Offline open-source RAG-based AI chatbot for Finnish-language software documentation on low-end hardware

Modular model-agnostic framework and AI judge for multilingual use

Bachelor's Thesis
Degree Programme in Computer Applications
Spring 2025
Quang Luong

DP	Degree Programme in Computer Applications	
Author	Quang Luong	Year 2025
Subject	Offline open-source RAG-based AI chatbot for Finnish-language software documentation on low-end hardware	
Supervisor	Tommi Lahti	

Abstract:

This thesis addresses the usability limitations of static software documentation and the confidentiality concerns that restrict the use of public AI tools. It aims to develop and evaluate an offline, open-source AI chatbot on low-end hardware, in partnership with Triplan Oy, prioritizing confidentiality and sustainability.

The chosen solution is Retrieval-Augmented Generation (RAG), which enhances Large Language Model (LLM) responses by dynamically retrieving relevant documents to ground the generated answers. The project employed a rapid prototyping approach, following the Design Science Research Process, where iterative design and continuous client feedback guided development and refinement. Key open-source technologies selected for the prototype included Ollama for efficient local LLM inference, FAISS for high-performance vector similarity search, and llama-index as the orchestration framework for data indexing and retrieval. Data preparation was critical for the Finnish documentation, focusing on proper UTF-8 encoding to preserve special characters (ä, ö, å) and implementing a section-based chunking strategy tailored to the document structure for optimal information retrieval.

The prototype was rigorously evaluated for both system latency and response quality. Median retrieval latencies stayed under one second even on low-end CPU, enabling fast, interactive display of relevant document chunks. GPU acceleration significantly reduced both semantic retrieval and response generation times. LLM size was identified as a dominant factor affecting latency on CPU, while embedding model choice had negligible impact. For response quality, the system achieved an 80% correctness rate overall, and 86% for aligned, in-scope queries, as assessed by an AI judge using the RAG Triad (Groundedness, Context Relevance, and Answer Relevance). The snowflake-arctic-embed2 embedding model also demonstrated numerically superior Recall@3 and statistically superior Recall@1 compared to bge-m3. However, the 1B LLM exhibited limitations when handling tricky or off-topic queries, leading to hallucinations and inconsistent abstention behavior, highlighting the quality trade-off of smaller models.

This work successfully demonstrates the feasibility of deploying secure, offline RAG-based AI chatbots in resource-constrained, language-specific environments. While the current prototype serves as a proof of concept with acknowledged limitations (e.g., restricted model diversity, single source document, basic error handling), it provides a flexible framework for future development. Subsequent efforts should focus on enhancing robustness, diversifying evaluation, integrating human-in-the-loop feedback, and refining prompt engineering for improved reliability and broader applicability.

Keywords: AI chatbot, AI judge, Retrieval-Augmented Generation (RAG), Model-agnostic, Multilingual, Modular, Low-end hardware, Open-source, Offline, Finnish language, Software documentation, Semantic Search, Natural language processing (NLP), Large Language Model (LLM).

Pages: 51 pages and appendices 20 pages

Glossary

AI chatbot: Software application that uses artificial intelligence to simulate human-like conversations.

Chunking: Process of splitting large text into smaller, manageable segments for analysis or embedding.

Command-Line Interface (CLI): Text-based interface that allows users to interact with a computer system by typing commands.

Context window: Maximum number of tokens an LLM can process at once, which limits how much text it can consider in a single interaction.

Cosine similarity: Measure of similarity between two vectors based on the cosine of the angle between them.

Embedding: Numerical encoding of text into high-dimensional space that preserves contextual meaning

Embedding model: Machine learning model that transforms text into numerical vector representations that capture semantic meaning.

Euclidean distance: A measure of straight-line distance between two vectors in space.

FAISS: High-performance library for efficient similarity search and clustering of dense vectors, often used in vector databases.

Fine-tuning: The process of continuing the training of a pre-trained language model on a task-specific dataset to adapt it to a particular domain or objective.

Hallucination: AI-generated information that is factually incorrect or not supported by the provided context.

Information retrieval: Locating relevant information or documents from a large dataset based on a query.

Large language model (LLM): Deep learning model trained on vast amounts of text data to understand and generate human language.

