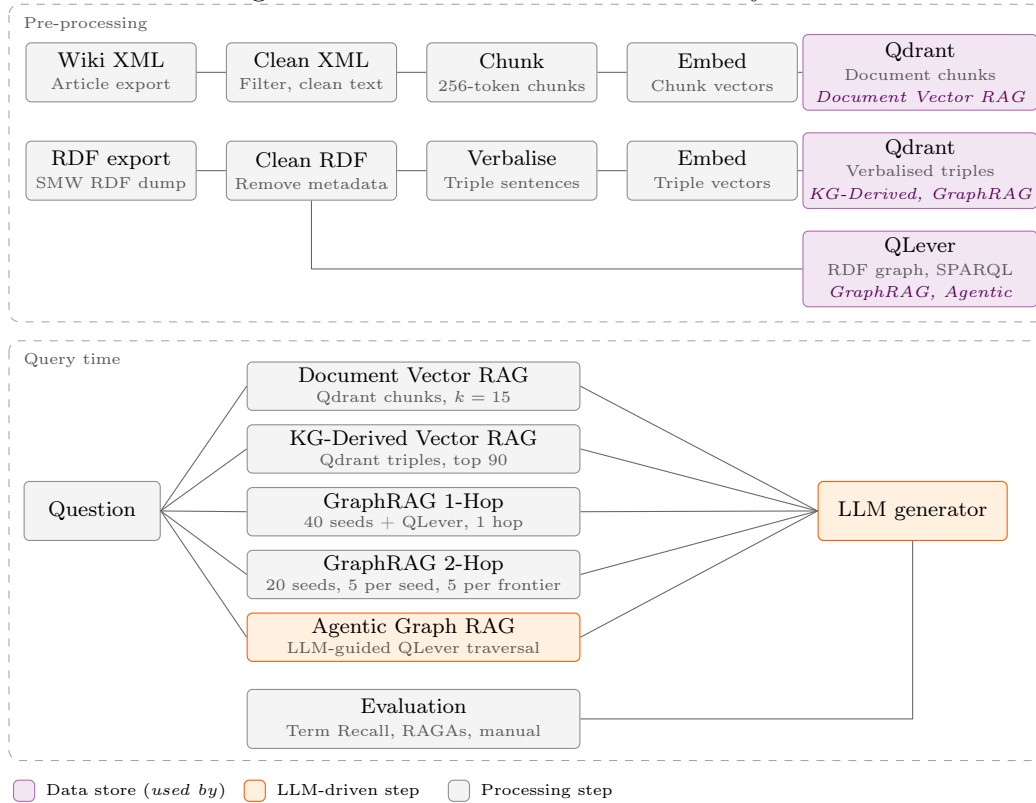


Figure 2 gives an overview of the complete system, from the pre-processing

of both knowledge representations, to the five retrieval pipelines, the generation step, and the evaluation.

6.1 Document Vector RAG

Document Vector RAG acts as the baseline that is used as a comparison for the other architectures. The implementation is divided into two phases. The first phase is carried out once during indexing. The XML MediaWiki dump from the Vienna History Wiki is parsed, cleaned, split into chunks, embedded, and written to Qdrant. The second phase runs when the query is submitted to the system. The query is embedded with the same embedding model, the most similar chunks are retrieved from Qdrant, and the retrieved context is sent to the generator. The sections below follow that order, describing how the index is built before turning to the query-time pipeline.

6.1.1 Input Pre-Processing

The document pipeline is built from the MediaWiki XML export of the Vienna History Wiki which represents every wiki page as one XML record with its title, namespace, revision, and raw wikitext content ⁵. Before the embedding process is conducted, the export results need to be filtered and cleaned so that the vector index contains historical article content instead of administrative page or wiki markup. The cleaned text is then divided into retrieval units.

Namespace Filtering Every MediaWiki page belongs to a namespace that indicates what the page is for, marked by a title prefix such as User:, Help:, or Talk: [33]. Each namespace is represented by a numerical identifier in the XML export [25]. For this pipeline, only three namespaces are kept, namely, namespace 0 (main), 6 (file), and 14 (category). Namespace 0 contains the primary article pages describing people, places, events, and objects in Viennese history. This is the core content of the corpus. Namespace 6 describes image files and other media, which often contain relevant captions and provenance notes about historical artefacts. Lastly, namespace 14 consists of category pages that group related articles and provide an organisational overview of topics covered by the wiki. All other namespaces such as talk pages, user pages, template pages, MediaWiki system pages, and help pages are excluded because they only contain administrative content instead of historical information. Keeping them would introduce noise into the embedding space and risk surfacing non-content pages in response to substantive queries.

⁵MediaWiki XML export schema 0.10, <https://www.mediawiki.org/xml/export-0.10.xsd>, accessed 21 January 2026.

Within each namespace that are being kept, redirect pages are removed. A redirect page contains no substantive text since its sole purpose is to forward the reader to another page. In German-language MediaWiki exports, redirect pages begin with either `#REDIRECT` (the English MediaWiki syntax) or `#WEITERLEITUNG` (the German equivalent). Any page whose first non-whitespace content matches one of these markers is dropped before further processing. Indexing redirect pages would create duplicate entries in the vector store since they point to pages whose content is already indexed under the article’s actual title.

Wikitext Cleaning The remaining pages contain raw wikitext which must be resolved or removed before embedding. Embedding models process plain text while markup characters are interpreted as literal content instead of formatting instructions. In this thesis, it is applied in five steps. First, the MediaWiki `<ref>` elements are removed, including both paired references such as `<ref>...</ref>` and self-closing forms such as `<ref.../>`. They are removed because citation metadata such as publication titles, page numbers, and URLs do not contribute to the factual content that the embedding model should encode. Therefore, their existence could reduce the semantic signal of the surrounding historical text.

Second, remaining HTML and XML tags, including elements such as `<br>`, `<gallery>`, `<div>`, and `<span>` are stripped because they mainly represent presentation without semantic meaning for retrieval. Their removal is performed with a single regex pass that matches any angle-bracket-delimited element. Third, internal wiki links are simplified to their visible text. For example, `[[Wien|Vienna]]` becomes *Vienna*. Fourth, HTML character entities are converted to their Unicode forms using Python’s `html.unescape()`, ensuring that German characters are represented as natural text rather than encoded entity strings. Finally, whitespace is normalised by removing repeated spaces and blank lines and trimming leading and trailing whitespace. This reduces unnecessary token usage and preserves clean paragraph boundaries for the subsequent chunking stage.

After these steps, each retained page is represented as a clean document record containing a document identifier, page title, canonical wiki URL, and cleaned plain-text body. This produces a consistent textual corpus that can be segmented into retrieval units in the subsequent chunking stage. The complete pre-processing procedure is provided in Figure 17 in Appendix D.

6.1.2 Chunking

As for this thesis, a chunk size of 256 tokens was selected. This is within the 512-token context window of the Granite-278M-Multilingual embedding model and is short enough to keep the semantic specificity. Token counts are measured using the model’s own tokeniser (loaded via HuggingFace Transformers) instead of a character count to ensure that the boundary is precise with respect to the model’s actual input representation.

Instead of splitting on a fixed character boundary, the pipelines in this thesis cut the text on the sentence boundary using the Punkt sentence tokeniser from NLTK that is configured for the German language (`nltk.sent_tokenize(..., language="german")`). The split that takes into account the sentence boundary prevents a cut in the middle of a sentence and helps produce chunks that are complete semantically, which is better than embedded forced-cut fragments. The Punkt tokeniser is trained on German abbreviation patterns such as *Dr.*, *z.B.*, and *bzw.*. As a result, it does not incorrectly split at abbreviation final periods, which are common in German historical texts.

Furthermore, an overlap of one sentence is applied between consecutive chunks. Once a chunk has been completed, the last sentence is carried to the next chunk as an opening sentence. This method keeps the reading flow intact across boundaries. Therefore, a fact spanning the tail of one chunk and the head of the next appears in full in at least one of them. One sentence is enough to achieve this without creating duplicated content or significantly inflating the vector index.

It is important to note that each completed chunk is additionally prefixed with its source title and its position within the article. The prefix preserves information about the source entity when a fragment is embedded independently from the rest of the article. The resulting chunk records contain the chunk identifier, source document identifier, page title, locator, contextualised text, chunk position, and total number of chunks for the source document.

During chunk construction, sentences are accumulated until adding another sentence would exceed the 256-token limit. The current chunk is then finalised and its last sentence is carried into the next chunk as overlap. Individual sentences exceeding the limit are divided into token-based segments of at most 256 tokens, ensuring that no retrieval unit exceeds the embedding limit. The complete chunk-construction algorithm, including the long-sentence fallback and overlap handling, is provided in Figure [18](#) in Appendix [D](#).

6.1.3 Embedding and Index Construction

Each contextualised chunk is encoded using the Granite-Embedding-278M-Multilingual that was chosen previously. This model produces one vector with 768 dimensions for every chunk, with L2 normalisation applied during encoding. The same procedure also applied to the user query to make sure that the corpus representation and query representation exist in the same vector space.

These vectors are stored in the `thesis_chunks` collection in Qdrant, which is configured with cosine similarity and HNSW index. Besides embedding, each point also has the document and chunk metadata, namely `doc_id`, `title`, `location`, `chunk_index`, `total_chunks`, and the text chunks that have been given contexts. This metadata is returned with each of the retrieval results and is to form context that is given to the generator.

This step completes the offline construction of the Document Vector RAG index. No further corpus processing is required during evaluation, and all remaining operations are performed independently for every question submitted. Figure 3 shows the payload of one point in the `thesis_chunks` collection, namely the last chunk of the article on Ferdinand Aberer. The chunk text is abbreviated.

Figure 3: Example Qdrant Payload of a Document Chunk

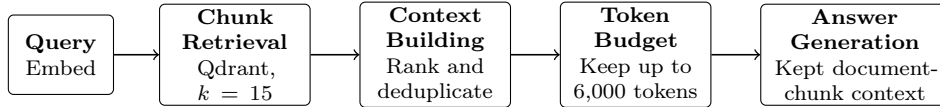
```
{
  "doc_id": 5,
  "title": "Ferdinand Aberer",
  "location": "Ferdinand Aberer#chunk3",
  "text": "Title: Ferdinand Aberer\nSection: 3/3\n\nWar maßgebend an der Erschließung der österreichischen Erdölfelder beteiligt, ... Goldenes Verdienstzeichen der Republik Österreich (1963). ...",
  "chunk_index": 2,
  "total_chunks": 3
}
```

6.1.4 End-to-End Pipeline

Once the document index has been constructed, each question is processed through the same query-time sequence. The entry point of the pipeline is the `ask()` function. It embeds the input question, retrieves the top-ranked document chunks from Qdrant, constructs the generation prompt, and submits

the resulting prompt to the selected language model. The complete data flow through the pipeline can be seen below:

Figure 4: End-to-End Pipeline of Document Vector RAG



The input question is first encoded using Granite-Embedding-278M-Multilingual with L2 normalisation that is used in the same way as the corpus indexing. The resulting 768-dimensional query is then sent to the Qdrant collection. There, HNSW index is explored, and the top 15 most similar chunks are returned in descending order of cosine similarity score. Every hit is returned with its similarity score and the payload fields.

Retrieved chunks are then inputted into the generation prompt based on the similarity score, each given a sequential number prefix and source metadata to allow for source tracing. The prompting principle in general and generation setting have been explained in Section 5.4.3, while the Figure 5 below shows the concrete template that is used for this pipeline. During the query time, context placeholder is replaced with the retrieved chunks and question placeholder is replaced with the actual question.

Figure 5: Prompt Template used by Vector RAG Pipeline

```

You are a research assistant for the Vienna History Wiki. Answer ONLY
from the provided context. If the answer is not present, respond
exactly: "I don't know based on the indexed data." Keep German proper
nouns in their original form (person, street, place, and building
names); translate all other text to English. Be concise and answer
only the specific fact asked. For counting questions, list each item
and then state the total.
Context:  numbered retrieved chunks
Question:  question
  
```

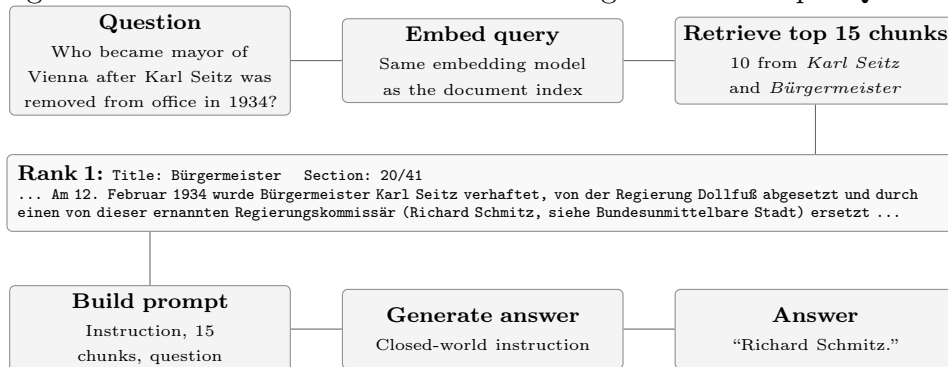
The completed prompt is submitted to either GPT-5.4 or Qwen 3.5 Plus, depending on the active provider configuration. Both providers receive the same retrieved context and prompt structure so that the retrieval pipeline remains unchanged when the generation provider is switched. The generated response is returned as the final answer for the input question. The complete query-time algorithm, from question embedding to answer generation, is summarised in Figure 6.

Figure 6: Algorithm of Document Vector RAG

Require: Qdrant collection `thesis_chunks`, Granite-278M-Multilingual embedder
Query embedding
1: $model \leftarrow \text{GETEMBEDDER}()$
2: $\mathbf{q} \leftarrow model. \text{ENCODE}([query], \text{normalise}=\text{True})[0]$
Retrieval
3: $points \leftarrow \text{QDRANTQUERY}(\text{thesis_chunks}, \mathbf{q}, k)$
4: $hits \leftarrow []$
5: **for** each point p in $points$ **do**
6: $h \leftarrow \{p.id, p.score\} \cup p.payload$
7: append h to $hits$
8: **end for**
Prompt construction
9: $blocks \leftarrow []$
10: **for** $i = 1$ **to** $|hits|$ **do**
11: $header \leftarrow \text{rank } i, \text{ title, doc_id, and chunk position of } h_i$
12: append $header \parallel h_i.text$ to $blocks$
13: **end for**
14: $context \leftarrow \text{JOIN}(blocks)$
15: $prompt \leftarrow \text{system_instruction} \parallel query \parallel context$
Generation
16: $answer \leftarrow \text{GENERATEANSWER}(prompt)$
17: **return** $\{query, answer, hits, neighbourhood_text=""\}$

As an example, consider the multi-hop question “Who became mayor of Vienna after Karl Seitz was removed from office in 1934?”. Figure 7 shows how this question is processed. Ten of the 15 retrieved chunks come from the articles *Karl Seitz* and *Bürgermeister*, and the highest-ranked chunk names Richard Schmitz in the same sentence as the removal of Karl Seitz.

Figure 7: Document Vector RAG: Processing of the Example Question



Based on this context, GPT-5.4 answered “Richard Schmitz.”. Since the

successor is named in the same sentence as the removal of Karl Seitz, a single retrieved chunk is sufficient to answer the question.

6.2 Knowledge Graph-Derived RAG

The three remaining static architectures work together on the structured RDF representation of the Vienna History Wiki and use the same one time pre-processing. The result of the RDF export is first cleaned and converted into N-Triples. Then, the resulting triples are verbalised and embedded into Qdrant for semantic retrieval, while the cleaned graph is indexed in QLever to allow the two GraphRAG variations for neighbourhood expansions through SPARQL. The following sections explain the preparation steps before turning to the query-time behaviour of each architecture.

6.2.1 RDF Pre-Processing

The purpose of this step is to clean the raw RDF export of the Vienna History Wiki and convert it into an N-triples file that can later be indexed in QLever. The raw Vienna History Wiki RDF export contains many triples that are not directly useful for question answering. These include internal MediaWiki and Semantic MediaWiki metadata, such as page IDs, revision IDs, page length, page creation dates, modification dates, approval status, and other system-level properties. If these triples were kept, they could add noise to the retrieval process and reduce the quality of the retrieved context.

The RDF file is first loaded into an RDF graph. Then, a namespace map is built using the `swikt:wikiNamespace` predicate. This namespace map is used to identify which wiki namespace each subject belongs to. The kept namespaces include namespace 0 for the main article, 6 for file namespace, 14 for category namespace, and 102 for property namespace. The first three namespaces contain the same content as in the document section and are kept with the same justification. However, for namespace 102, it does not have the equivalent in the article export. This namespace contains the property definition of Semantic MediaWiki, which gives a readable label for every predicate, making it important in the verbalisation process. Triples whose subject contains a namespace outside of the allowed namespaces are also removed.

Each triple in the graph is then passed through a sequence of cleaning rules. Triples with empty subjects and skipped predicates are removed because they do not describe the historical subject of a page. The skipped predicates include `wikiPageSortKey`, `wikiPageCreationDate`, `wikiPageModificationDate`, `wikiPageContentLanguage`, `specialImportedFrom`, as well as wiki-specific prop-

erties such as `Number_of_revisions`, `Page_ID`, `Page_length`, `Revision_ID`, `Is_a_new_page`, `Approved_revision`, `Approval_status`, `Icon`, `Has_query`, and image-metadata predicates, including `schema:thumbnailUrl`, `schema:creator`, `schema:license`, `Bildquelle`. Predicates whose URI ends with `-23aux` are also removed as these represent SMW subobject auxiliary triples with no semantic content.

The script also removes triples whose subject or object URI contains `QUERY`, as these are generated by inline wiki queries instead of facts about an entity. The literal object is checked last. An empty literal is removed because it states a property without any value, and literals that start with `REDIRECT` markings, such as `#REDIRECT` or `#WEITERLEITUNG`, are removed because they carry the forwarding instruction instead of any content about the entity.

The remaining literal values are cleaned by normalising the whitespace and removing stray tabs and carriage returns. This prevents the verbalised text from inheriting formatting artefacts from the export. Triples that pass all the filters are serialised as valid N-Triples, with subject and object URIs enclosed in angle brackets, blank nodes written in the `(.)` for, and literals escaped according to the N-Triples specification. The resulting cleaned graph is used as the common structural representation for both QLever graph indexing and verbalisation pipeline. The complete preprocessing algorithm and the full filtering logic are provided in Figure 19 in Appendix D.

6.2.2 QLever Graph Construction

The cleaned N-Triples file produced in the previous step is a static file that cannot be queried efficiently on its own. Consequently, this file needs to be upserted into QLever to make the RDF relations accessible through SPARQL query during the neighbourhood expansion.

The QLever instance is deployed in a dedicated Docker container with eight worker thread and memory limit of 5 GB. After the upserting is done, the server provides a local SPARQL endpoint that can be queried by the retrieval pipeline. This graph remains unchanged throughout the evaluation and is used by both GraphRAG 1-hop and 2-hop whenever neighbourhood information is required.

6.2.3 RDF Triple Verbalisation

RDF triples exist as URI-to-URI or URI-to-literal relationships that a neural embedding model cannot directly process. To make the knowledge graph accessible to the same Granite model used, each triple must first be converted

to a plain-text string that the model can encode into a vector. This conversion is also important because it makes access to the knowledge graph more deterministic instead of relying on the LLM to generate SPARQL queries at runtime. This is essentially the idea of verbalisation, which transforms a machine-readable triple such as `<Karl_Seitz> <Has_birth_date> "1869-09-04"^^xsd:date` into a human-readable line such as *Karl Seitz Has birth date 4 September 1869*, which the embedding model can encode into the same vector space as a natural language question.

The verbalisation script reads the cleaned N-triples file produced in the first step above and writes one JSON record per triple. Unlike a simple one-pass conversion, the script runs over the input file twice because a large share of the Vienna History Wiki’s structured facts, addresses, awards, tenures, do not hang directly off an entity but off an intermediate Semantic MediaWiki *subobject*. This anonymous node only identifiable by a hash fragment in its URI that bundles several related properties into one record. Verbalising a subobject’s triples one at a time, in isolation, would throw away the information that makes the record meaningful. For instance a `startDate` triple with no indication of which office it belongs to. The two-pass design exists to recover that context before anything is written out.

Pass 1 scans every triple in the file but only acts on the ones whose subject is a subobject URI, detected by matching the pattern `-23[0-9a-fA-F_]{4,}$` against the final path segment of the URI. For each such triple, the predicate is simplified to a readable label and checked against a short skip list of Semantic MediaWiki predicates. Surviving predicates are accumulated into a per-subobject context dictionary keyed in the subobject’s URI. The `masterPage`, which names the entity the subobject belongs to, is stored once per subobject together with the master entity’s own URI if the object itself is a URI. Every other predicate is stored as a *list* of values instead of a single value because a subobject property can legitimately repeat. For example, two `streetAddress` entries for a person who practised at more than one location, and keeping only the first occurrence would silently drop the second from ever appearing as context for the subobject’s other facts.

Pass 2 scans the file a second time and emits exactly one output record per input triple. For a triple whose subject is *not* a subobject, verbalisation is direct, the subject, predicate, and object are each simplified and joined into a single-line string of the form *Subject Predicate Object*. For a triple whose subject is a subobject, the script first looks up that subobject’s Pass 1 context. The display name of the record is the `masterPage` value if one was recorded, falling back to the subobject’s own simplified URI otherwise. The script then assembles a sibling context where every other predicate stored for that subobject is included, and if the current triple’s own predicate repeats

elsewhere in the context, only the *other* value or values are included, so a fact is never listed as its own sibling. When sibling context exists and the current predicate is not the `masterPage`, the verbalised text takes the form *Master [siblingPredicate value, value; otherPredicate value] predicate value*. When there is none, it falls back to the plain format. Every output record, subobject or not, carries eight fields which consists of the raw subject URI, the raw master URI (empty for non-subobject triples), a simplified subject label, the raw predicate URI, a simplified predicate label, the raw object URI when the object is itself an entity, the simplified object value, and the full verbalised text string that will later be embedded. The raw URI fields are preserved specifically because the two GraphRAG variants need them at runtime to issue SPARQL neighbourhood queries.

Both passes rely on a shared pair of helper functions. `simplify_uri` turns a raw URI into a human-readable label: it takes the fragment after `#` if one exists or the final path segment otherwise, percent-decodes the result, and resolves the wiki’s custom character escapes (`-3A→:`, `-28→(`, `-29→)`, `-2D→-`, `-2F→/`, `-22→"`, `-2C→,`, `-23→#`, `-26→&`, `-27→'`, `-21→!`, `-2B→+`), before stripping `Property:/Category:` namespace prefixes, replacing underscores with spaces, and finally removing any trailing SMW hash suffix of sixteen or more hexadecimal characters that survived the earlier steps. `simplify_object` handles the object side. URI objects are passed through `simplify_uri`, while literal objects are unescaped and then tested numerically. A numeric value above one million with a fractional part is treated as a Julian Day Number and converted to a calendar date of the form *6 February 1918* using a standard Julian-Day-to-Gregorian-date algorithm. A numeric value above one million without a fractional part is rendered as a plain integer, and a smaller whole-valued number is also rendered without its decimal point. One exception is that if the original text began with a leading zero, in which case it is left untouched so that zero-padded codes and identifiers are not corrupted into shorter numbers. Any literal that does not parse as a number and any decimal value below the one-million threshold is passed through unchanged. At the end of a full run, the script reports the total input lines, the number of records written, the number skipped, and explicitly confirms whether every skipped line was accounted for by the expected blank or comment case. The complete procedure is provided in Figure 20 in Appendix D.4.

6.2.4 Triple Embedding and Qdrant Index Construction

After the verbalisation process, the textual representation from each triple is embedded using Granite-Embedding-278M-Multilingual, which is the same

model that is used on other vector-retrieval components. L2 normalisation is applied in encoding time, resulting in a 768-dimensional vector that is stored with the related metadata in one Qdrant collection. This collection is configured with the HNSW. The payload field that is brought by each point with its examples are detailed in the table below.

Table 7: Qdrant Payload Fields and Examples for the Triple Collection

Field	Description	Example value
<code>subject_uri</code>	Raw URI of the triple’s subject	<code>.../Günter_Kenesei</code>
<code>master_uri</code>	Raw URI of the subject’s master entity; empty for non-subobject triples	<code>""</code>
<code>subject_label</code>	Simplified label of the subject	<code>Günter Kenesei</code>
<code>predicate_uri</code>	Raw URI of the triple’s predicate	<code>.../Property-3A-Parteizugehörigkeit</code>
<code>object_value</code>	Simplified object label or literal value	<code>Österreichische Volkspartei</code>
<code>object_uri</code>	Raw URI of the object; empty if the object is a literal	<code>""</code>
<code>text</code>	Full verbalised string used as the context line in prompts	<code>Günter Kenesei Parteizugehörigkeit Österreichische Volkspartei</code>

URIs are abbreviated; the full prefix is

`http://www.geschichtewiki.wien.gv.at/Special:URIResolver/`.

Upload proceeds in batches of 256 points. The uploader supports resuming, so if the collection already exists, it counts its current point total and skips rows that have already been uploaded. This makes it possible to recover from an interrupted upload without re-processing triples from the beginning.

6.2.5 Shared Runtime Components

The process that has been explained is run offline once to produce SPARQL graph and verbalised triple collection. Instead, the mechanism in this section runs on every query because it depends on the question being asked and therefore cannot be prepared in advance. They are described together because the three knowledge-graph-based architectures share them, so the differences between the architectures come from how the graph is traversed instead of from how the query is handled around it. All three of them use the query embedding procedure, seed retrieval, token budget for generation, and the same generation prompt. Additionally, for the two GraphRAG variants, they use runtime verbalisation, URI canonicalisation, cosine-reranking towards the SPARQL result, and deduplication during the neighbourhood expansion.

Query Embedding and Seed Retrieval. The submitted query is encoded once using the Granite-Embedding-278M-Multilingual with L2 normalisation that is applied in the same way as in the indexing process which produces 768-dimensional vector. This single vector is reused for every ranking operation conducted in the pipeline, including ranking towards SPARQL results for every expansion hop to allow all candidates to be scored against the same representation of the question. This vector is then sent to the verbalised triple collection in Qdrant which will send back a number of seed hits according to the configuration in decreasing order of cosine similarity. Every hit brings the same field payload that was presented in the Table 7. The `subject_uri` and `object_uri` are the ones that allow for graph expansion to be done since both of them keep the raw RDF identifier that is needed for SPARQL query. KG-Derived Vector RAG also keeps both fields in the payload but never queries further.

Runtime Verbalisation. A local helper turns each raw SPARQL returned triple into a single display line at query time in the form of subject, predicate, object. Both the subject and object if is itself a URI are canonicalised and then simplified with the same URI simplification logic used for the offline verbalisation in previous subsection. Literal objects are passed through unchanged as they come back from QLever. The helper produces one flat line per triple.

Canonicalisation removes Semantic MediaWiki subobject suffixes in the form of `-23_...`, for example `.../Karl_Seitz-23_a1b2...`. As a result, the parent entity can be recovered for display. This normalisation is only applied to triples that are returned by QLever, and never to the seed URI before a neighbourhood query is issued. The query must target a URI that is exactly the same as retrieved, since changing subobject URI with the parent URI can pick the neighbourhood belonging to the parent entity, not to the entities that are connected with the subobject itself.

Neighbourhood Ranking. A single SPARQL expansion returns triples that are far more than what can be kept by the per hop limit. Consequently, the pipeline needs a method to choose which ones to pass. Without this, the selection will only be based on the random ordering that is returned by QLever. Therefore, each returned triple is scored by how closely its verbalised text matches the question, with the highest-scoring triple filling the available slots. The batch’s verbalised texts are encoded in one call with embeddings normalised, and the score is obtained through the product of matrix–vector to the query vector. The ranking itself is done locally in every resulting batch

and not globally with the whole candidates. As a result, every seed has its own limit, and one entity with a dense connection cannot fill the budget on its own.

Deduplication. There are two forms of deduplication that happens during the expansion process. First, the pipeline records the subject-object URI pair that is connected with every neighbourhood query. If the same pair has been expanded for the seed hit before, the SPARQL query will not be fired again. Second, every candidate fact is identified through its raw subject-predicate-object tuple. A candidate whose identity has already been selected before is skipped and will not consume one of the available selection spots.

The selection procedure continues through the ranked candidate list until either the configured per-hop limit has been reached or no further candidates remain. Consequently, duplicate candidates are ignored without reducing the number of distinct facts that can be selected. In GraphRAG 2-hop, the same set of selected triple identities is shared between the first and second hops. This prevents a fact selected during the first expansion from being inserted again during the second.

Token-Budget Enforcement. The evidence that can be inputted into the generation prompt is limited to up to 6,000 tokens. The total number of tokens is computed using `tiktoken`. The candidate context block is inputted greedily one by one to the generation prompt as long as there are still remaining tokens. The process stops when adding the next tuple would exceed the remaining budget. In the script, a context line is either included in full or excluded and never cut in half.

The 6,000-token limit applies only to retrieved evidence instead of to the complete prompt. In other words, the fixed system instructions and the question are not counted for this accounting. For KG-Derived Vector RAG, this evidence consists of the retrieved verbalised triples. For GraphRAG 1-hop and GraphRAG 2-hop, the budget is shared between the seed-hit context and the neighbourhood evidence.

Prompt Construction. The three KG-Derived architectures use the same generation prompt template and only differ in the evidence that is inputted into the context. Aside from the same principles that have been explained in Subsection [5.4.4](#), there are several additional instructions that are related to the characteristic of RDF data and benchmark questions. One is that the answer to the question that requires aggregation is limited only to

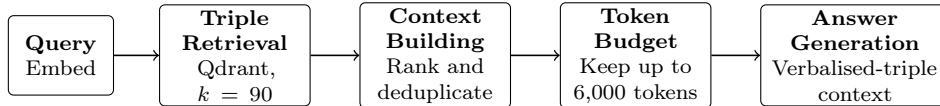
the items that actually appear in the retrieved context. Second, additional attention is given towards dates. Third, **Titel** property is separated between **jobTitle** and **Funktionsanzeige**. Lastly, the answer to descriptive questions is limited to the fields that are explicitly asked by the question. These instructions do not appear in the document baseline prompt because these instructions are related to the characteristics of the verbalised triple format. This does not affect the retrieval performance in an unjust way since these are not related to the question answering itself. The complete prompt template is provided in Appendix [F](#). At query time, the **{query}** is replaced with the actual question while the **{context}** is replaced with the retrieved context by the chosen architecture.

6.2.6 End-to-End KG-Derived Vector RAG

The verbalised triple RAG pipeline treats the knowledge graph as a flat collection of embedded facts with no graph traversal. The pipeline is structurally identical to the vector RAG baseline but retrieves from the Qdrant collection of verbalised triples instead of the collection of text chunks. Since it never expands beyond its seed hits, it uses only a subset of the shared runtime components. It retrieves verbalised triples and enforces the token budget, but has no need for the SPARQL neighbourhood queries, runtime verbalisation, or cross-hop deduplication that the one-hop and two-hop versions rely on. Since an individual triple carries far less information compared to a paragraph-length chunk, a larger retrieval width is used which is $\text{top_k} = 90$. This is around 6 times the vector RAG baseline so that the retrieved context still contains enough material to answer a question even though each unit of context is much smaller.

Given a natural language question from the user and the seed count of 90 from Qdrant, the pipeline returns an answer string together with the retrieved hits. Figure [8](#) summarises the end-to-end query-time pipeline. The detailed procedure is provided in Figure [23](#) in Appendix [G](#).

Figure 8: End-to-End Pipeline of KG-Derived Vector RAG



The first stage is the question embedding and seed retrieval. Both of them work the same way as in the subsection [6.2.5](#). The query vector is sent to the verbalised triple collection in Qdrant, then the top 90 hits are returned in

descending order of similarity score. Each hit contains the cosine similarity score, verbalised text, and metadata fields from the payload.

The second stage is the context building. The retrieved hits are formatted into a block of context one by one according to the ranking from Qdrant. Every block starts with a header that contains the rank, similarity score, subject label from that hit which was taken from the `subject_label` field, and the verbalised text. Every time a hit's block is formatted, its token count is added to a running total. The moment that the next block would push that total pass the limit, no further hits are added.

The last stage is the generation stage. The completed prompt is sent to the active provider generator. Then, the resulting text is processed further by removing the Markdown bold markers before returned as the final answer.

As an example, for the question "When was Johann Strauss (the son) born?". Among the 90 retrieved triples, the highest-ranked triples contain birth dates of three different persons who share the same name, namely Johann Strauss (Vater), Johann Strauss (Sohn), and Johann Strauss (Enkel), as shown in the figure below. The triple that answers the question is ranked second, and the same fact appears again in a different date format at rank six.

Figure 9: KG-Derived Vector RAG: Top Six Retrieved Triples for the Example Question

```
[1] Johann Strauss (Vater) birthDate 1804-03-14
[2] Johann Strauss (Sohn) birthDate 1825-10-25
[3] Johann Strauss (Vater) birthDate 14 March 1804
[4] Johann Strauss (Enkel) birthDate 1866-02-16
[5] Johann Strauss (Enkel) birthDate 16 February 1866
[6] Johann Strauss (Sohn) birthDate 25 October 1825
```

Since each triple states its subject explicitly, the generator can separate the three people, and the generator answered "Johann Strauss (Sohn) was born on 25 October 1825."

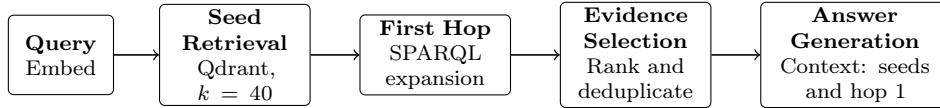
6.2.7 End-to-End GraphRAG 1-Hop

The GraphRAG 1-Hop is the first version that uses the graph as a connected structure instead of just a flat collection of facts. It keeps the seed retrieval of the verbalised triple RAG but adds a single round of SPARQL neighbourhood expansion around each seed. Hence, the evidence reaching the generator includes not only the retrieved triples but also the facts directly linked to

them. Therefore, this pipeline uses the full set of shared runtime components explained in Section 6.2.5. On the construction side, it reuses both prepared resources, retrieving seeds from the Qdrant triple collection and expanding them against the QLever SPARQL graph.

As mentioned in the methodology, GraphRAG 1-hop retrieves the top 40 seed triples from Qdrant. After ranking and deduplication, at most five new triples are retained per seed hit. The combined seed and neighbourhood evidence is constrained to the shared 6,000 token budget. The algorithm summary can be seen in Figure 13 with the complete procedure detailed in the following steps.

Figure 10: End-to-End Pipeline of GraphRAG 1-Hop



Once a query is submitted, the first stage is question embedding and seed retrieval. Both of these run the same way as explained in Section 6.2.5. The query vector returns the top 40 seed triples from the verbalised triple collection in Qdrant. Aside from the similarity score and the verbalised text, each hit also keeps the `subject_uri`, `predicate_uri`, `object_value`, and `object_uri` in their raw forms. These subject and object URIs provide the entry point to the graph for the expansion process.

The second stage is for the neighbourhood expansion. For each seed hit whose (subject URI, object URI) pair has not already been queried in this pass, the pipeline issues a SPARQL query against the QLever endpoint at localhost. When the seed’s object is itself a URI, the query uses a four-branch UNION, one branch each for edges incoming to the subject, outgoing from the subject, incoming to the object, and outgoing from the object. The complete SPARQL query used for this expansion is provided in Appendix E.1.

Each branch carries its own LIMIT instead of one shared limit over the whole UNION. A single shared limit would let QLever fill the entire cap from whichever branch it happens to evaluate first, usually the subject’s incoming edges, which silently starves the other three. When the seed’s object is a literal instead of URI, the last two branches are omitted and the query collapses to the two branch from around the subject alone. The subject and object URIs are used exactly as retrieved from Qdrant with no canonicalisation applied beforehand. The pattern is symmetric because the graph carries useful information on both sides of most relationships. For example, an architect and the building they designed, a politician and the

office they held, and a query that only followed outgoing edges from the subject would miss half of what is available.

Every triple QLever returns for a seed hit is verbalised at runtime, and the whole batch is ranked by cosine similarity against the question vector in one batched encode call. The pipeline then walks through that ranked list from the top and applies two levels of deduplication as explained in Section 6.2.5. Collection stops once `hop1_limit` new, non-duplicate triples have been taken for that seed, or the ranked list is exhausted. Finally, triples that share a Semantic MediaWiki subobject are merged into one bundled display line for the generator.