Model-agnostic: Tools or methods that can operate across multiple models without being restricted to a specific one.

llamaIndex: Framework that enables structured data indexing and retrieval pipelines for use with large language models.

Natural Language Processing (NLP): Field of AI focused on enabling computers to understand, interpret, and generate human language.

Ollama: Local deployment platform for running LLMs efficiently on personal or offline environments.

Prompt engineering: The practice of crafting effective prompts to guide language models toward more accurate and useful outputs.

RAG Triad: Set of three criteria - Groundedness, Context Relevance, and Answer Relevance - used to evaluate RAG responses.

Retrieval-Augmented Generation (RAG): System that enhances LLM responses by retrieving relevant documents to ground generated answers.

Semantic search: Search technique that uses vector similarity to find results based on meaning rather than exact keyword matching.

Tokenization: Process of dividing text into smaller units called tokens, which are the inputs for language models.

Vector database: A specialized database designed to store and search high-dimensional vector data for tasks like semantic retrieval.

Vector representation: see Embedding.

Table of Contents

1 Introduction 1

2 Background 2

2.1 AI chatbot fundamentals 2

2.2 Natural language processing (NLP) foundations and Finnish considerations 3

2.2.1 Natural language processing (NLP) fundamentals 4

2.2.2 Text processing and vector representation for natural language understanding 5

2.2.3 Semantic similarity and information retrieval 9

2.2.4 Large language models and prompt engineering for text generation 11

2.3 Retrieval-Augmented Generation (RAG) 13

2.3.1 Rationale for choosing RAG 14

2.3.2 RAG architecture 14

2.4 Ethical and sustainability considerations 16

3 Methodology 17

3.1 Functional and non-functional requirements 17

3.2 Prototype development 18

3.2.1 Technology and RAG component selection and parameter configuration 19

3.2.2 Environment, project directory, and configuration settings 23

3.2.3 Jupyter notebooks and python files 25

3.2.4 Data pipeline notebook 26

3.2.5 RAG pipeline notebook 28

3.3 Prototype evaluation 29

3.3.1 Evaluation metrics and analysis 30

3.3.2 Batch inference module and AI judge notebooks 33

4 Result and Discussion 34

4.1 Development of the RAG-based AI chatbot 34

4.1.1 Key considerations in applying rapid prototyping for chatbot development and evaluation (Research question 1) 35

4.1.2 Data preparation for Finnish-language software documentation in the RAG prototype (Research question 2) 36

4.2 Evaluation of the RAG-based AI chatbot 38

4.2.1 Evaluating the latency of the RAG prototype (Research question 3) 38

4.2.2 Evaluating the quality of the RAG prototype (Research question 4) 42

4.3 Limitations and improvement options 48

5 Conclusions 49
6 Summary 51
References 52

Figures

Figure 1. Types of chatbots 3
Figure 2. Modalities of the human language with a focus on the written form 3
Figure 3. Vector representation for the phrase “Hämeen linna on Hämeenlinnassa” 7
Figure 4. Euclidean distance between two vectors in 2-dimensional space 9
Figure 5. Cosine similarity of two vectors in a 2-dimensional space 10
Figure 6. RAG question answering mechanism 15
Figure 7. RAG system essential component architecture 15
Figure 8. Project directory structure for the RAG system 23
Figure 9. Environment settings 24
Figure 10. The four notebooks, their inputs and outputs 26
Figure 11. Steps in data preparation 26
Figure 12. Semantic Retrieval Latency comparison between CPU and GPU 40
Figure 13. Response Generation Latency comparison between CPU and GPU 42
Figure 14. Effect of LLM size on Response Generation Latency (CPU) 42
Figure 15. Recall@3 for the two embedding models: snowflake-arctic-embed2 and bge-m3 43
Figure 16. Distribution of RAG Triad scores 45
Figure 17. Correlation heatmap of the RAG Triad scores 45
Figure 18. Distribution of RAG outcome categories as classified using RAG Triad scores 46
Figure 19. Distribution of RAG output categories by question type 47

Tables

Table 1. Main NLP and non-NLP techniques used in the thesis. 4
Table 2. Phases of natural language understanding 5
Table 3. Tokenization comparison across languages 7
Table 4. RAG components and their roles 16
Table 5. Functional and non-functional requirements 17
Table 6. Technology and RAG components 19
Table 7. LLMs pre-selection for the RAG prototype 22

Table 8. Key system parameter configuration 23
Table 9. Questionnaire for RAG-based AI chatbot evaluation 30
Table 10. Metrics for RAG-based AI chatbot evaluation 30
Table 11. Variables used in quality analysis 31
Table 12. Failure points related to Retrieval-Augmented Generation 32
Table 13. RAG outcome heuristic categories and criteria 33
Table 14. Semantic Retrieval Latency measurements across four configurations 39
Table 15. Semantic Retrieval Latency comparison across four configurations 39
Table 16. Generation Response Latency measurements across five configurations 41
Table 17. Generation Response Latency comparison across configurations 41
Table 18. Recall@3 for the two embedding models: snowflake-arctic-embed2 and bge-m3 43

Program codes

Program code 1. Python scripts to load the project root directory and configuration 25
Program code 2. Flags to preserve Finnish character encoding 37
Program code 3. Configuration file 59
Program code 4. Importing libraries and setting configuration 61
Program code 5. PDF to TXT converter 63
Program code 6. DOCX to TXT converter 64
Program code 7. Blank lines remover 65
Program code 8. Page separator, header, and footer remover 65
Program code 9. Chunker with regular expressions 66
Program code 10. Functions to save chunks to TXT and JSON for review 67
Program code 11. Embedding vector dimension computing 68
Program code 12. Functions for embedding and saving FAISS index to vector store 69
Program code 13. Local Ollama server status checker 70
Program code 14. Main function to build vector store 71
Program code 15. Embedding and large language models setter 71
Program code 16. Vector store loader 71
Program code 17. Checking the embedding model and dimension matches 72
Program code 18. Query engine builder 73
Program code 19. Chunk Retriever and displayer class 73
Program code 20. Prompt customization function 74

Program code 21. Building the context from retrieved chunks 74
Program code 22. Generating the LLM response 74
Program code 23. Main RAG function 75
Program code 24. Main function adaptation for the batch inference module 77
Program code 25. Modified model setter for the AI judge 77
Program code 26. Modified prompt customizing function for the AI judge 78
Program code 27. Modified response generating function for the AI judge 78
Program code 28. Main function adaptation for the AI judge 79
Program code 29. RAG evaluation postprocessing functions 79

Appendices

Appendix 1. Data management plan 58
Appendix 2. Program codes 59

1 Introduction

Software documentation is a critical resource for developers to understand and utilize software technologies (Nassif & Robillard, 2023). Traditional static formats are associated with significant usability limitations, due to the lack of effective navigation tools and powerful search functions (Meng et al., 2019). Difficulty finding the needed information may negatively impact user experience and reduce productivity (Smith, 2022).

The constraints of static software documentation highlight a clear need for improvement. This thesis addresses this challenge in partnership with Triplan Oy, a Finnish software company focusing on document and case-handling solutions and archiving systems for public administrations (Triplan Oy, n.d.). The objectives of this work are to develop and evaluate a chatbot prototype based on Retrieval-Augmented Generation (RAG) that assists users by conversing in Finnish and providing access to Finnish-language software documentation using open-source components. With a focus on confidentiality and sustainability, the chatbot is specifically required to operate offline and on low-end hardware. This thesis is guided by the following research questions:

1. What are the key considerations in using rapid prototyping for developing and evaluating the chatbot described in the objectives?
2. What data preparation practices are critical for Finnish-language software documentation for this RAG prototype?
3. What are the latencies for vector store building, information retrieval, and response generation observed in the RAG system, as measured by Vector Store Building Latency, Semantic Retrieval Latency, and Response Generation Latency, under varying system and configuration conditions?
4. What levels of RAG output quality are observed, based on Recall@3, and the LLM-evaluated Groundedness, Context Relevance, and Answer Relevance with human oversight?

By addressing these questions, this thesis seeks to advance the application of artificial intelligence and system optimization for documentation usability in settings where privacy, language specificity, and resource efficiency are critical. The resulting prototype is intended to serve as both a proof of concept and a foundation for future development in similar contexts.