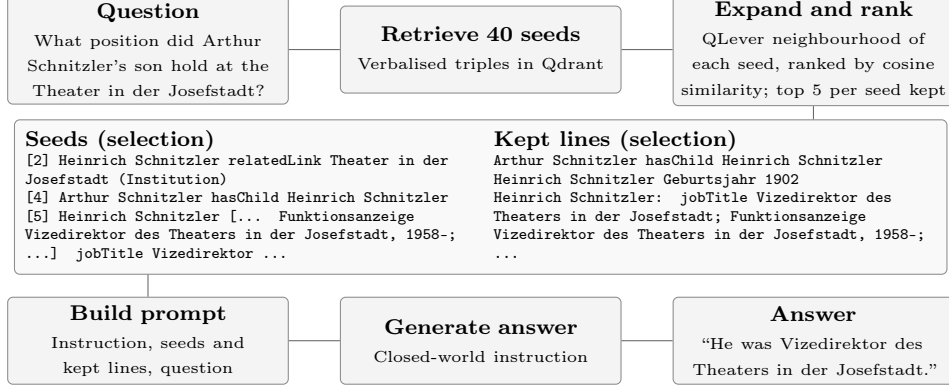
The third stage is the token budget enforcement. The seed hit context block is measured first. This comprises only the formatted hits without any system instruction, question text, or neighbourhood section. Then, its token count is subtracted from the 6,000 token budget to decide how much remains for the neighbourhood. Afterwards, the neighbourhood lines are admitted one at a time in ranked order until that remainder is exhausted. The fixed system prompt and question are not accounted for at all.

The final stage is the prompt assembly and generation. The final prompt combines the system instruction, question, seed hit block with every hit given the label of similarity score and a simplified subject label, and the neighbourhood lines. The set prompt is sent to the generator, which then processes the response further to remove the Markdown bold markers.

This pipeline can be accessed through the `ask(query, top_k, hop1_limit)`. The neighbourhood text itself is returned either under `neighbourhood_text` or `h1_text`.

As an example, consider the multi-hop question “What position did Arthur Schnitzler’s son hold at the Theater in der Josefstadt?”. The son is not named in the question and has to be identified first. As shown in Figure 11, the seed triple *Arthur Schnitzler hasChild Heinrich Schnitzler* at rank 4 provides this link, but does not state which position he held. Each seed is then expanded in QLever: the returned triples are verbalised, scored by their cosine similarity to the question, and the five highest-scoring triples per seed that have not yet been selected are kept, with triples of the same subobject merged into one line. Among the kept lines is the function record of Heinrich Schnitzler as Vizedirektor of the theatre, which also appears among the seeds at rank 5.

Figure 11: GraphRAG 1-Hop: Processing of the Example Question



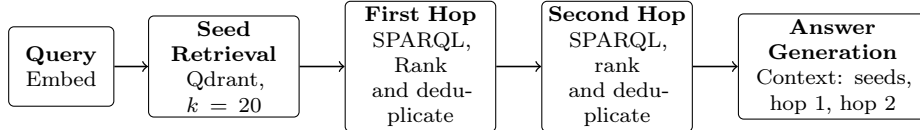
Based on this evidence, GPT-5.4 answered “He was Vizedirektor des Theaters in der Josefstadt.”

6.2.8 End-to-End GraphRAG 2-Hop

GraphRAG 2-hop extends the 1-hop architecture with a second round of SPARQL neighbourhood expansion. The first stage is identical in principle to GraphRAG 1-hop where verbalised triples are retrieved from Qdrant and used as seed points for neighbourhood expansion in QLever. The difference is that entities discovered among the selected first-hop triples can themselves become entry points for a second expansion round. This allows the pipeline to retrieve evidence that is separated from the initial seed by an additional graph edge. Therefore, the second-hop frontier is determined by the evidence discovered during the first hop rather than directly from the original question.

Each expansion round introduces significantly more contexts, which makes this configuration have lower seed triples when compared to the GraphRAG 1-hop. This pipeline retrieves the top 20 seed triples from Qdrant. From here, the top five triples that have been ranked and deduplicated are kept for each seed in the first hop and the five per frontier URI in the second hop. Evidence from the seed, first, and second hops together shares the same token cap of 6,000 tokens. Figure 12 shows the complete data flow, and the procedure is detailed in the steps below.

Figure 12: End-to-End Pipeline of GraphRAG 2-hop



Once a query is sent, as usual, the first step is question embedding and seed retrieval. Both work in the same way as described in Section 6.2.5, with the query vector returning the top 20 seed triples, and each hit keeping its raw `subject_uri`, `predicate_uri`, `object_value`, and `object_uri`.

The second step is the first hop expansion, which follows the same procedure as GraphRAG 1-hop. Each seed is expanded through the four-branch SPARQL query, or the two branch if the object is a literal. The triples returned are then verbalised, ranked, and deduplicated before up to five triples are kept per seed. Both the set of admitted first-hop triples and the set of already-queried entity pairs are forwarded to the second round.

The next step is the frontier construction. Before the second expansion begins, entity URIs already encountered through the seed triples are marked as known. Both their raw and canonicalised forms, that is, with and without the subobject suffix, are considered so that the same entity is not expanded again only because it occurs in two URI representations. Then, the pipeline examines the chosen first-hop triples and collects the previously unseen URI-valued subjects and objects. These entities are the ones that create the second hop frontier. For each frontier URI, the first-hop triple that discovered it is also recorded so that the traversal path is preserved in the final context.

As for the fourth step, each frontier URI is expanded with a two-branch SPARQL query that retrieves the edges incoming to and outgoing from that URI. The SPARQL query is provided in Appendix E.2. Unlike the first hop, there is no object side to expand since a frontier entry is a single entity instead of a subject-object pair. The returned triples are verbalised, ranked by cosine similarity to the question within their own batch, and admitted one at a time, up to five new triples per frontier URI. Importantly, the duplicate check at this stage uses the same raw-identity set that is used by the first hop. Hence, a fact that is already admitted while expanding a seed cannot be re-admitted.

The fifth step is the token budget enforcement, which happens in three steps. The seed hit context is measured first, then the total number of tokens is subtracted from the 6,000 token limit. The remaining spots are filled with the first hop lines, one by one, as long as it is still possible. The remaining tokens would be used by the second hop lines. Similar to other architectures, a context is only filled when it fits completely within the remaining budget, while system instructions and questions are outside of this computation.

The final stage is the prompt assembly and generation. The final context includes the retrieved seed triples, the first-hop, and second-hop evidence that fits within the token budget. This evidence, along with the question, is input into the same prompt template used in GraphRAG 1-hop, then sent to the chosen provider.

The pipeline is exposed through `ask(query, top_k, hop1_limit,`

hop2_limit). The algorithm in Figure 13 summarises the two-round procedure. Here, v is the query embedding, H_1 and H_2 are the triples selected in each hop, L_1 and L_2 their grouped display lines, b the remaining token budget.

Figure 13: Algorithm of Graph RAG 1 and 2 Hop

```

1:  $v \leftarrow \text{EMBEDQUERY}(q)$ 
2:  $hits \leftarrow \text{RETRIEVE}(q, k)$  ▷ top- $k$  Qdrant seed triples
3:  $seenUris \leftarrow \emptyset$ ;  $seenIds \leftarrow \emptyset$ ;  $H_1 \leftarrow []$ 
4: for all  $h \in hits$  do ▷ first hop
5:    $pair \leftarrow (h.subject, h.object)$ 
6:   if  $h.subject \neq ""$  and  $pair \notin seenUris$  then
7:     add  $pair$  to  $seenUris$ 
8:      $C \leftarrow \text{SPARQLNEIGHBOURHOOD}(h.subject, h.object)$  ▷ 4-branch
9:      $C \leftarrow \text{VERBALISE}(C)$ ; rank  $C$  by cos with  $v$ 
10:    take up to hop1_limit new triples into  $H_1$  (skip if raw s,p,o  $\in seenIds$ , without
        consuming a slot; update  $seenIds$ )
11:   end if
12: end for
13: if  $H_1 = \emptyset$  then return answer from  $hits$  alone
14: end if
15:  $known \leftarrow \{u, \text{CANONICAL}(u) : u \in seenUris\}$ 
16:  $frontier \leftarrow []$ 
17: for all  $t \in H_1$  do ▷ build second-hop frontier
18:   for all  $u \in \{t.S, t.O \text{ if } t.O \text{ is a URI}\}$  do
19:     if  $u \notin known$  and  $u \notin frontier$  then
20:       append  $u$  to  $frontier$ ; record  $\text{parent}(u) \leftarrow t$ 
21:     end if
22:   end for
23: end for
24:  $H_2 \leftarrow []$ 
25: for all  $u \in frontier$  do ▷ second hop
26:    $C \leftarrow \text{SPARQLNEIGHBOURHOOD}(u)$  ▷ two-branch
27:    $C \leftarrow \text{VERBALISE}(C)$ ; rank  $C$  by cos with  $v$ 
28:   take up to hop2_limit new triples into  $H_2$  (checked against shared  $seenIds$ ), each
        tagged with  $\text{parent}(u)$ 
29: end for
30:  $L_1 \leftarrow \text{GROUP}(H_1)$ ;  $L_2 \leftarrow \text{GROUPWITHPATH}(H_2)$  ▷ hop-2 lines prefixed parent -> fact
31:  $b \leftarrow 6000 - \text{TOKENS}(\text{CONTEXT}(hits))$ 
32:  $kept_1 \leftarrow \text{GREEDYFIT}(L_1, b)$ ;  $kept_2 \leftarrow \text{GREEDYFIT}(L_2, b - \text{TOKENS}(kept_1))$ 
33:  $prompt \leftarrow \text{BUILDPROMPT}(q, hits, kept_1 || kept_2)$ 
34: return  $\text{GENERATE}(prompt)$ 

```

An example is the description question, namely “What is the Konzerthaus? State only: what type of structure it is, the district where it is located, its

street addresses, and its architects.” The seeds found the type of structure with the street addresses, while the first hop found the district and first architect. From the relations found in the first hop, the second hop found the names of architects. Using this context, the generator listed the architects as “Ferdinand Fellner; Hermann Helmer”, which are the name forms used in the caption.

6.3 Agentic Graph RAG

This section describes the third and most graph-intensive system implemented for this thesis, which is an agentic pipeline based on Think-on-Graph (ToG) [32]. Instead of retrieving once, the language model repeatedly decides which part of the Vienna History Wiki graph to explore next which builds an explicit reasoning paths step by step. In the taxonomy of graph retrieval-augmented generation, this means an iterative retrieval where multiple retrieval steps are used and each search depends on the results of the previous one [26]. The implementation is based on ToG version-1 but it is adapted to the Semantic MediaWiki data model, the German-language property space, and the closed-world evaluation setting of this thesis. The section walks through each stage of the adapted pipeline, from topic entity extraction through relation and entity exploration to path bookkeeping and termination.

6.3.1 System Overview

The adapted pipeline, referred to as Agentic Graph RAG in this thesis, is implemented in four Python modules. A client module which consists of all SPARQL access to the cleaned Vienna History Wiki graph served by the local Qlever endpoint at <https://localhost:7001>. A function module which contains the language-model interface and the search, prune, scoring, and answer-generation logic. A prompt module which holds five domain-specific few-shot prompts, and a driver which implements the depth loop and the path bookkeeping. All the language-model calls go to LLM, the same generator used by the other four systems through a provider abstraction that can also route to alternative model families. This follows the plug-and-play property claimed for the original framework [32]. Every call carries a fixed system message instructing the model to act as a research assistant for the cultural heritage and to answer only from the relations, candidates, or triplets provided in that step, never from its own knowledge.

The prompt module holds five domain-specific few-shot prompts, one for each decision the language model makes during traversal, namely topic entity extraction, relation pruning, entity scoring, sufficiency evaluation, and answer

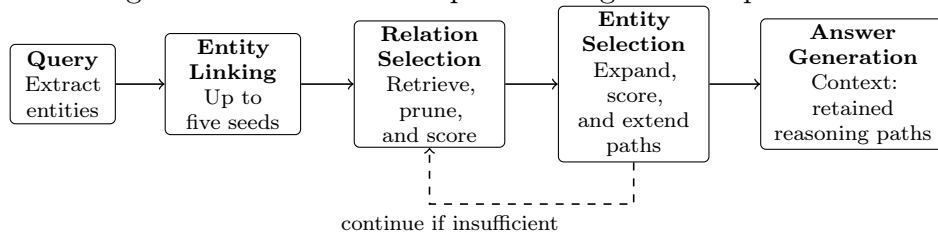
generation. Each pairs a fixed instruction with examples from the wiki. Each of the prompt is described in the subsection covering the stage it belongs to.

The configuration follows the retrieval settings established in the previous section, namely a beam width of 20, a maximum depth of 3, and up to 5 linked seed entities, along with the shared 6,000 token budget on the serialised paths. Two settings are specific to this architecture. Relation pruning and entity scoring run at a temperature of 0.4, which allows some variation in the candidate rankings. All SPARQL access goes to the local QLever endpoint, providing the cleaned Vienna History Wiki graph.

The beam width that is used in this thesis is 20. This is wider than the width limit of 3 that is used in the original experiment [32]. The reason lies in the graph. Unlike Freebase, this graph mixes schema.org properties with the German Semantic MediaWiki properties. Consequently, one question that is in English can translate to several names in German. As an example, **birthDate**, **Geburtsjahr**, **Geburtsmonat**, and **Geburtstag** all encode birth information. If the beam is too narrow, the property that contains the answer can be cut even before it gets explored. Aside from this, there are two additional formats for each provider. In Qwen, the internal thinking mode is disabled, and in GPT-5.4, the reasoning effort is set to none.

The Figure 14 below shows the data flow when a query is submitted. Unlike the static architectures, the middle three stages keep repeating until the model judges that the collected evidence is sufficient or the maximum depth is reached.

Figure 14: End-to-End Pipeline of Agentic Graph RAG



6.3.2 Topic Entity Extraction and Linking

This pipeline starts by asking the language model to extract the topic entity from the question and localise it on the knowledge graph. A few-shot prompt asks the language model to recognise the named entities and return German entity names where available. This prompt also contains the domain-specific instruction for selecting the starting entity. As for questions regarding a place where someone lived, worked, or practised, the person themselves is used as

the main anchor because the address information stays on the person’s page. On the other hand, a building is used as an anchor if there is a question about the architect or location of a building submitted. This prompt also supports the disambiguated Vienna History Wiki names such as *Theodor Körner (Politiker)*, *Johann Strauss (Sohn)*, and *Johann Strauss (Vater)*.

Linking the extracted names to URIs is done without any full-text index, purely through the URI conventions of GeschichteWiki namespace. The client generates a family of candidates from each name. For example, spaces become underscores, parentheses are encoded -28 and -29, hyphens in compound names such as Karl-Marx-Hof as -2D, and non-ASCII characters are tried both percent-encoded and raw since the store is inconsistent in this respect. Candidate URIs are tested through small SPARQL existence queries. If a probe succeeds, `owl:sameAs` redirect nodes are resolved so that, for example, *Secession* lands on *Secession (Institution)*. If the name alone does not resolve and it carries no qualifier, the client additionally probes a fixed list of fifteen common disambiguation suffixes (*Politiker*, *Architekt*, *Komponist*, and so on) so that homonymous pages are collected as separate seeds. Only as a last resort does the client fall back to a substring search over URIs, which is the one expensive query in the stage. Up to five seed entities are retained per question. If no entity can be linked at all, the system does not guess. Instead, it will immediately return the closed-world fallback.

6.3.3 Relation Exploration

Traversal continues iteratively from the current frontier entities. At each depth, the graph client retrieves both outgoing and incoming predicates for every frontier entity. An outgoing predicate represents a relation in which the frontier entity occurs as the subject, while an incoming predicate represents a relation in which it occurs as the object. This follows the bidirectional relation-search procedure used by the original ToG framework [32]. Without the incoming direction, a question about who designed a building would fail whenever the relation is stored from the architect’s side.

The technical predicates are removed first using the same block list that was applied in the RDF pre-processing. This removal is to keep the system-level Semantic MediaWiki bookkeeping from competing with the historical content. The remaining relation URI is simplified to a readable label. The same relation label that appears from different directions is differentiated by the suffix `[incoming]` or `[outgoing]` to ensure that the model’s choice maps back to exactly one URI and direction. Relations that have been used to expand an entity in the previous depth are also removed to prevent the beam from exploring the same edge.

In the next step, the resulting relation list is passed to the relation-pruning prompt that asks the language model to choose and score up to the configured beam width of 20 relations. This prompt also contains a domain-specific instruction that maps the natural language concept to the property that is used in the cultural historic wiki. The returned selections are validated against the available relation list, deduplicated by URI and direction, clamped to valid values, renormalised, and restricted to the beam width. As a result, every relation that goes to the next step is guaranteed to correspond to an edge that exists in the graph.

6.3.4 Entity Exploration

Each selected relation is then expanded through SPARQL. For an outgoing relation, the system queries $(e, r, ?)$, while an incoming relation uses $(?, r, e)$. Each directional query is limited by the configured neighbourhood limit of 100 results. The returned candidates can be ordinary entity URIs, literal values, or Semantic MediaWiki subobject nodes.

To make the subobject URI that was originally unreadable into a readable format during candidate scoring, the client constructs a preview label. A simple SPARQL query reads the subobject’s semantic type with the useful internal property name without showing the values. Technical predicates, noise fields, and `masterPage` relations are not included in the preview. As a result, a candidate can be shown, for example, postal address subobject containing the properties *streetAddress*, *address*, and *Bezirk*. This shows the model the type of information that is stored in the node without revealing the answer one hop early. If the subobject is kept, it will be the frontier entity, and the real triple’s value can be explored in the next iteration. This design is important because Semantic MediaWiki subobjects are spread out and not directly connected to the parent entity, which makes their relevance non-obvious during traversal. Withholding the values also keeps the comparison fair since the static GraphRAG versions expose full subobject values only once a subobject has actually been retrieved instead of while deciding whether to retrieve it.

Before the candidates are scored, literal values are normalised into a form that can be interpreted more consistently by the language model. Large non-integer numerical literals representing Julian Day Numbers are converted into readable calendar dates. Integer-valued literals are displayed without unnecessary decimal components, while placeholder values such as *Unknown* and *N/A* are removed.

The cleaned candidates of each relation are then passed to the scoring prompt which divide the relevancy score among those candidates based on the

question and relation label. Candidates that do not contribute in answering the question are scored zero. The whole candidates that have been scored from all relations in that specific depth is then combined, sorted, and cut to the global top beam width with candidates scoring zero removed.

The surviving candidates are converted into directed graph triples according to the direction of their respective relations. URI-valued candidates, including selected Semantic MediaWiki subobjects, can form the frontier for the following traversal depth, while literal-valued candidates terminate their respective graph branches. These surviving triples are then passed to the path management stage, where they extend the accumulated reasoning paths and are used to determine whether further traversal is required.

6.3.5 Path Bookkeeping, Reasoning, and Termination

For every discovered entity, the driver keeps the set of paths that lead from a seed entity to it together with a pool of finished leaf paths whose endpoints are literals or previously visited entities. A visited set over URIs prevents cycles, which means an expansion that arrives back at a known entity closes its path as a leaf instead of re-opening the entity as a frontier. Subobject nodes also participate in this bookkeeping through their complete hash URIs and can therefore be expanded once as distinct entities. At any moment, the effective evidence is the union of all leaf paths and all paths of not-yet-expanded entities, which keeps intermediate hops visible to the reasoning model without duplicating them.

After every depth that results in surviving triples, the resulting paths are serialised and passed to a sufficiency-evaluation prompt. This step corresponds to the reasoning stage in the original algorithm, where the model decides whether the retrieved evidence is sufficient to answer the question or not [32]. The prompt is adapted in one important way for the benchmark of this thesis. For aggregation questions that ask *how many* or request a complete list, the model is instructed to answer *yes* only when the paths plausibly contain the full set of matching facts. This is because a partial set of siblings or awards would otherwise terminate the search early and produce a wrong count confidently. Before both evaluation and final generation calls, the chain that has been serialised first needs to pass the token-budget guard, which keeps the evidence within the 6,000 token cap.

Exploration terminates in one of three ways. If the model judges the accumulated evidence to be sufficient, the answer is generated immediately. If a depth produces no usable relations or candidates, if every combined candidate scores zero, or if the frontier empties because every new entity has already been visited, the system performs a half stop and generates from

whatever paths exist. If the maximum depth is reached, generation likewise proceeds from the collected paths. The differences between this adaptation and the original algorithm are summarised in Appendix [H](#).

The final generation prompt instructs the active language model to answer only from the retained reasoning paths. If no seed entity can be linked or no relevant reasoning path can be constructed, the pipeline returns the closed-world fallback “*Not available in the indexed data.*”

The complete algorithm can be found in Figure [15](#), with E_0 is the set of linked seed entities, R is the selected relations at depth d , R_e the relations of entity e , and C the scored candidate entities.

Figure 15: Algorithm of the Agentic Graph RAG(q)

```

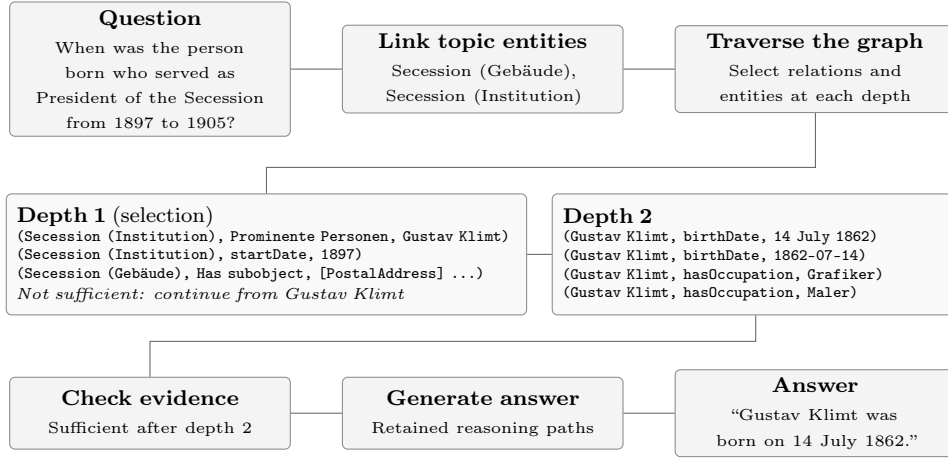
1:  $names \leftarrow \text{EXTRACTENTITIES}(q)$ 
2:  $E_0 \leftarrow \text{LINKENTITIES}(names, k=5)$ 
3: if  $E_0 = \emptyset$  then return “Not available in the indexed data.”
4: end if
5:  $paths \leftarrow$  empty path map anchored at  $E_0$ ;  $visited \leftarrow E_0$ 
6: for  $d \leftarrow 1$  to  $depth$  do
7:    $R \leftarrow \emptyset$ 
8:   for each entity  $e \in \text{frontier}$  do
9:      $R_e \leftarrow \text{SPARQLRELATIONS}(e)$ 
10:     $R_e \leftarrow \text{FILTERTECHNICAL}(R_e) \setminus \text{USEDRELATIONS}(e)$ 
11:     $R \leftarrow R \cup \text{LLMPRUNERELATIONS}(q, e, R_e, width)$ 
12:   end for
13:   if  $R = \emptyset$  then break
14:   end if
15:    $C \leftarrow \emptyset$ 
16:   for each  $(e, r) \in R$  do
17:      $cands \leftarrow \text{SPARQLEXPAND}(e, r)$ 
18:      $cands \leftarrow \text{PREVIEWSUBOBJECTRELATIONS}(cands)$ 
19:      $cands \leftarrow \text{NORMALISELITERALS}(cands)$ 
20:      $C \leftarrow C \cup \text{LLMSCOREENTITIES}(q, r, cands) \cdot \text{score}(r)$ 
21:   end for
22:   keep top- $width$  of  $C$ ; merge subobject groups; extend  $paths$ ; update  $visited$ 
23:    $chain \leftarrow \text{TRUNCATE}(paths, 6000)$ 
24:    $sufficient \leftarrow \text{LLMSUFFICIENT}(q, chain)$ 
25:   if  $sufficient$  then break
26:   end if
27: end for
28:  $chain \leftarrow \text{TRUNCATE}(paths, 6000)$ 
29: return  $\text{LLMGENERATE}(q, chain)$ 

```

As an example, consider the multi-hop question “When was the person born who served as President of the Secession from 1897 to 1905?”. As shown

in Figure 16, the question is linked to two entities, *Secession (Gebäude)* and *Secession (Institution)*. At the first depth, the traversal keeps triples from both, including the relation **Prominente Personen**, which leads from the institution to Gustav Klimt. Since this evidence is not yet sufficient, the traversal continues from Gustav Klimt at the second depth, where it finds his birth date.

Figure 16: Agentic Graph RAG: Processing of the Example Question



Based on this evidence, GPT-5.4 answered "Gustav Klimt was born on 14 July 1862.". The example also shows that the traversal can keep irrelevant triples next to the relevant path, such as the address subobject of the Secession building.

7 Results

This chapter provides a description of the evaluation results of the five systems across the three measurement layers explained in the previous chapter. The three layers are used to address different questions. The retrieval quality is using term recall to check if the evidence a question needs ever reached the generator, independent of what the generator then did with it. The RAG Assessment metrics act as a LLM judge to the generated answers, while the manual assessment is the manual statement-level evaluation of the generated answers.

7.1 Retrieval-Based Evaluation Results

Term Recall measures the proportion of gold evidence terms that appear in the retrieved context before final answer generation. Therefore, it evaluates retrieval independently of how the generation model subsequently uses the retrieved evidence. The metric is calculated for all 1,200 evaluated answers, covering five architectures, two generation providers, and 120 benchmark questions. Table 8 shows the term recall for each architecture under each generation provider and overall.

Table 8: Term Recall per Architecture and Generation Provider

Architecture	OpenAI	Qwen	Overall
Document Vector RAG	0.943	0.943	0.943
KG-Derived Vector RAG	0.893	0.893	0.893
GraphRAG (1-hop)	0.884	0.884	0.884
GraphRAG (2-hop)	0.830	0.830	0.830
Agentic Graph RAG	0.919	0.889	0.904

Among the static systems, Document Vector RAG achieves the highest at 0.943, Verbalised Triple RAG at 0.893, GraphRAG 1-hop at 0.884, and GraphRAG 2-hop at 0.830. Under the shared token budget, neighbourhood expansion competes with the initially retrieved evidence instead of adding to it. This effect is strongest for the two-hop configuration, since it begins from fewer seed triples to cover the additional expansion. For the four static architectures, the OpenAI and Qwen values are identical. Their retrieval stage is independent of the final generation provider and therefore produces the same retrieved context under both conditions. The Agentic Graph RAG differs because the selected provider participates directly in graph traversal. Its Term Recall is 0.919 with OpenAI and 0.889 with Qwen, corresponding to a difference of 0.030.

7.1.1 Results by Question Category

The overall results in Table 8 do not show the variation across question types, a single architecture may ground well on one kind of question and poorly on another. Therefore, the table below breaks term recall down by the four benchmark categories, combined across both providers. Each cell is the share of gold evidence terms retrieved for the 30 questions in that category, averaged over the two provider conditions.

Table 9: Term Recall per Architecture and Question Category with Combined LLM Providers

Architecture	single_val.	multi_hop	aggreg.	descript.
Document Vector RAG	0.950	0.878	0.961	0.983
KG-Derived Vector RAG	0.983	0.864	0.938	0.789
GraphRAG (1-hop)	0.996	0.829	0.961	0.749
GraphRAG (2-hop)	0.976	0.827	0.861	0.657
Agentic Graph RAG	0.914	0.809	0.967	0.926

Single-value questions achieve high Term Recall across all architectures, ranging from 0.914 for the Agentic Graph RAG to 0.996 for GraphRAG 1-hop. KG-Derived Vector RAG and GraphRAG 2-hop also exceed 0.97 in this category. This shows that the evidence for single-value questions can be retrieved successfully by all of the architectures. In other words, in this question category, the retrieval seldom the limiting factor.

Multi-hop is the category where every system scores the lowest. All five sit between 0.809 and 0.878. Document vector RAG is the highest at 0.878, followed by KG-Derived Vector RAG at 0.864, GraphRAG 1-hop at 0.829, GraphRAG 2-hop at 0.827, and the Agentic Graph RAG at 0.809.

For aggregation questions, four of the five systems sit between 0.938 and 0.967, with the Agentic Graph RAG being the highest. This is followed by Document Vector RAG and Graph RAG 1-hop both record at 0.961, and KG-Derived Vector RAG at 0.938. GraphRAG 2-hop is the only architecture below 0.9 in this category at 0.861. Although, it still retrieves the majority of the required evidence.

The largest differences can be seen in descriptive questions. Document Vector RAG reaches 0.983 and the Agentic Graph RAG reaches 0.926. The three static triple-based architectures record 0.789 for KG-Derived Vector RAG, 0.749 for GraphRAG 1-hop, and 0.657 for GraphRAG 2-hop. A document chunk can contain several attributes from the same entities while the triple-based system must retrieve each of those facts one by one within the same budget.

Overall, the retrieval of broad document coverage provides the highest evidence coverage. On the other hand, static graph expansion trades off retrieval coverage with relational neighbourhood evidence under the same budget. Whether lower retrieval coverage can still produce better final answers is determined by the subsequent evaluation layers.

7.2 RAG Assessment (RAGAs) Results

The second evaluation layer reports the four RAGAs metrics defined in Evaluation Methodology, namely, Faithfulness, Answer Relevancy, Context Recall, and Answer Correctness. These four metrics measure different aspects of retrieval and generation, which do not always rank the architectures in the same order from one metric to another. Consequently, these four are read together as diagnostic measure instead of interchangeable indicators of answer quality. The Table 10 shows the four metrics for every architecture, combined from the two generation providers. Each value is the mean judge score over the 240 answers each architecture produced, with 120 questions under each provider.

Table 10: Faithfulness, Answer Relevancy, Context Recall, and Answer Correctness per Architecture

Architecture	Faithful.	Answer rel.	Context rec.	Answer corr.
Document Vector RAG	0.916	0.618	0.881	0.711
KG-Derived Vector RAG	0.818	0.443	0.767	0.671
GraphRAG (1-hop)	0.816	0.459	0.730	0.658
GraphRAG (2-hop)	0.788	0.415	0.668	0.615
Agentic Graph RAG	0.892	0.786	0.894	0.811

Answer Correctness directly compares the generated answer to the gold answer. As can be seen from the table, the Agentic Graph RAG scores the highest at 0.811, followed by Document Vector RAG at 0.711, KG-Derived Vector RAG at 0.671, GraphRAG 1-hop at 0.658, and GraphRAG 2-hop at 0.615. Two observations can be made here. One, the fact that the document baseline places second means that access to the graph does not produce more correct answers than chunks. Second, the highest score being 0.811 means that even the strongest system leaves a share of answers missing or unsupported.

Context recall measures the proportion of reference claims supported by the retrieved context. The Agentic Graph RAG also records the highest Context Recall at 0.894. Document Vector RAG follows closely at 0.881. This means that around nine in ten gold statements are covered, which shows that retrieval is not the main source of the issue if the generated answer receives a lower score. The three static triple-based architectures record 0.767 for KG-Derived Vector RAG, 0.730 for GraphRAG 1-hop, and 0.668 for GraphRAG 2-hop. This is the same ordering as the term recall in the previous section. Since the two decline together, the pattern is consistent with expansion displacing supporting evidence instead of than adding to it under the fixed token budget.

Faithfulness measures whether the generated claims are supported by the retrieved context and not whether they match the gold answer. As can be seen from the table, Document Vector RAG achieves the highest score at 0.916, followed by the Agentic Graph RAG at 0.892. KG-Derived Vector RAG and GraphRAG 1-hop achieve similar values of 0.818 and 0.816, respectively, while GraphRAG 2-hop reaches 0.788. In practice, all five score well. A faithfulness of 0.916 means that roughly nine in ten of the answer’s claims are traceable to the retrieved context, and even the lowest system keeps about four in five of its claims grounded. Based on this metric, hallucination, in the strict sense of inventing content that is not present in the context, is uncommon across every architecture.

The difference between faithfulness and answer correctness is particularly informative for Document Vector RAG. Its faithfulness is 0.916 while its Answer Correctness is 0.711, showing a difference of 0.205, which is the largest gap observed. The corresponding differences are 0.147 for KG-Derived Vector RAG, 0.158 for GraphRAG 1-hop, 0.173 for GraphRAG 2-hop, and 0.081 for the Agentic Graph RAG. Faithfulness asks whether a claim is based on the retrieved context, while answer correctness asks whether it matches the gold answer. This difference means that Document Vector RAG answers based on the context but also provides answers that are not requested, which lowers the correctness score. On the other hand, the Agentic Graph RAG retrieves evidence chains built on one relation at a time. Therefore, the generator does not have a lot of information to generate answers with surrounding contexts that are not requested.

As for answer relevancy, the Agentic Graph RAG ranks first at 0.786, followed by Document Vector RAG at 0.618, GraphRAG 1-hop at 0.459, KG-Derived Vector RAG at 0.443, and GraphRAG 2-hop last at 0.415. From this ordering, the Agentic Graph RAG commits to the answer the most, while GraphRAG 2-hop refuses to answer the most. Looking at how the numbers are computed, they are held down by how the metric treats refusals. The metric returns zero whenever all three reconstructions flag an answer as noncommittal. With the closed-world instruction, this makes the fallback response more common. Consequently, refusals are scored the same as irrelevant answers. Therefore, a low relevancy score is a count of how often a system declines to answer and is not a measure of whether the answer is relevant to the question or not.

7.2.1 Effect of the Generation Provider

This table below splits the four metrics by generation provider. The retrieval configuration is identical in both columns for the four static systems. Hence,

any difference within a row is due to the generator and not because of what was retrieved.

Table 11: Automated RAG Assessment Metrics per Architecture and Generators

Architecture	Faithfulness		Answer rel.		Context recall		Answer corr.	
	OpenAI	Qwen	OpenAI	Qwen	OpenAI	Qwen	OpenAI	Qwen
Document Vector RAG	0.899	0.932	0.549	0.687	0.881	0.882	0.779	0.643
KG-Derived Vector RAG	0.806	0.831	0.355	0.531	0.766	0.768	0.717	0.625
GraphRAG (1-hop)	0.784	0.848	0.382	0.536	0.728	0.732	0.718	0.598
GraphRAG (2-hop)	0.763	0.813	0.333	0.497	0.672	0.664	0.667	0.564
Agentic Graph RAG	0.914	0.870	0.769	0.804	0.916	0.871	0.850	0.772

The provider-level results show a clear difference between the reference-free and reference-based metrics. For all four static architectures, Faithfulness is higher under Qwen than under OpenAI. Document Vector RAG increases from 0.899 to 0.932, KG-Derived Vector RAG from 0.806 to 0.831, GraphRAG 1-hop from 0.784 to 0.848, and GraphRAG 2-hop from 0.763 to 0.813. Answer Relevancy is also higher under Qwen for all five architectures.

Answer Correctness shows the opposite pattern. It is higher under OpenAI for every architecture. Document Vector RAG decreases from 0.779 under OpenAI to 0.643 under Qwen, KG-Derived Vector RAG from 0.717 to 0.625, GraphRAG 1-hop from 0.718 to 0.598, GraphRAG 2-hop from 0.667 to 0.564, and the Agentic Graph RAG from 0.850 to 0.772. The Agentic Graph RAG records the highest answer correctness under both providers.

As for the static architectures, this opposite movement shows the difference in how the generator model use the same retrieval evidence instead of the difference in the retrieval process. The results show that Qwen produces more material that remains supported by the context but not always suitable with the information that is requested. Faithfulness can reward such behaviour because it checks whether a statement is grounded in the context rather than whether the statement was required by the question. The manual statement-level evaluation examines this provider difference more directly through false-positive and false-negative statements.

Context Recall remains very similar between providers for the four static architectures since they have fixed retrieved contexts. Document Vector RAG records 0.881 and 0.882 under OpenAI and Qwen, KG-Derived Vector RAG 0.766 and 0.768, GraphRAG 1-hop 0.728 and 0.732, and GraphRAG 2-hop 0.672 and 0.664. However, the Agentic Graph RAG differs because the chosen provider also have a role in the graph traversal. Its context

recall decreases from 0.916 with OpenAI to 0.871 with Qwen. Therefore, the provider difference in the Agentic Graph RAG can reflect the differences in selected retrieval paths and final answer generation. These two effects cannot be separated in the same way as for the four static architectures.

7.2.2 Results by Question Category

The previous sections report four RAGAs metrics at the architecture level, where each figure averages over all 120 questions. This subsection presents Answer Correctness based on the question categories and generation providers, since this metric compares the generated and gold answers directly. This makes it more suitable for comparing the performance across question types than the reference-free metrics.

Table 12: Answer Correctness by Architecture, Category, and Provider (O = OpenAI, Q = Qwen)

Architecture	s. val		m. hop		aggr		desc	
	O	Q	O	Q	O	Q	O	Q
Document Vector RAG	0.820	0.640	0.620	0.472	0.781	0.611	0.896	0.850
KG-Derived Vector RAG	0.879	0.674	0.569	0.472	0.716	0.657	0.703	0.697
GraphRAG (1-hop)	0.891	0.656	0.521	0.416	0.792	0.659	0.669	0.662
GraphRAG (2-hop)	0.899	0.652	0.544	0.435	0.661	0.581	0.564	0.586
Agentic Graph RAG	0.857	0.767	0.784	0.699	0.867	0.736	0.892	0.886

As can be seen from the table [12](#), it shows that the value of each retrieval strategy depends on the information structure of the question. Single-value questions are answered best overall, from 0.730 for Document Vector RAG to 0.812 for the Agentic Graph RAG when the two providers are averaged. The three triple-based static systems perform above the document baseline here, and under OpenAI GraphRAG 2-hop performs the highest single-value figure of any system at 0.899. This is intuitive because a verbalised triple states one fact without surrounding unrelated information, which is suitable for questions that ask for exactly one value. Single-value questions are also where the penalty from moving to Qwen is most visible, with the four static systems each losing between 0.180 and 0.247. The Agentic Graph RAG is less affected here with only losing 0.090, although its Qwen Answer Correctness remains below its OpenAI score in every category, by between 0.006 for description and 0.131 for aggregation.