2 Background

Software documentation has been extensively studied to improve its quality and usability (Raglianti et al., 2023). It has been observed that developers prefer an interactive software document format over a static one (Nassif & Robillard, 2023). In case programmers need help to solve a problem, they tend to prefer information sources outside the official software documentation (Meng et al., 2018). However, public forums such as Stackoverflow (Stackoverflow, n.d.), GitHub Discussions (GitHub, n.d.), or AI tools like Google NotebookLM (Google NotebookLM, n.d.) or Microsoft Copilot (Microsoft Copilot, n.d.) are restricted from use with confidential documentation, leading to increased helpdesk use.

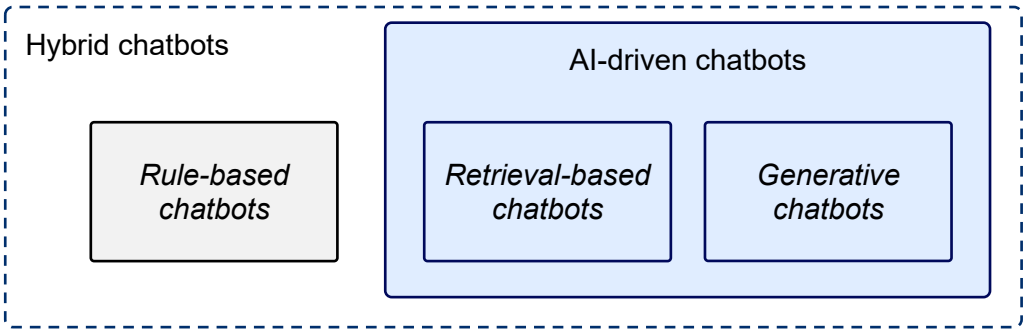
This limitation highlights the need for solutions that provide private, secure, and efficient access to software documentation. To address this gap, this thesis explores the development and evaluation of an offline Retrieval-Augmented Generation (RAG)-based AI chatbot capable of delivering interactive, context-aware support based solely on internal documentation sources.

The development and evaluation of the proposed RAG-based AI chatbot prototype starts with a comprehensive review of the conceptual framework. This includes an overview of AI chatbot fundamentals; key natural language processing (NLP) concepts relevant to RAG and the Finnish language; architecture and tools for RAG systems; optimization strategies for low-end hardware; considerations for data protection and management; RAG evaluation fundamentals; together with ethical and sustainability concerns.

2.1 AI chatbot fundamentals

Chatbots are software applications designed to mimic human conversation. Based on their complexity, chatbot can be classified into three main types, as visualized in Figure 1, along with hybrid chatbots that combine features of more than one type. Rule-based chatbots, following predefined rules, are typically used for simple tasks such as answering FAQs or providing basic information. AI-driven chatbots, using Natural Language Processing (NLP) techniques to converse, include retrieval-based and generative chatbots. Retrieval-based chatbots use deep learning techniques, such as embedding-based similarity and ranking, to select the best matching response from a predefined set. Generative chatbots, the most advanced type, use language models to generate answers in natural language and allow for more flexible and diverse interactions. (Cuantum Technologies LLC, 2025)