Description questions are the next best answered. Averaged across the two providers, the three triple-based static systems fall in depth order at

0.700, 0.666, and 0.575, compared with document chunks at 0.873 and Agentic Graph RAG at 0.889. A description question needs around nine facts about one entity, which one chunk provides together, and a triple-based system must combine from nine different retrievals under a fixed budget. Among the static architectures, Document Vector RAG records the highest scores, reaching 0.896 under OpenAI and 0.850 under Qwen, while the Agentic Graph RAG records 0.892 and 0.886, respectively.

Aggregation questions are placed between the two, from 0.621 to 0.802, and are the one category where one-hop expansion pays for itself. GraphRAG 1-hop reaches 0.726 against 0.687 for flat triple retrieval. Among the static architectures, GraphRAG 1-hop records the highest aggregation score under both providers, reaching 0.792 under OpenAI and 0.659 under Qwen. The Agentic Graph RAG again records the highest score overall, at 0.867 under OpenAI and 0.736 under Qwen.

Multi hop questions are the weakest category for every architecture, from 0.469 to 0.742, and no system reaches the level recorded in any other category. The Agentic Graph RAG records 0.784 under OpenAI and 0.699 under Qwen, while the four static architectures range from 0.521 to 0.620 under OpenAI and from 0.416 to 0.472 under Qwen. This is the widest gap between the Agentic Graph RAG and the static systems in any category. It is also the only category in which two-hop expansion does not incur a cost. GraphRAG 2-hop variant reaches 0.490 compared to 0.469 for the one-hop variant, and the difference holds in the same direction under both providers, at 0.544 compared to 0.521 under OpenAI and 0.435 compared to 0.416 under Qwen. The margin is small, but its consistency across the two conditions suggests that the second hop supplies connecting evidence that a single hop does not reach, which is the one question type where that evidence is required.

Overall, the Agentic Graph RAG ranks first on three of the four metrics, while Document Vector RAG is the most faithful to its retrieved context and places second in answer correctness. Hence, access to the graph does not, by itself, produce more correct answers than document chunks. The two providers separate by metric type instead of by architecture, with Qwen ranking higher on the reference-free metrics and OpenAI ranking higher on answer correctness throughout. The category breakdown shows that no single strategy is best everywhere, since triple-based retrieval suits single-value questions, document chunks suit descriptive ones, one-hop expansion pays for itself in aggregation, and the second hop earns its cost only on multi-hop questions. However, these metrics cannot distinguish a refusal from an incorrect answer, which is what the manual statement-level evaluation addresses.

7.3 Manual Evaluation Results

This section shows the manual statement-level evaluation result. Every generated answer is decomposed by hand into atomic statements, which are then labelled as a true positive, false positive, or false negative against the gold facts in the benchmark. Precision is the share of the generated atomic statements that correctly match the required gold claims, while recall is the share of the gold statements that the answer states. F1 is their harmonic mean. This evaluation is used as the primary measure of answer quality because it distinguishes between unsupported and missing information.

Across the 1,200 generated answers, the evaluation produced 5,272 classified statements consisting of 3,518 true positives, 824 false positives, and 930 false negatives. Table 13 shows the combined statement-level outcomes for each architecture and for each question category across both generation providers.

Table 13: Statement-Level Outcomes by Architecture and Question Category

	TP	FP	FN	P	R	F_1
<i>By architecture</i>						
Document Vector RAG	832	304	49	0.732	0.944	0.825
KG-Derived Vector RAG	676	126	214	0.843	0.760	0.799
GraphRAG (1-hop)	665	146	234	0.820	0.740	0.778
GraphRAG (2-hop)	549	171	344	0.762	0.615	0.681
Agentic Graph RAG	796	77	89	0.912	0.899	0.906
<i>By question category</i>						
<code>single_value</code>	372	97	47	0.793	0.888	0.838
<code>multi_hop</code>	249	148	132	0.627	0.654	0.640
<code>aggregation</code>	1272	407	186	0.758	0.872	0.811
<code>description</code>	1625	172	565	0.904	0.742	0.815
Total	3518	824	930	0.810	0.791	0.800

According to the table, by architecture, the Agentic Graph RAG achieves the highest overall F1 at 0.906 with precision of 0.912 and recall of 0.899. Document Vector RAG follows with an F1 of 0.825. It achieves the highest recall among the five architectures at 0.944 and precision of 0.732. KG-Derived Vector RAG achieves an F1 score of 0.799, precision of 0.843, and recall of 0.760. Furthermore, GraphRAG 1-hop records an F1 score of 0.778, with precision and recall of 0.820 and 0.740 respectively. Lastly, GraphRAG 2-hop shows the lowest overall F1 at 0.681 with precision of 0.762 and recall of 0.615. This ordering matches the Answer Correctness ranking reported in Section 7.2, which is the strongest cross-layer agreement in the evaluation.

However, the F1 score does not show the full picture, as the four fixed-retrieval systems achieve their results through different balances of precision and recall. Document Vector RAG recovers almost every gold statement in the benchmark, missing only 49, which results in its recall of 0.944, the highest of any system. However, it also contains 304 unsupported claims which results in a score of 0.732 for precision, which is the lowest of any system. The three knowledge-graph based systems show the opposite pattern. KG-Derived Vector RAG produces fewer than half as many false positives as the document baseline, but it also leaves 214 gold claims unstated. As a result, its precision of 0.843 is higher than its recall of 0.760. This gap becomes larger with each retrieval stage. For example, GraphRAG 2-hop misses 344 gold statements, and its recall falls to 0.615, meaning that almost four in ten required facts are not included in the answer. The Agentic Graph RAG is the only architecture that does not have this trade-off and achieves both high precision and recall. This shows that document chunks provide broader information coverage while also providing unrequested additional information. In contrast, the individual triples favour precision but offer limited recovery when a required fact is not retrieved.

The second part of Table [13](#) shows different error patterns. Single-value questions achieve an overall F1 of 0.838 and miss only 47 required claims, since a single-value question requires one fact, hence retrieving that fact answers it completely. Multi-hop questions show the opposite pattern and are the most difficult category overall, with the lowest precision at 0.627, recall at 0.654, and F1 at 0.640. Their difficulty lies in connecting evidence across multiple steps, where a later fact may only become relevant after an intermediate entity or relation has been successfully retrieved. Aggregation questions achieve the highest recall of 0.872, while still producing 407 out of 824 false positives. This shows that systems often recover many of the required items but may also include additional or incorrect entries, making both retrieval completeness and set selection important. Description questions achieve the highest precision of any category at 0.904 but have a lower recall of 0.742. Therefore, the main difficulty in description questions is not primarily producing incorrect information but recovering and including all of the required details within the available context. The architecture- and provider-specific results are examined in the following subsections.

7.3.1 Results by Architecture and Question Type

The overall results hide the substantial variation between question categories. Therefore, this subsection breaks the manual evaluation result based on the architectures and question category by separating the precision, recall ,and

F1 for every type of question and generation provider. The architecture-level perspective shows which information structures each retrieval strategy handles most effectively, while the cross-architecture perspective shows which retrieval strategy performs best for each type of question.

Table 14: Precision, Recall, and F1 per Architecture and Question Category Under Each Generation Provider

Architecture	Category	OpenAI			Qwen			Overall		
		<i>P</i>	<i>R</i>	<i>F</i> ₁	<i>P</i>	<i>R</i>	<i>F</i> ₁	<i>P</i>	<i>R</i>	<i>F</i> ₁
Document Vector RAG	<code>single_value</code>	.875	1.000	.933	.618	1.000	.764	.724	1.000	.840
	<code>multi_hop</code>	.634	.703	.667	.424	.676	.521	.510	.689	.586
	<code>aggregation</code>	.818	.931	.871	.487	.924	.638	.611	.928	.737
	<code>description</code>	.982	.991	.986	.816	.986	.893	.892	.988	.938
	<i>All categories</i>	.884	.948	.915	.624	.941	.751	.732	.944	.825
KG-Derived Vec. RAG	<code>single_value</code>	1.000	.952	.976	.812	.929	.867	.898	.940	.919
	<code>multi_hop</code>	.828	.615	.706	.512	.595	.550	.639	.605	.622
	<code>aggregation</code>	.900	.869	.884	.703	.837	.764	.790	.853	.820
	<code>description</code>	.981	.697	.815	.872	.682	.765	.924	.689	.790
	<i>All categories</i>	.940	.770	.847	.763	.749	.756	.843	.760	.799
GraphRAG (1-hop)	<code>single_value</code>	.929	.929	.929	.722	.929	.813	.812	.929	.867
	<code>multi_hop</code>	.750	.615	.676	.556	.513	.533	.647	.564	.603
	<code>aggregation</code>	.905	.912	.908	.713	.905	.798	.798	.908	.849
	<code>description</code>	.952	.630	.758	.825	.614	.704	.884	.622	.730
	<i>All categories</i>	.913	.749	.823	.743	.730	.737	.820	.740	.778
GraphRAG (2-hop)	<code>single_value</code>	1.000	.786	.880	.672	.929	.780	.791	.857	.823
	<code>multi_hop</code>	.867	.634	.732	.475	.500	.487	.643	.570	.604
	<code>aggregation</code>	.875	.772	.821	.632	.710	.669	.739	.741	.740
	<code>description</code>	.895	.505	.645	.736	.482	.582	.810	.493	.613
	<i>All categories</i>	.895	.629	.739	.659	.600	.628	.762	.615	.681
Agentic Graph RAG	<code>single_value</code>	.872	.810	.840	.641	.610	.625	.756	.711	.733
	<code>multi_hop</code>	.842	.865	.853	.633	.838	.721	.724	.851	.783
	<code>aggregation</code>	.944	.925	.934	.883	.938	.910	.912	.931	.922
	<code>description</code>	.981	.940	.960	.980	.904	.941	.981	.922	.950
	<i>All categories</i>	.946	.916	.931	.878	.882	.880	.912	.899	.906
All architectures		.915	.802	.855	.725	.780	.751	.810	.791	.800

Performance by Architecture

The results in the table above show that the five architectures behave differently across the four question categories. Some systems perform better by covering more of the required information while others are more selective. The following analyses each architecture and relates its overall performance to the category-level results.

Document Vector RAG achieves an overall precision of 0.732, Recall of 0.944, and F1 of 0.825. Recall stays higher than precision in every category. Its strongest category is description, with an F1 of 0.938, precision of 0.892, and recall of 0.988, where retrieving relatively rich article chunks is advantageous because a single chunk can contain several facts about the same entity. This is followed by single-value questions, which also achieve a perfect recall of 1.000, with a lower precision of 0.724. The lower precision here indicates that the generated answers contain additional unrequested information. Aggregation scores an F1 of 0.737 with a high recall of 0.928 and a precision of 0.611. Its weakest category is multi-hop, with an F1 of 0.586. Here, independently retrieved text chunks may contain relevant individual facts without explicitly showing the relations needed to connect them.

KG-Derived Vector RAG achieves an overall F1 of 0.799, a precision of 0.843, and a recall of 0.760. Its strongest performance is on single-value questions (F1 = 0.919), because each verbalised triple provides a focused fact with limited surrounding information. This architecture also performs well on aggregation, with an F1 of 0.820. For description questions, its high precision but lower recall shows that the retrieved facts are generally accurate but do not always cover all required information. Its weakest category is multi-hop with F1 of 0.622, where a fact connected through an intermediate entity may not rank highly enough to enter the context.

GraphRAG 1-Hop achieves an overall precision at 0.820, recall at 0.740, F1 at 0.778. It performs the best on single value with F1 of 0.867 and aggregation questions with F1 0.849. This strong aggregation result shows the benefit of retrieving facts from the direct neighbours of an entity when the required items are directly connected to it. However, its performance falls on description and multi-hop questions. Description reaches an F1 of 0.730 with precision of 0.884 and recall of 0.622 which shows that the architecture is accurate, but does not recover enough of the separate facts required for a complete description. Its weakest category is multi-hop, with an F1 of 0.603, because one expansion step may not reach the complete relational chain required by the question.

GraphRAG 2-Hop achieves an overall precision of 0.762, a recall of 0.615, and an F1 score of 0.681. Its strongest category is single-value with an F1 of 0.823. The next best is aggregation with F1 of 0.740 and almost equal precision and recall of 0.739 and 0.741, respectively. In multi-hop, GraphRAG 2-Hop is performing better than GraphRAG 1-hop, which indicates that a

second expansion can provide additional connected evidence. However, under the shared token budget, the second hop is only able to retrieve a smaller seed set, which reduces the retrieval breadth. As can be seen from the table, this causes lower recall, especially for description questions.

Agentic Graph RAG achieves an overall precision of 0.912, a recall of 0.899, and an F1 of 0.906. According to the table, its strongest categories are description and aggregation, with F1 scores of 0.950 and 0.922, respectively. Multi hop questions reach an F1 of 0.783, with 0.724 precision and 0.851 recall. This high recall score on multi-hop questions shows that iterative traversal helps the system recover facts that depend on intermediate entities. Its weakest category is single-value with F1 of 0.733, where a traversal is unnecessary for one direct fact and can introduce unrequested information.

Cross-Architecture Comparison by Question Type

The architecture-level results show how each retrieval strategy behaves internally, while a comparison by question type shows which type of retrieval is most suitable for each task. The four categories also differ in their overall error patterns. Across all architectures and providers, single-value questions reach an F1 of 0.838, description score of 0.815, aggregation score of 0.811, and multi-hop score of 0.640.

Single-Value questions are led by KG-Derived Vector RAG with a combined F1 at 0.919, followed by GraphRAG 1-Hop at 0.867, Document Vector RAG at 0.840, GraphRAG 2-Hop at 0.823, and the Agentic Graph RAG at 0.733. Document Vector RAG records the highest recall in the category at 1.000. The relatively high recall shows that the required factual value is usually available to the systems. Therefore, the main difference between architectures is how directly they can return that value without adding unrequested information. The strongest result from flat triple retrieval shows that additional graph traversal is generally unnecessary when the question asks for one property of an already identified entity

Multi-hop is the most difficult category, with an overall precision of 0.627, a recall of 0.654, and F1 of 0.640. The Agentic Graph RAG achieves the highest F1 at 0.783, followed by KG-Derived Vector RAG at 0.622, GraphRAG 2-Hop at 0.604, GraphRAG 1-Hop at 0.603, and Document Vector RAG at 0.586. The fact that all graph-based retrieval performs better than the baseline is expected for multi-hop questions because answering them requires connecting facts through one or more intermediate entities

or relations. Therefore, graph-based retrieval is the most suitable retrieval system for this category because it can explicitly follow these connections. The main difference comes from recall, with the Agentic System reaching 0.851 compared with 0.689 for Document Vector RAG. GraphRAG 2-Hop also performs slightly better than 1-hop, which is consistent with the required evidence placed further than the immediate neighbourhood. The much stronger performance of the Agentic Graph RAG further shows that depth is most effective when the system can follow the intermediate relations that are relevant to the question instead of expanding the graph uniformly. Together with aggregation, this is one of only two categories in which every graph-based architecture scores above the document baseline.

Aggregation questions achieve an overall precision of 0.758, a recall of 0.872, and F1 of 0.811. As with multi-hop questions, every graph-based architecture scores at or above the document baseline here, since the items an aggregation question asks for are stored as separate triples attached to the same entity. The Agentic Graph RAG performs best with an F1 of 0.922, while GraphRAG 1-hop is the strongest static architecture at 0.849. KG-Derived Vector RAG follows at 0.820, GraphRAG 2-Hop at 0.740, and Document Vector RAG at 0.737. With a recall of 0.908 against 0.853 for KG-Derived Vector RAG, GraphRAG 1-Hop shows that static neighbourhood expansion is most useful when the required items are directly connected to the same entity. The weaker result from GraphRAG 2-Hop shows that expanding further does not necessarily add useful information once the relevant immediate neighbourhood has already been reached.

Description questions achieve an overall precision of 0.904, a recall of 0.742, and F1 of 0.815. The Agentic Graph RAG achieves the highest combined F1 at 0.950, followed closely by Document Vector RAG at 0.938. KG-Derived Vector RAG reaches 0.790, GraphRAG 1-hop 0.730, and GraphRAG 2-Hop 0.613. The Agentic Graph RAG records the highest precision at 0.981, while Document Vector RAG records the highest recall at 0.988. The recall values show the difference better. Document Vector RAG reaches 0.988 compared with 0.922 for the Agentic Graph RAG, 0.689 for KG-Derived Vector RAG, 0.622 for GraphRAG 1-Hop, and 0.493 for GraphRAG 2-hop. Description questions require many facts about the same entity. Document chunks carry several of these facts together, whereas triple-based systems require more of the individual facts to be retrieved separately. Fixed expansion becomes increasingly restrictive as more of the available context is spent on neighbourhood material.

7.3.2 Effect of the Generation Provider

Across all architectures, OpenAI achieves a precision of 0.915, a recall of 0.802, and an F1 score of 0.855. Under Qwen, precision is 0.725, recall is 0.780, and F1 is 0.751. Therefore, the difference is much larger for precision, which decreases by 0.190 overall, compared to recall, which decreases by only 0.022. This indicates that the two providers are relatively similar in how often required gold claims appear in their answers, but differ considerably in how selectively they formulate those answers. Under the manual evaluation protocol, the lower precision for Qwen reflects a greater number of additional or incorrect factual statements relative to the gold answers.

For the four static architectures, the retrieval configuration and retrieved context stay the same across providers. Therefore, any provider-level difference arises after retrieval. All four architectures achieve higher F1 under OpenAI. Document Vector RAG shows the largest overall difference, decreasing from 0.915 to 0.751. Its recall remains almost unchanged at 0.948 and 0.941, while precision decreases from 0.884 to 0.624. The broad textual context therefore appears sufficient for both providers to recover most required information, but the providers differ in how selectively they use the surrounding material when constructing the final answer. The remaining three architectures follow the same direction, with KG-Derived Vector RAG decreasing from an F1 of 0.847 to 0.756, GraphRAG 1-Hop from 0.823 to 0.737, and GraphRAG 2-Hop from 0.739 to 0.628. One working example is the question "Who became mayor of Vienna after Karl Seitz was removed from office in 1934?" GPT-5.4 answered only "Richard Schmitz." while Qwen 3.5 Plus added that Schmitz was appointed as government commissioner by the Dollfuß government on 12 February 1934 and elected as mayor on 1 May 1934. Both answers contain the required gold claims but the additional statements from Qwen lower its score.

An interesting category-level difference occurs for GraphRAG 2-Hop on multi-hop questions. Even though overall, GraphRAG 2-hop performs better than 1 hop in multi-hop, under OpenAI, it achieves a precision of 0.867, a recall of 0.634, and an F1 of 0.732, making it the strongest static architecture in this category. Under Qwen, precision decreases to 0.475, recall to 0.500, and F1 to 0.487, making it the weakest architecture in the category. Therefore, the usefulness of expanded graph context can depend on the generator as well, since providing additional relational evidence does not by itself guarantee better final answers. The generator must also be able to distinguish relevant evidence from information that should not be included in the response.

The Agentic Graph RAG requires a different interpretation because changing the provider affects both retrieval and generation. The model participates

in relation selection, entity selection, and traversal decisions before producing the final answer. The Agentic Graph RAG is more stable overall, with precision, recall, and F1 of 0.946, 0.916, and 0.931 under OpenAI compared with 0.878, 0.882, and 0.880 under Qwen. Its F1 decreases from 0.934 to 0.910 for aggregation, from 0.960 to 0.941 for description, from 0.853 to 0.721 for multi-hop, and from 0.840 to 0.625 for single-value questions. Nevertheless, the comparatively small overall difference between providers shows that the architecture remains relatively resilient to the provider change, although single-value and multi-hop questions show larger category-level differences. Because the provider also participates in retrieval-path selection, these differences can reflect both retrieval and final answer generation.

Most category-level rankings remain stable across providers. KG-Derived Vector RAG is strongest for single-value questions under both OpenAI and Qwen, with F1 scores of 0.976 and 0.867, respectively. The Agentic Graph RAG remains strongest for multi-hop at 0.853 and 0.721, and for aggregation at 0.934 and 0.910. Description is the main exception since, under OpenAI, Document Vector RAG leads with an F1 of 0.986 compared with 0.960 for the Agentic Graph RAG, whereas under Qwen, the Agentic Graph RAG leads at 0.941 while Document Vector RAG falls to 0.893. Therefore, the provider substantially influences absolute performance, especially through precision, but does not overturn most of the architecture-by-question-type patterns observed in the evaluation.

Overall, the manual evaluation shows that the benefit of graph integration depends on the question category. Graph-based retrieval is strongest where the answer requires relations, namely multi-hop questions and aggregation questions. On the other hand, vector-based retrieval is strongest for single-value questions. However, for description questions, the Agentic Graph RAG performs the best, which is contrary to the prediction.

8 Discussion

The main research questions ask whether increasing graph integration can improve the performance of conversational access to cultural heritage knowledge bases. According to the three layers of evaluation, the answer is that the right level of graph integration depends on how much connectedness is needed for the question. Graph integration becomes more useful as relational complexity increases. This pattern holds across the benchmark. Flat triple retrieval performs best on single-value questions, one hop expansion is the strongest static architecture on aggregation questions, and the Agentic Graph RAG performs substantially better on multi-hop questions, while on description

questions, the Agentic Graph RAG and Document Vector RAG perform almost equally. Therefore, the level of graph integration depends on the connective structure of the question instead of increasing uniformly.

The three measurement layers also support this reading, with two of the layers that measure the final answers showing strong agreement. The F1 rank from the manual evaluation and the rank for Answer Correctness from RAGAs place the five architectures in the same overall rank. Since one layer is using the judge model while the other layer is not using any judge model, the agreement represents the highest level of validation of this evaluation. Context Recall and Term Recall also move in the same way for the static architectures, which shows that the retrieval level measurement captures the same underlying behaviour.

Nevertheless, the retrieval quality measurement gives a different overview for multi-hop questions. The Agentic Graph RAG has the lowest Term Recall of all architectures in this category at 0.809, but the highest F1 score at 0.783. Therefore, it answers these questions well while retrieving less of the required evidence than the other architectures. The reason is that traversal follows the relations linking one fact to the next, so the generator receives a connected path instead of a set of separately matched facts. For questions that require connection, how the retrieved evidence is linked matters more than how much of it there is. The following paragraphs examines each research question and hypothesis in turn against these results.

Graph-Based and Vector-Based Retrieval Across Question Categories Research question 1 asks whether the relative claim level F1 of vector based and knowledge graph based RAG depends on the question category and in which direction. Hypothesis 1 predicted higher F1 for the graph systems on multi-hop and aggregation questions and lower F1 on single value and description questions. Following the architecture-family comparison defined in Section 5.1, the first hypothesis is partially supported, specifically supported for three out of four categories, and is not supported for description questions.

The predictions for relational questions hold. On multi-hop questions, every graph-based architecture exceeds the document baseline, with the Agentic Graph RAG achieving an F1 score of 0.783 compared to 0.586 for the Document Vector RAG. On aggregation questions, the Agentic Graph RAG reaches an F1 score of 0.922, and GraphRAG 1-hop reaches an F1 score of 0.849. Answer Correctness also agrees by placing GraphRAG 1-hop as the highest-performing architecture among the static architectures on aggregation at 0.726 against 0.687 for flat triple retrieval. The prediction for single-value questions also holds, with the KG-Derived Vector RAG achieving the highest

F1 score of 0.919 and the Agentic Graph RAG achieving 0.733.

Description questions are the exception. The prediction holds for the two static graph architectures, which perform lower than the F1 score of 0.938 for Document Vector RAG. However, the Agentic Graph RAG scores the highest F1 with 0.950. Answer Correctness also follows the same division, with 0.873 for the Document Vector RAG and 0.889 for the Agentic Graph RAG. This hypothesis assumes that graph-based retrieval, in general, is not suitable for descriptive questions since a description needs a lot of attributes from one entity, which is provided at once by the text chunks but not by the triple. Instead, the research findings show that the weakness is not in the graph access itself, because the same graph, when explored selectively, performs the best in this category. What differentiates between architectures is not whether the graph is used, but how the graph is explored.

This category dependence is consistent with existing comparative work. Han et al. report that conventional and graph-based retrieval show complementary strengths instead of clear dominance. Conventional retrieval performs well on single-hop, fact-centric queries, while graph-based retrieval performs better on reasoning-intensive multi-hop queries [11]. This thesis reproduces that pattern across four categories instead of two. A further difference concerns the graph itself, since the graphs in that study are constructed from text by language models, and the authors report that performance is sensitive to the quality of that construction, whereas the graph used here is curated by the City of Vienna through Semantic MediaWiki and is not the product of automated extraction [16].

Effect of Retrieval Unit Granularity on Precision and Recall

Research question 2 asks how the granularity of the retrieval unit, document chunks compared with verbalised RDF triples, affects the balance between claim-level precision and recall. Hypothesis 2 predicted that triple-based retrieval would result in higher precision but lower recall than document-based retrieval. The comparison between these two architectures is significant because both of them use embedding-based retrieval without traversal. The second hypothesis is supported. As shown in Table 13, the switch from document chunks to verbalised triples increases precision from 0.732 to 0.843 and lowers recall from 0.944 to 0.760, resulting in a rise of 0.111 compared to a decrease of 0.184.

The retrieval layer confirms that the recall cost originates before generation, as Term Recall decreases from 0.943 to 0.893 and Context Recall from 0.881 to 0.767. Faithfulness is 0.916 for Document Vector RAG compared to 0.818 for KG-Derived Vector RAG, even though its claim-level precision is

lower. Document chunks contain several relevant facts together, which provide broader coverage and redundancy, but also surrounding information that leads to additional statements that reduce precision. Verbalised triples provide more focused evidence because each retrieval unit represents an individual fact, which supports greater selectivity but provides less redundancy when a required fact is not retrieved, which reduce recall.

Previous work frames the granularity as a matter of chunk size and overlap in the chunk prose. Retrieving too little causes the model to lack evidence, while taking too much evidence inputs redundant and irrelevant context [35]. Gao et al. also see the correct granularity as something that depends on the downstream task instead of following a single fixed setting [10]. The results from this thesis show that this trade-off still applies at the extreme, which is when one retrieval-unit is no longer a paragraph but one atomic fact. Moreover, the redundancy of prose chunks, usually discussed as a weakness named the lost-in-the-middle effect [26], is a source of recall here because it increases the chance that a needed fact is still present in the retrieved context.

Agentic Traversal Compared with Fixed Expansion Research Question 3 asks whether adding an agentic LLM layer to Graph RAG improves overall claim-level F1 compared with Graph RAG using fixed traversal. Hypothesis 3 predicted a higher claim-level F1 for Graph RAG enhanced with an agentic LLM layer, especially on multi-hop and aggregation questions. Based on the findings, Hypothesis 3 is supported. The Agentic Graph RAG achieves an overall F1 of 0.906, compared with 0.778 for one-hop GraphRAG and 0.681 for two-hop GraphRAG, and the highest F1 on multi-hop and aggregation questions.

RAGAs confirms this across every metric it reports, for example Answer Correctness of 0.811 against 0.658 and 0.615 for GraphRAG 1-hop and 2-hop, and Context Recall of 0.894 against 0.730 and 0.668. The advantage can also be seen for multi-hop and aggregation questions. The Agentic Graph RAG also records the highest precision of 0.912. This shows that as long as the triple is retrieved selectively, it can outperform the inherent trade-off observed for other architectures.

The interesting part is the single-value category, where the Agentic Graph RAG scores below one hop GraphRAG and two hop GraphRAG. Questions that require one directly accessible fact gain little from iterative relation selection and entity scoring. For such questions, the additional traversal introduces costs and additional opportunities for irrelevant evidence without providing a structural benefit. The reduction is considerably larger for Qwen 3.5 Plus in this category, which is examined further under research question

4.

These results aligned with Think-on-Graph, which is used as the main underlying architecture for this agentic approach. Sun et al. report that iterative beam search on a knowledge graph supports deeper multi-hop reasoning without additional training and achieves the best performance on six of nine datasets [32]. The evaluation in this thesis expands this evidence to a context that is different in two aspects. Think-on-Graph was evaluated on Freebase with English benchmarks and a beam width of 3, whereas this thesis uses a German cultural heritage wiki and a beam width of 20. The depth limit itself is the same between the original architecture and the one in this thesis. Therefore, the advantage over fixed traversal transfers across domains, languages, and search widths.

Answer Quality Across the Two Generation Models This research question asks how the choice of generation model affects claim-level precision, recall, and F1 across the evaluated retrieval architectures. Hypothesis 4 predicted that the frontier GPT higher overall answer quality than Qwen across the evaluated retrieval architectures and question categories. Hypothesis 4 is supported. Across all architectures, OpenAI achieves a precision of 0.915, recall of 0.802, and F1 of 0.855. On the other hand, Qwen achieves a precision of 0.725, a recall of 0.780, and an F1 score of 0.751.

The provider difference is considerably larger for precision than for recall. Precision differs by 0.190, while recall differs by only 0.022. Since the retrieved evidence was held constant across providers for the static architectures, this means that Qwen more frequently included additional statements beyond the required gold answer. In other words, either Qwen fails to follow the benchmark’s closed world instructions or produces additional statements that are wrong, which results in higher false positives. RAGAs shows the same behaviour from the other direction, since Qwen records higher faithfulness for all four static architectures while recording lower answer correctness throughout. This means that its additional statements remain supported by the retrieved context but are not the ones the question required. The difference between providers is also not uniform for all architectures. The F1 difference between providers happens the most in Document Vector RAG, which is 0.164, followed by GraphRAG 2-hop at 0.111, KG-Derived Vector RAG at 0.091, and GraphRAG 1-hop at 0.086, with the lowest for Agentic Graph RAG at 0.051. This shows that the architecture that passes the most material beyond the required facts is the most sensitive to the choice of LLM. On the other hand, the architecture that selects evidence relation by relation is the least sensitive. For the Agentic Graph RAG, this interpretation

comes with a qualification. Since the generation provider also chooses the relationships, the differences between providers also reflect both the chosen paths and the final generation process. These two also cannot be separated as in the four static architectures.

This pattern is consistent with Ru et al., who find that GPT-4 shows higher context utilisation and lower noise sensitivity than three openly released generators, which remain faithful but tend to trust the retrieved context as a whole [30]. Similarly, Qwen 3.5 Plus achieves a higher faithfulness score compared to the GPT-5.4 on every fixed architecture. The score rises from 0.899 to 0.932 on Document Vector RAG but obtains a lower Answer Correctness score that is lower than GPT on the same architecture. Given the same context, Qwen keeps more of it in the answer instead of selecting from it, and the two LLMs differ almost entirely in what they add instead of in which required facts they recover. The result also agrees with public benchmarks, on which GPT-5.4 scores higher than the corresponding open-weight Qwen 3.5 model [6].

Implications for the Use Case The use case section derived three requirements from the consultation with the domain experts, and this thesis addresses all three. The first requirement is that the answer needs to be based on the retrieved content without unsupported speculation. The Faithfulness score is at least 0.788 for every architecture and reaches 0.916 for Document Vector RAG. Hence, invention in the strict sense is uncommon. The second requirement was support for several kinds of information needs. The experts identified aggregation and multi-hop questions as the most difficult to answer through the current Vienna History Wiki interface. These are the two categories in which selective traversal contributes the most. The last requirement is that the comparisons must take into account the difference between article content and RDF representation, which is addressed by using the source-equivalence condition. That condition carries one qualification. Because ontology coverage is uneven, some concepts appear only in articles without being formally encoded, and questions of that kind are excluded from the benchmark. Therefore, the results hold for conditions where both representations contain the required information.

⁶LLM Stats, <https://llm-stats.com/leaderboards/llm-leaderboard?orgs=openai,qwen>, accessed 21 March 2026.

9 Conclusion

This thesis investigated whether increasing graph integration improves conversational access to a cultural heritage knowledge base and whether the value of graph-based retrieval depends on the type of question being answered. Five retrieval-augmented generation architectures were implemented over the Vienna History Wiki, namely Document Vector RAG over unstructured page chunks, KG-Derived Vector RAG over verbalised RDF triples, GraphRAG with one-hop neighbourhood expansion, GraphRAG with two-hop neighbourhood expansion, and Agentic Graph RAG that dynamically selects its traversal steps. Each architecture was evaluated with OpenAI’s GPT 5.4 and Alibaba’s Qwen 3.5 plus on a balanced benchmark of 120 questions covering single-value, multi-hop, aggregation, and description questions.

To answer the main research question, the findings of this thesis show that the appropriate degree of graph integration depends on the structure and complexity of the information required by the question. For single-value questions where the answer is usually in the form of a direct fact, flat retrieval towards verbalised triples performs the best among all five architectures. The aggregation questions benefit from limited graph integration, with the Agentic Graph RAG performing the best in this category. However, the one-hop GraphRAG also performs strongly because some values that are related often can be gathered from the immediate neighbourhood of an entity. On the other hand, multi-hop questions need a higher graph integration because the answer depends on the evidence that is connected through an intermediate entity. In this category, Agentic Graph RAG performs the best because it could iteratively choose which relation or entity that need to be followed while also building the connected evidence paths. Lastly, for the description questions, even though it is performed best by the Agentic Graph RAG, document retrieval is still very competitive because one text chunk can contain several relevant attributes to answer the question.

These category-dependent results show that greater graph integration is more beneficial when the relational need from the question increase, but simply expanding more of the graph is not sufficient. One-hop expansion is effective when the evidence needed is concentrated around the immediate neighbourhood, which can be seen in aggregation questions. Two-hop GraphRAG shows that expanding the graph even further could provide additional benefit when the question needs evidence that is located outside of the immediate neighbourhood. This can be seen from the multi hop questions where the two-hop is performing better than the one-hop Graph RAG. Under the shared token budget, second-hop expansion also changes the balance be-

tween retrieval breadth and traversal depth, since part of the available context capacity is used for evidence from more distant neighbourhoods. These results suggest that deeper graph traversal has potential when additional relational evidence is required. Its benefit becomes stronger when traversal is performed more selectively which is demonstrated by the Agentic Graph RAG.

Overall, the findings suggest that graph integration should be adapted to the structural complexity of the question. Simple factual questions require little or no graph traversal, questions involving sets of related values can benefit from local graph expansion, and questions requiring connected reasoning benefit more from deeper traversal. However, as the level of graph integration increases, selectivity becomes increasingly important to prevent irrelevant context from reducing the value of additional relational evidence. Therefore, the main finding of this thesis is not that graph-based retrieval universally replaces vector-based retrieval, but that the retrieval strategy should be matched to the information structure of the questions.

Furthermore, this thesis contributes a reproducible pipeline for applying RAG to Semantic MediaWiki-based cultural heritage wikis, a 120-question benchmark across four question categories, a systematic comparison of five retrieval architectures, and an agentic graph traversal system adapted to the Vienna History Wiki. The code and data are publicly available.

9.1 Limitations

Several limitations need to be taken into account when interpreting the results. First, the evaluation is conducted on a single German-language cultural heritage knowledge base. The Vienna History Wiki has its own ontology, Semantic MediaWiki structure, multilingual predicates, and data characteristics. As a result, the conclusions of this study cannot be assumed to generalise to other knowledge graphs, domains, ontologies, or languages.

Second, the benchmark contains 120 manual constructed questions, with thirty questions within each category. This number allows for cross architecture comparison, but the sample size itself in each category is still small for drawing general conclusion. Third, this study only uses two generation models that are accessed through API. They allow the effect of the generator on the final answer to be analysed, but they do not represent the wide model variations that exist today. Therefore, the provider effects observed in this study may not generalise to models with different sizes, reasoning capabilities, training procedures, or deployment methods.

The retrieval evaluation also has its own limitations. It mainly measures whether the evidence required to answer a question is present in the retrieved context, but not how much irrelevant or redundant information is retrieved

alongside it. This is important since a larger context does not only add noise that confuses the model in choosing the relevant information, but also increases the input tokens, which increases inference cost and latency. The current evaluation does not measure the trade-off between the amount of relevant evidence retrieved, the amount of additional context introduced, and the computational cost of processing it.

Another limitation is that this study cannot determine whether an answer is generated because the model actually uses the retrieved context or because the information is already available in the LLM’s parametric knowledge. This is particularly relevant for simple direct or single-value questions such as facts about well-known people or events, which the model may be able to answer without retrieval. Although faithfulness can indicate whether a statement is supported by the retrieved context, such support does not prove that the context was the actual source of the answer.

Lastly, the improved performance of the Agentic Graph RAG comes at a higher computational cost. The system requires multiple language-model calls and SPARQL queries for relation selection, candidate selection, sufficiency checking, and final answer generation. Although the traversal is bounded by a maximum depth, this study does not provide a detailed empirical analysis of the average number of traversal steps, model calls, or SPARQL queries required for each question category. Therefore, the quality gains from agentic traversal should be interpreted together with an efficiency cost that has not yet been fully measured or optimised.

9.2 Future Work

Several research directions can be developed based on the findings and limitations of this study. First, the benchmark can be extended both in the number of questions and in the types of questions included. For example, it could include questions that have no answer in the wiki to test whether the systems correctly return the fallback instead of producing an unsupported answer. Second, the evaluation should be extended to other Semantic MediaWiki instances, cultural heritage collections, domains, and languages. Together, these extensions would help determine how far the findings of this thesis generalise beyond the Vienna History Wiki. Third, the model comparison can include other types of models. Specifically, locally deployed open-weight models with different sizes can be evaluated to see whether the effects of retrieval architecture and generation provider remain consistent.

Future work can also isolate the contribution of retrieval from the parametric knowledge of the LLM. The same benchmark questions can be evaluated with and without the retrieved context from which the generated answer

is compared. This is especially relevant to simple questions or single-value questions that can be answered correctly by the LLM without external evidence. This can help differentiate between the condition when the retrieval gives additional knowledge and the conditions when retrieval only provides grounding for the information that the model already knows.

Retrieval evaluation can also be extended by measuring unnecessary context. A complementary metric can be used to measure the proportion of retrieved statements or tokens that actually contribute to the expected answer. With this, retrieval quality can be evaluated not only based on evidence coverage but also based on context efficiency. Since longer contexts increase the total number of input tokens, this metric can also connect retrieval quality directly with inference cost and latency.

The agent architecture can also be analysed in more detail, especially regarding its efficiency. Future work could note the average traversal depth, total relation-selection calls, candidate-scoring calls, SPARQL queries, processed tokens, latency, and monetary cost for each question category. This can provide a better picture regarding the trade-off between the increase in answer quality from agentic traversal and the necessary resources. Caching entity-linking results, the use of cheaper models for intermediate decisions, and stronger early-termination criteria can be tested as strategies to lower the cost.

Future work could also compare the retrieval-based approaches of this thesis with direct structured-query methods. Instead of retrieving graph evidence and converting it into context for the LLM, the model could directly formulate SPARQL queries or interact with the knowledge graph through a tool-based interface such as SPARQL-MCP [6]. Such a comparison would show under which conditions direct query formulation is more suitable than graph-based retrieval for natural-language access to structured cultural heritage data.

Finally, the category-dependent results show the potential use of a routing architecture. Since no single retrieval strategy is optimal for every question type, an initial classifier could identify the information need of an incoming question and route it to the most suitable pipeline. Simpler direct questions could be handled using document or triple retrieval, while questions requiring connected or aggregated evidence could be directed to graph expansion or agentic traversal. A hybrid retrieval approach combining document chunks with verbalised graph facts could also be explored to determine whether the higher recall of document retrieval and the higher precision of atomic graph facts can be combined within a single architecture.

References

- [1] Sarat Ahmad, Zeinab Nezami, Maryam Hafeez, and Syed Ali Raza Zaidi. Benchmarking vector, graph and hybrid retrieval augmented generation (rag) pipelines for open radio access networks (oran), 2025.
- [2] Felipe Almeida and Geraldo Xexéo. Word embeddings: A survey, 2023.
- [3] Simran Arora, Avner May, Jian Zhang, and Christopher Ré. Contextual embeddings: When are they worth it?, 2020.
- [4] Naga Sowjanya Barla and Jacopo de Berardinis. Competency questions as executable plans: a controlled rag architecture for cultural heritage storytelling, 2026.
- [5] Shengyuan Chen, Chuang Zhou, Zheng Yuan, Qinggang Zhang, Zeyang Cui, Hao Chen, Yilin Xiao, Jiannong Cao, and Xiao Huang. You don’t need pre-built graphs for rag: Retrieval augmented generation with adaptive reasoning structures, 2025.
- [6] Daniel Dobriy, Frederik Bauer, Amr Azzam, Debayan Banerjee, and Axel Polleres. Agentic sparql: Evaluating sparql-mcp-powered intelligent agents on the federated kgqa benchmark, 2026.
- [7] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization, 2025.
- [8] Harun Elkiran and Jawad Rasheed. Evarag: Evaluating advanced rag techniques with indexing and distance metrics. *IEEE Access*, 13:215724–215747, 2025.
- [9] Shahul Es, Jithin James, Luis Espinosa-Anke, and Steven Schockaert. Ragas: Automated evaluation of retrieval augmented generation, 2024.
- [10] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024.
- [11] Haoyu Han, Li Ma, Yu Wang, Harry Shomer, Yongjia Lei, Zhisheng Qi, Kai Guo, Zhigang Hua, Bo Long, Hui Liu, Charu C. Aggarwal, and Jiliang Tang. Rag vs. graphrag: A systematic evaluation and key insights, 2026.

- [12] Haoyu Han, Yu Wang, Harry Shomer, Kai Guo, Jiayuan Ding, Yongjia Lei, Mahantesh Halappanavar, Ryan A. Rossi, Subhabrata Mukherjee, Xianfeng Tang, Qi He, Zhigang Hua, Bo Long, Tong Zhao, Neil Shah, Amin Javari, Yinglong Xia, and Jiliang Tang. Retrieval-augmented generation with graphs (graphrag), 2025.
- [13] Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh V. Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. G-retriever: Retrieval-augmented generation for textual graph understanding and question answering, 2024.
- [14] Bowen Jin, Chulin Xie, Jiawei Zhang, Kashob Kumar Roy, Yu Zhang, Zheng Li, Ruirui Li, Xianfeng Tang, Suhang Wang, Yu Meng, and Jiawei Han. Graph chain-of-thought: Augmenting large language models by reasoning on graphs, 2024.
- [15] Paul Kelly, Jonathan Schild, and Amir Jafari. FolkRAG: A retrieval-augmented generation system for cultural heritage materials. *Neural Computing and Applications*, 37:20281–20297, 2025.
- [16] Bernhard Krabina. Building a knowledge graph for the history of vienna with semantic mediawiki. *Journal of Web Semantics*, 76:100771, 2023.
- [17] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.
- [18] Zijian Li, Qingyan Guo, Jiawei Shao, Lei Song, Jiang Bian, Jun Zhang, and Rui Wang. Graph neural network enhanced retrieval for question answering of llms, 2024.
- [19] Qi Liu, Matt J. Kusner, and Phil Blunsom. A survey on contextual embeddings, 2020.
- [20] Yang Liu, Lan Lan, Jiahuan Cao, Hiuyi Cheng, Kai Ding, and Lianwen Jin. Large-scale corpus construction and retrieval-augmented generation for Ancient Chinese poetry: New method and data insights. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 779–817, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics.

- [21] Le Ma, Ran Zhang, Yikun Han, Shirui Yu, Zaitian Wang, Zhiyuan Ning, Jinghan Zhang, Ping Xu, Pengjiang Li, Ziyue Qiao, Wei Ju, Chong Chen, Dongjie Wang, Kunpeng Liu, Pengyang Wang, Pengfei Wang, Yanjie Fu, Chunjiang Liu, Yuanchun Zhou, and Chang-Tien Lu. A comprehensive survey on vector database: Storage and retrieval technique, challenge, 2026.
- [22] Shengjie Ma, Chengjin Xu, Xuhui Jiang, Muzhi Li, Huaren Qu, Cehao Yang, Jiabin Mao, and Jian Guo. Think-on-graph 2.0: Deep and faithful large language model reasoning with knowledge-guided retrieval augmented generation, 2025.
- [23] Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020.
- [24] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.
- [25] MediaWiki. Mediawiki xml export schema, version 0.10. <https://www.mediawiki.org/xml/export-0.10.xsd>, 2026. Accessed 21 January 2026.
- [26] Boci Peng, Yun Zhu, Yongchao Liu, Xiaohe Bo, Haizhou Shi, Chuntao Hong, Yan Zhang, and Siliang Tang. Graph retrieval-augmented generation: A survey, 2024.
- [27] Ragas. Ragas documentation: Metrics. <https://docs.ragas.io/en/latest/concepts/metrics/>. Accessed: 2026-02-21.
- [28] David Rau, Shuai Wang, Hervé Déjean, and Stéphane Clinchant. Context embeddings for efficient answer generation in rag, 2024.
- [29] Ishak Riali, Raphaël Gerssen, Elizabeth Rodríguez Estrada, Gabriel Spautz Vieira, and Martin Berger. Generation of multi-modal narratives of cultural objects from knowledge graphs and LLMs. In *Proceedings of the Third International Workshop of Semantic Digital Humanities (SemDH 2026), co-located with ESWC 2026*, CEUR Workshop Proceedings, Dubrovnik, Croatia, 2026. CEUR-WS.org.
- [30] Dongyu Ru, Lin Qiu, Xiangkun Hu, Tianhang Zhang, Peng Shi, Shuaichen Chang, Cheng Jiayang, Cunxiang Wang, Shichao Sun, Huanyu Li, Zizhao Zhang, Binjie Wang, Jiarong Jiang, Tong He, Zhiguo Wang,

- Pengfei Liu, Yue Zhang, and Zheng Zhang. Ragchecker: A fine-grained framework for diagnosing retrieval-augmented generation. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 21999–22027. Curran Associates, Inc., 2024.
- [31] Alireza Salemi and Hamed Zamani. Evaluating retrieval quality in retrieval-augmented generation. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2395–2400, 2024.
- [32] Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Lionel M. Ni, Heung-Yeung Shum, and Jian Guo. Think-on-graph: Deep and responsible reasoning of large language model on knowledge graph, 2024.
- [33] Denny Vrandečić and Markus Krötzsch. Semantic mediawiki. In John Davies, Marko Grobelnik, and Dunja Mladenić, editors, *Semantic Knowledge Management: Integrating Ontology Management, Knowledge Discovery, and Human Language Technologies*, pages 171–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
- [34] Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. Multilingual e5 text embeddings: A technical report, 2024.
- [35] Tolga Şakar and Hakan Emekci. Maximizing rag efficiency: A comparative analysis of rag methods. *Natural Language Processing*, 31(1):1–25, 2025.

Appendix

This appendix consists of supporting materials referenced throughout the thesis. It documents the ablation used to fix the retrieval parameters before the main evaluation, the decomposition and scoring rules applied in the manual evaluation, the algorithms of Document and RDF pre-processing pipelines, the differences between the agentic architecture and the original algorithm, the SPARQL queries used for neighbourhood expansion, the generation prompt shared by the knowledge-graph-based architectures, and the end-to-end retrieval procedure of KG-Derived Vector RAG.

A Code and Data Availability

The implementation of all five retrieval architectures, the pre-processing and indexing scripts, the evaluation pipeline, and the retrieval configuration ablation are available at https://github.com/ceceching-blip/Thesis_Repository. The repository documents the setup, build order, and commands required to reproduce the experiments. Since the Vienna History Wiki is a live resource, the dated data snapshot, the benchmark, and the generated answers are archived in Zenodo. The code is archived at <https://doi.org/10.5281/zenodo.21680028> and the dataset at <https://doi.org/10.5281/zenodo.21680523>.

B Retrieval Configuration Ablation

The knowledge graph pipelines are constructed by three retrieval parameters, namely the number of seed triples retrieved from the vector store, the number of neighbourhood triples kept per seed hit at the first hop (`hop1_limit`), and the number kept per newly discovered entity at the second hop (`hop2_limit`). The values used in the main evaluation are not chosen randomly. They are selected from the grid search reported in this appendix under a fixed rule stated below.

The purpose of this ablation is to fix these parameters before the main evaluation is run. As a result, the comparison between architectures is not based on random choice of retrieval depth or breadth. The measurement used here is deterministic and retrieval-side only, no generation is involved and no answer-quality metric is used to make the selection. This keeps the parameter choice independent of the system and metrics it will later be used to evaluate.

B.1 Grid and Measurement

The grid varies `top_k` over (15, 20, 40, 60, 90) for the document family and over (20, 40, 60, 90) for the triple-based families. The values of `hop1_limit` and `hop2_limit` are drawn from (0, 5, 10) with a non-zero second hop limit permitted only when the first hop limit is also non-zero. The resulting configurations fall into four families, distinguished by how far they expand the seed set:

- *konly* configurations perform no expansion at all, with `hop1_limit = 0`. These correspond to the KG-Derived Verbalised Triple RAG, which retrieves triples and stops.
- *hop1* configurations perform a single hop of neighbourhood expansion (`hop1_limit > 0`, `hop2_limit = 0`). These correspond to GraphRAG 1-hop.
- *full* configurations perform two hops (`hop1_limit > 0`, `hop2_limit > 0`). These correspond to GraphRAG 2-hop.
- *doc* configurations retrieve document chunks rather than triples and perform no graph traversal. They are included so that the document baseline can be selected using the same token-admissibility and context-recall rule as the triple-based families.

Each configuration is run over the full benchmark with no token limit imposed, so that the retrieved context is whatever the parameters produce instead of whatever fits a cap. This is intended since the point of the ablation is to observe the natural size and coverage of each configuration before the 6,000 token budget of the live pipeline is applied, so that the budget can then be treated as a constraint the chosen configuration satisfies instead of silently reshapes it.

Two quantities are recorded for each configuration, averaged over all benchmark questions. Context recall is the share of a question’s gold signals that appear in the retrieved context, computed as the number of gold signals found divided by the number expected. Prompt tokens is the mean length of the retrieved context in tokens. The ablation also records several other quantities, including hit rates at fixed cut offs and a token efficiency ratio, but these play no part in the selection rule and are not used here.

B.2 Selection Rule

The live pipelines enforce a hard context cap of 6,000 tokens. A configuration whose mean context length is close to that cap will exceed it on the individual

questions with large neighbourhoods and those questions will be truncated. To leave a margin against this, the selection rule admits only configurations whose mean prompt length is below 5,000 tokens, one thousand tokens below the cap. Among the admitted configurations, the one with the highest recall score is chosen.

The rule is applied independently within each expansion family, so that the GraphRAG 1-hop draws its parameters from the hop1 configurations and GraphRAG 2 hop from the full configurations. This is what makes the two variants comparable in the main evaluation. Each is the best its family can do within the same token budget. Hence, their difference shows expansion depth under a shared constraint instead of one version given more room than the other.

Context recall is used as the selection criterion instead of any answer-quality measure because it is the quantity the parameters most directly control and because it is generator independent. A configuration that places more of the required evidence in the context is the better retrieval configuration regardless of how any particular generator then uses that evidence, and choosing on this basis keeps the parameter selection separate from the architecture and provider comparisons that follow.

B.3 Results

This table below shows every configuration in the grid, ordered by context recall. The two right columns mark whether each configuration falls under the 5,000 token admission threshold and among those that do, which is selected for its family.

Table 15: Ablation Grid of Retrieval Configurations Ordered by Context Recall

Family	Config	top_k	h1	h2	Ctx. rec.	Tokens	Adm.	Sel.
full	k90 h1=10 h2=10	90	10	10	0.950	44 166		
full	k90 h1=10 h2=5	90	10	5	0.949	29 834		
hop1	k90 h1=10	90	10	0	0.944	12 119		
full	k90 h1=5 h2=10	90	5	10	0.944	25 948		
full	k90 h1=5 h2=5	90	5	5	0.943	18 383		
full	k60 h1=10 h2=10	60	10	10	0.938	31 741		
hop1	k90 h1=5	90	5	0	0.934	9 559		
full	k60 h1=10 h2=5	60	10	5	0.932	21 275		
full	k60 h1=5 h2=10	60	5	10	0.931	18 043		
full	k60 h1=5 h2=5	60	5	5	0.927	12 870		
hop1	k60 h1=10	60	10	0	0.921	8 395		
full	k40 h1=10 h2=10	40	10	10	0.916	22 760		
hop1	k60 h1=5	60	5	0	0.914	6 636		
full	k40 h1=10 h2=5	40	10	5	0.912	15 149		
full	k40 h1=5 h2=10	40	5	10	0.907	12 440		
full	k40 h1=5 h2=5	40	5	5	0.904	8 919		
hop1	k40 h1=10	40	10	0	0.896	5 896		
konly	k90	90	0	0	0.892	4 919	✓	✓
hop1	k40 h1=5	40	5	0	0.884	4 647	✓	✓
konly	k60	60	0	0	0.868	3 540	✓	
doc	k90	90	-	-	0.866	23 060		
doc	k60	60	-	-	0.864	15 629		
doc	k40	40	-	-	0.854	10 600		
doc	k20	20	-	-	0.846	5 597		
doc	k15	15	-	-	0.845	4 323	✓	✓
hop1	k20 h1=10	20	10	0	0.841	3 397	✓	
full	k20 h1=5 h2=10	20	5	10	0.836	6 675		
konly	k40	40	0	0	0.835	2 537	✓	
full	k20 h1=5 h2=5	20	5	5	0.828	4 914	✓	✓
hop1	k20 h1=5	20	5	0	0.803	2 648	✓	
konly	k20	20	0	0	0.754	1 532	✓	

B.3.1 The Chosen Configuration

Under the rule, GraphRAG 1 hop takes $k = 40$, $\text{hop1} = 5$, at context recall at 0.884 and mean length 4,647 tokens. This is the highest recall hop1 configuration that fits the budget. The two hop1 configurations with higher recall, $k=60$, $\text{hop1} = 10$, at 0.921 and $k=60$, $\text{hop1} = 5$ at 0.914 both exceed the threshold at 8,395 and 6,636 tokens.

GraphRAG 2 hop takes $k=20$, $\text{hop1}=5$, and $\text{hop2}=5$ at context recall 0.828 and mean length 4,914 tokens. It is the only two-hop configuration in the entire grid that falls under the threshold. Every other configuration

exceeds it with the closest being $k=20$, $\text{hop1}=5$, and $\text{hop2}=10$ at 6,675 tokens. This is the concrete form of the argument made in the evaluation chapter where it states that two expansion hops are expensive in context length and fitting them within the budget forces the seed count down to 20, half of what the one-hop configuration can afford.

The two selected configurations therefore differ in seed breadth, 40 against 20, due to the shared token budget. The one-hop pipeline spends its budget on more seeds while the two-hop pipeline spends part of it on a second hop and must retrieve fewer seeds to stay within the same cap. This is the sense in which the two variants are budget-matched instead of seed-matched.

B.3.2 KG-Derived Vector RAG Configuration and Document Vector RAG configuration

The document vector RAG baseline and the KG-derived verbalised triple RAG both retrieve in a single pass with no graph expansion. Hence, their only retrieval parameter is the seed count. Both are fixed directly from the table under the same rule.

For the document baseline, $k=15$ places 0.845 of the required gold signals in a context of only 4,323 tokens. Raising the document seed count to $k=90$ adds almost nothing to recall of 0.866 while inflating the context more than fivefold to 23,060. Hence, the larger setting is rejected and the $k=15$ is kept.

For the KG-derived pipeline, which retrieves single fact triples rather than paragraph-length chunks, a far wider seed set is affordable. At $k=90$ it reaches a context recall of 0.892, the highest of any no-expansion configuration, while its mean context of 4,919 tokens still fits within the budget. This is why the KG-derived vector RAG is deployed at $k=90$. It is the point at which flat triple retrieval maximises coverage without exceeding the token cap.

The table above also shows why the two GraphRAG versions cannot share this seed count. At $k=90$ a single hop of expansion already costs between 9,559 and 12,119 tokens, and two hops between 18,383 and 44,166 tokens. All of which are far beyond the 6000 tokens. Because each hop multiplies the material attached to every seed, the expanding pipelines must retrieve far fewer seeds to remain within the same cap, which is the reason GraphRAG 1 hop and GraphRAG 2 hop are configured with reduced seed counts.

B.3.3 Observations from the Grid

Three patterns in the grid table show up. First, the context recall rises with every parameter, but the token cost of each rises faster. The highest-recall configuration in the grid reaches context recall of around 0.95 but consumes

44,166 tokens, more than seven times the live budget. The gain over the chosen one-hop configuration is 0.066 of context recall for roughly seven times the context length. The steep part of this trade-off is quite early in the grid, and almost all of the achievable coverage is already available within the budget.

Second, within a fixed token cost, seed breadth and expansion depth are close substitutes for context recall. The chosen one-hop configuration and the chosen two-hop configuration occupy similar length while the one-hop configuration achieves higher recall. Within the same budget, spending on more seeds returns more coverage instead of spending on the second hop.

Third, the second hop contributes little coverage for the cost it pays. Comparing the two-hop configurations with their one-hop counterparts at equal `top_k`, and `hop1_limit`, the addition of a second hop raises context recall by a few hundredths while multiplying context length several times over. The gold signals a question needs are, on this corpus, largely reachable within one hop of the retrieved seeds. Hence, the second hop mostly adds neighbourhood text that the question does not require.

Applying the selection rule to this grid gives the parameters used in the main evaluation. This results in `k=40` and `hop1 = 5` for GraphRAG 1 hop and `k=20`, `hop=5`, and `hop2=5` for GraphRAG 2 hop. The KG derived Vector RAG pipeline, which performs no expansion and whose context length stays below the cap, is set at `k=90` which has the highest context recall of its family. Furthermore, the document vector RAG baseline whose chunks are long enough, is set to `k = 15`.

C Evaluation Scoring Details

This appendix documents the rules applied in the manual statement-level evaluation described in manual evaluation section. The atomic gold claim rules can be seen in the table below.

Table 16: Decomposition Rules for Atomic Gold Claim

Gold-answer type	Decomposition rule
Person, organisation, role, award, place, spouse, or list member	One atomic claim per required item.
Full title	The complete title is one item, matched as a whole (e.g. <i>Kaiser von Österreich</i>).
Full date	The complete date is one item, matched as a whole.
Grave plot	Three claims: <i>Gruppe</i> , <i>Reihe</i> , and <i>Nummer</i> .
Aggregation list	One claim per required member, plus one claim for the count when the gold explicitly states it.
Additional explanation or note	Scored as a false positive when it introduces factual information not supported by the gold answer.

D Pre-Processing and Data Preparation

This appendix contains the algorithms behind the two preparation pipelines described in the implementation section. The document pipeline extracts and cleans the MediaWiki dump and divides the resulting articles into retrieval units, while the RDF pipeline cleans the exported triples and converts them into the verbalised form used for embedding. Each subsection gives the procedure in full, including the handling of the edge cases.

D.1 Document Pre-Processing

Figure 17 summarises the complete pre-processing workflow. The XML dump is streamed one page at a time using ITERPARSE, after which pages outside the retained namespaces are discarded. For each remaining page, the title and raw revision text are extracted, redirects and empty pages are removed, and the wikitext-cleaning steps described above are applied. The output of this stage is a structured table where each row has a document identifier (`doc_id`), the page title (`title`), the canonical wiki URL (`url`), and the cleaned plain-text body (`text`). The URL is constructed by concatenating the base address (<https://www.geschichtewiki.wien.gv.at/>) with the title, with spaces replaced by underscores.

Figure 17: Document Chunks Pre-Processing

Require: MediaWiki XML dump

```
1: records  $\leftarrow$  []
2: for each page in ITERPARSE(XML) do
3:   ns  $\leftarrow$  page.namespace
4:   if ns  $\notin$  {0, 6, 14} then
5:     continue
6:   end if
7:   title  $\leftarrow$  page.title
8:   text  $\leftarrow$  page.revision.text
9:   if text is empty or is a redirect then
10:    continue
11:  end if
12:  remove <ref> blocks and HTML/XML tags from text
13:  replace wiki links with their displayed text
14:  resolve HTML entities
15:  normalise whitespace
16:  if text  $\neq$   $\emptyset$  then
17:    construct doc_id, title, url, and text
18:    append record to records
19:  end if
20:  free page from memory
21: end for
22: return records
```

D.2 Document Chunk Construction

This subsection gives the chunking procedure described in the implementation section.

Figure 18: Document chunk construction

Require: Cleaned documents, chunk size $C = 256$, sentence overlap $O = 1$

```

1:  $records \leftarrow []$ 
2: for each document  $d$  do
3:    $S \leftarrow \text{GERMANSENTENCETOKENIZE}(d.text)$ 
4:    $chunks \leftarrow []$ ;  $current \leftarrow []$ ;  $n \leftarrow 0$ 
5:   for each sentence  $s$  in  $S$  do
6:      $t \leftarrow \text{TOKENCOUNT}(s)$ 
7:     if  $t > C$  then
8:       split  $s$  into token slices of at most  $C$ 
9:       append the decoded slices to  $chunks$ 
10:      continue
11:    end if
12:    if  $n + t > C$  and  $current \neq []$  then
13:      append  $\text{JOIN}(current)$  to  $chunks$ 
14:       $current \leftarrow$  final  $O$  sentences of  $current$ 
15:       $n \leftarrow \sum_{s' \in current} \text{TOKENCOUNT}(s')$ 
16:    end if
17:    append  $s$  to  $current$ ;  $n \leftarrow n + t$ 
18:  end for
19:  if  $current \neq []$  then
20:    append  $\text{JOIN}(current)$  to  $chunks$ 
21:  end if
22:  for each chunk  $c_i$  in  $chunks$  do
23:    prepend title and section position  $i/|chunks|$  to  $c_i$ 
24:    construct record with  $c_i$  and its document metadata
25:    append record to  $records$ 
26:  end for
27: end for
28: return  $records$ 

```

For each incoming sentence, three cases are possible:

- **The sentence alone exceeds 256 tokens:** This triggers the hard-slice fallback. Any sentences already in the buffer are first saved as a chunk. Then the long sentence is tokenised into a list of token IDs, which is sliced into fixed-length pieces of 256 IDs each. Each piece is decoded back to text and saved directly as its own chunk. The buffer is then reset and the loop goes on.
- **Adding the sentence would overflow the buffer.** The current buffer is saved as a chunk. The last sentence from that buffer is kept and carried forward into the new buffer as the one-sentence overlap. The token count is recalculated to reflect only what remains in the new buffer.
- **The sentence fits.** It is simply appended to the buffer and the token counter is updated.

After the loop finishes, any sentences remaining in the buffer that never triggered a save are written out as the final chunk.

D.3 RDF Pre-Processing Algorithm

Figure 19 summarises the cleaning procedure.

Figure 19: RDF corpus pre-processing

0.5em

Require: Vienna History Wiki RDF/XML export

```
1: load RDF graph
2: construct namespace map from triples with predicate swivt:wikiNamespace
3: for each triple  $(s, p, o)$  in the graph do
4:   if  $s$  is empty or  $p \in \text{SKIP\_PREDICATES}$  then
5:     continue
6:   end if
7:   if  $p$  ends with -23aux then
8:     continue
9:   end if
10:  if  $s$  or  $o$  contains QUERY then
11:    continue
12:  end if
13:  if  $s$  has a recorded namespace and  $\text{namespace}(s) \notin \{0, 6, 14, 102\}$  then
14:    continue
15:  end if
16:  if  $o$  is a literal then
17:    if  $o$  is empty or begins with a redirect marker then
18:      continue
19:    end if
20:    normalise whitespace in  $o$ , preserving language tag and datatype
21:  end if
22:  write  $(s, p, o)$  to the cleaned N-Triples output
23: end for
24: return cleaned N-Triples file
```

Figure 19 applies the namespace, predicate, URI, and literal filters sequentially to every RDF triple. Only triples that pass all filtering conditions are written to the cleaned graph. This cleaned N-Triples representation is subsequently used both to construct the QLever graph index and as the input to the triple-verbalisation procedure.

D.4 RDF Triple Verbalisation Algorithm

This subsection gives the verbalisation procedure described in verbalisation subsection. The first pass collects the context of each subobject, recording its master page and the other predicates stored under it, and the second pass produces one record for each kept triple. Triples whose subject is not subobject are verbalised directly as subject, predicate, and object, while triples that belong to a subobject are verbalised together with their sibling predicates, so that a fact separated from its entity by the Semantic MediaWiki subobject structure is still readable on its own.

Figure 20: VerbaliseFile(input_nt, output_jsonl)

```

1:  $ctx \leftarrow \{\}$  ▷ Pass 1: per-subobject context, keyed by subobject URI
2: for each parsed triple  $(s, p, o)$  in  $input\_nt$  do
3:   if IsSubobjectURI( $s$ ) then
4:      $pred \leftarrow \text{SimplifyURI}(p)$ 
5:     if  $pred \in \text{SkipPredicates}$  then continue
6:   end if
7:   if  $pred = \text{"masterPage"}$  then
8:     set  $ctx[s].masterPage \leftarrow \text{SimplifyObject}(o)$  ▷ first-wins
9:     if  $o$  is a URI then
10:      set  $ctx[s].masterURI \leftarrow o$  ▷ raw master URI, first-wins
11:    end if
12:  else
13:    append  $\text{SimplifyObject}(o)$  to  $ctx[s][pred]$  ▷ list, property may repeat
14:  end if
15: end if
16: end for
17: for each parsed triple  $(s, p, o)$  in  $input\_nt$  do ▷ Pass 2: one record per non-skipped triple
18:    $pred \leftarrow \text{SimplifyURI}(p)$ ;  $val \leftarrow \text{SimplifyObject}(o)$ 
19:   if IsSubobjectURI( $s$ ) then
20:     if  $pred \in \text{SkipPredicates}$  then continue
21:   end if
22:    $master \leftarrow ctx[s].masterPage$  or  $\text{SimplifyURI}(s)$ 
23:    $master\_uri \leftarrow ctx[s].masterURI$  or  $\text{" "}$  ▷ empty if none recorded
24:    $siblings \leftarrow$  every other content predicate in  $ctx[s]$  (excluding  $masterPage$ ),
25:   removing  $val$  from  $pred$ 's own value list
26:   if  $siblings \neq \emptyset$  and  $pred \neq \text{"masterPage"}$  then
27:      $text \leftarrow \text{"master [siblings] pred val"}$ 
28:   else
29:      $text \leftarrow \text{"master pred val"}$ 
30:   end if
31: else
32:   if  $pred$  contains  $\text{"Has\_subobject"}/\text{"hasSubobject"}$  or  $\text{"Processing\_error"}$  then
33:     continue
34:   end if
35:    $master\_uri \leftarrow \text{" "}$ 
36:    $text \leftarrow \text{"SimplifyURI}(s) pred val"$ 
37: end if
38: write  $\{subject\_uri, master\_uri, subject\_label,$ 
39:    $predicate\_uri, predicate\_label, object\_uri,$ 
40:    $object\_value, text\}$  to  $output\_jsonl$ 
end for

```

E Retrieval Queries

This appendix contains the SPARQL query templates used for neighbourhood expansion in the two static GraphRAG architectures. Both retrieve the edges incoming to and outgoing from a given URI. Consequently, a fact is reachable regardless of whether the entity appears as the subject or object of the triple.

E.1 One-Hop SPARQL Neighbourhood Query

The first hop expands both the subject and the object URI of each retrieved seed triple, which gives the four branches below.

Figure 21: One-Hop SPARQL Neighbourhood Query Template

```
SELECT * WHERE {  
  { SELECT ?S ?P ?O WHERE { ?S ?P <subject> . BIND (<subject> AS ?O) } LIMIT 50 }  
  UNION  
  { SELECT ?S ?P ?O WHERE { <subject> ?P ?O . BIND (<subject> AS ?S) } LIMIT 50 }  
  UNION  
  { SELECT ?S ?P ?O WHERE { ?S ?P <object> . BIND (<object> AS ?O) } LIMIT 50 }  
  UNION  
  { SELECT ?S ?P ?O WHERE { <object> ?P ?O . BIND (<object> AS ?S) } LIMIT 50 }  
}
```

E.2 Second-Hop SPARQL Neighbourhood Query

Each frontier URI is then expanded with a two-branch SPARQL query that retrieves the edges incoming to and outgoing from that URI.

Figure 22: Second-Hop SPARQL Neighbourhood Query Template

```
SELECT * WHERE {  
  { SELECT ?S ?P ?O WHERE { ?S ?P <frontier> . BIND (<frontier> AS ?O) } LIMIT 50 }  
  UNION  
  { SELECT ?S ?P ?O WHERE { <frontier> ?P ?O . BIND (<frontier> AS ?S) } LIMIT 50 }  
}
```

F Generation Prompt for KG-Based Architectures

The following prompt is used for final answer generation by KG-Derived Vector RAG, GraphRAG 1-Hop, and GraphRAG 2-Hop. The architectures use the same instructions but input the context placeholders with evidence produced by their respective retrieval procedures.

You are a research assistant for the Vienna History Wiki (GeschichteWiki). Answer only from the provided context. If the answer is not in the context, say: “I don’t know based on the indexed data.” Answer in English. Keep proper nouns in their original German form. This includes person names, such as “Karl Lueger” and “Emilie Flöge”; street and place names, such as “Mariahilfer Straße” and “Heiligenstädter Straße”; and building or institution names, such as “Rathaus” and “Stephansdom”. Translate all other words to English. For example, use “Vienna” rather than “Wien”, “mayor” rather than “Bürgermeister”, and “painter” rather than “Maler”. Be concise. Give only the specific fact requested by the question. For counting questions, count only the items explicitly present in the provided context. Do not add items from general knowledge. List each item found in the context and then state the total. The context is a top-ranked retrieval subset rather than a guaranteed-complete result set. If additional matching items may exist beyond the retrieved evidence, indicate this rather than presenting the count as certainly exhaustive. If a date in the context reads exactly “1 January”, for example, “1 January 1842”, this may represent only year-level precision in the source data rather than a confirmed exact date. State the year confidently but describe the day as uncertain, for example, “in 1842” rather than “on 1 January 1842”, unless the context corroborates the exact day elsewhere. The property “Titel” refers to academic or honorific titles, such as Dr., Professor, or Geheimrat. Official political or administrative offices are recorded separately under “jobTitle” or “Funktionsanzeige”. When the question asks about official titles, offices, roles, or positions held, prioritise facts recorded under “jobTitle” or “Funktionsanzeige” over facts recorded under “Titel”. If the question requests a description, for example, “Who was X?” or “What is Y?”, and lists the fields to state, answer using exactly those fields and nothing else. Do not add any unrequested fact type, such as titles, awards, addresses, relatives, memberships, or burial information, even when it appears in the context. Answer using short sentences. If a requested field is not present in the context, write “not found in the indexed data” for that field. If a description question does not list specific fields, state at most the following:

- for a person: occupation, birth date, birth place, titles, and awards recorded in the context;
- for a place: building type, street, district, titles, and awards recorded in the context.

Do not add background information or explanations beyond what is asked. **Question:** {query} **Context:** {context} {neighbourhood_section}

G End-to-End Algorithm of KG-Derived Vector RAG

This appendix gives the full retrieval and generation procedure of KG-Derived Vector RAG. The architecture embeds the query, retrieves the 90 most similar verbalised triples from the triple collection, and passes them to the generator without any graph traversal.

Figure 23: $\text{ASK}(query, top_k = 90)$ - Verbalised Triple RAG

```
1:  $\mathbf{q} \leftarrow \text{EMBEDQUERY}(query)$ 
2:  $hits \leftarrow \text{QDRANT.SEARCH}(collection=\text{triples}, vector=\mathbf{q}, limit=top\_k)$ 
3:  $blocks \leftarrow []$ ;  $tokens \leftarrow 0$ 
4: for each hit  $h$  in  $hits$  (descending cosine order) do
5:    $block \leftarrow \text{FORMAT}(h.rank, h.score, h.subject\_label, h.text)$ 
6:   if  $tokens + \text{TOKENCOUNT}(block) > \text{TOKEN\_BUDGET}$  then
7:     break
8:   end if
9:    $blocks.APPEND(block)$ ;  $tokens += \text{TOKENCOUNT}(block)$ 
10: end for
11:  $prompt \leftarrow system\_prompt + query + \text{JOIN}(blocks)$ 
12:  $answer \leftarrow \text{STRIPMARKDOWNBOLD}(\text{GENERATE}(prompt, temperature=0))$ 
13: return  $\{query, answer, hits, neighbourhood\_text=""\}$ 
```

H Adaptation from Think-on-Graph

The table below combines the adaptations discussed above. The changes can be divided into three groups, those forced by the data source (entity linking, subobject handling, literal normalisation, the multilingual relation hints), those made for comparability within this thesis (the shared generator, the closed-world fallback, the token-budget safety cap), and one tuning decision (the wider beam).

Table 17: Differences Between the Original Think-on-Graph and this Thesis Adaptation

Aspect	Original ToG	Agentic System
Knowledge graph	Freebase	Cleaned Vienna History Wiki RDF served by QLever
Topic entities	Provided by benchmark datasets	Extracted by the LLM and linked through URI probes, redirects, encoding variants, and disambiguation suffixes
Seed entities	Dataset-dependent topic entities	Up to five linked seed URIs per question
Beam width / depth	$N = 3, D_{\max} = 3$	width = 20, depth = 3
Composite facts	Not a central concern in Freebase	Semantic MediaWiki subobjects retained as explicit nodes and expanded through genuine RDF edges
Literals	Used as returned	Julian Day numbers converted to readable dates and integer literals normalised
Relation menu	Raw KG relations	Direct RDF relations only; technical predicates filtered and multilingual relation hints supplied in the prompt
Candidate pruning	LLM-based relation and entity pruning	Single candidates inherit relation score; multiple candidates are LLM-scored and global top 20 retained
Exhausted search	May answer using LLM’s inherent knowledge	Closed-world response based only on retained paths; refusal if no evidence exists
Aggregation questions	No special completeness instruction	Sufficiency evaluation requires matching set to appear complete
Context-size control	Beam width and depth	Beam width and depth as primary constraints; 6,000-token path budget as secondary cap