Figure 1. Types of chatbots

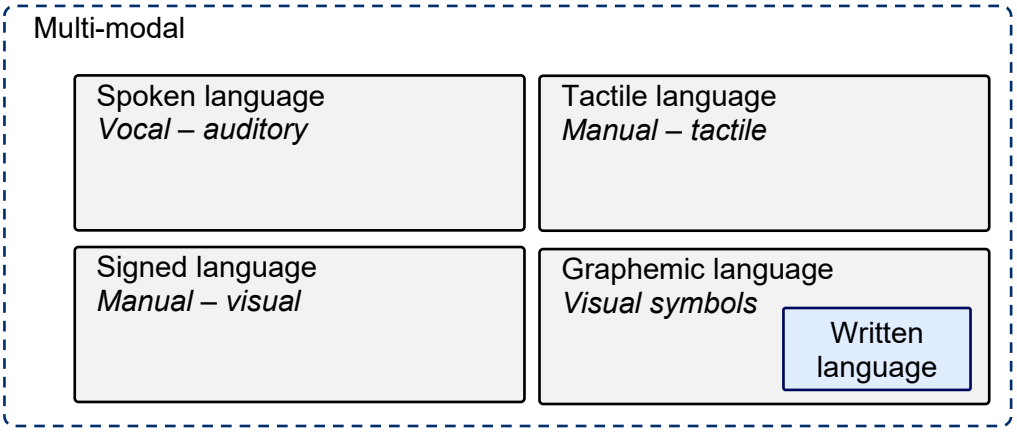


This thesis, aimed at developing an AI-powered chatbot, adopts a practical hybrid approach to leverage the strengths of different chatbot techniques. Rule-based elements can be used for simple tasks such as handling exit commands, displaying customized messages, disclaimers, and warnings. They can also be applied to minimize the instruction portion of the prompt to allocate more room for the retrieved context and user query ("prioritizing the token budget"). To achieve this, a placeholder is used and later replaced with a standard message when the retrieved context is not relevant to the query.

2.2 Natural language processing (NLP) foundations and Finnish considerations

Human language is naturally multi-modal with different modalities: spoken, signed, tactile, and graphemic languages (Doner, 2025). This thesis, aimed at chatbot development, focuses specifically on the written form of graphemic language, as highlighted in Figure 2, and adopts terminology and analytical frameworks relevant to this particular modality. Considerations about the Finnish language are discussed where appropriate.

Figure 2. Modalities of the human language with a focus on the written form



2.2.1 Natural language processing (NLP) fundamentals

Natural language processing (NLP), a subfield of artificial intelligence, seeks to enable computers to understand, interpret, and generate human languages, allowing the use of natural language in human-computer interaction (Cuantum Technologies LLC, 2025). NLP can be divided into natural language understanding (NLU) and natural language generation (NLG), covering both directions of language processing (IBM, n.d.-a). NLP capabilities are applied in Search Engine, Machine Translation, Sentiment Analysis, Text Summarization, Virtual Assistants, and Chatbots, etc. (Cuantum Technologies LLC, 2025).

This work involves some NLP techniques essential to the development and evaluation of a RAG-based AI chatbot, including text preprocessing, text processing, text embedding, similarity scoring, information retrieval, text generation, and LLM-based evaluation. These techniques and relevant non-NLP techniques are summarized in Table 1, by order of their sequence in the pipelines, to guide the discussion topic selection in this thesis.

Table 1. Main NLP and non-NLP techniques used in the thesis.

		NLP technique	*Non-NLP technique*
RAG development	*Text preprocessing*		✓
	Text processing	✓	
	Chunking *		✓
	Text embedding	✓	
	Semantic similarity	✓	
	Information retrieval **	✓	
	Prompt engineering	✓	
	Text generation	✓	
RAG evaluation	*Recall@k*		✓
	Retrieval time		✓
	Generation time		✓
	LLM-based evaluation	✓	
Note, in this thesis: ** Chunking during data preparation using regular expressions and string processing techniques, different from syntactic chunking which is a part of syntactic analysis.* *** Information retrieval using semantic search, different than statistical techniques such as TF-IDF or BM25.*			

This section focuses on essential NLP techniques. While relevant non-NLP techniques are also critical to the thesis objectives, they are out of the scope of this NLP section and will be addressed in sections dedicated to each topic, respectively.

2.2.2 Text processing and vector representation for natural language understanding

NLP uses different techniques to help computers understand the human language. Natural language understanding consists of different phases, illustrated in Table 2, including text processing, lexical, syntactic, semantic, discourse, and pragmatic analysis. (Cuantum Technologies LLC, 2025; Jurafsky & Martin, 2025)

Table 2. Phases of natural language understanding

Phase	**Purpose**	**Typical techniques / tasks**
Text Processing	Transforming raw text into a standardized format for analysis	Sentence segmentation Text normalization: *case folding, stemming, lemmatization, text cleaning* Tokenization: w*ord, subword*
Lexical analysis	Understanding word formation	Part-of-speech (POS) tagging
Syntactic analysis	Understanding structure and grammar	Parsing: c*onstituent, dependency* Syntactic chunking
Semantic analysis	Understanding the meaning of words and texts in their context	Named entity recognition (NER) Semantic role labeling Word sense disambiguation
Discourse analysis	Understanding coherence across sentences or dialogue turns	Coherence Relations Coreference resolution
Pragmatic analysis	Understanding the intended meaning and situational context beyond the literal content	Sentiment analysis Intent recognition

Text processing is the first and foundational phase, which prepares and structures raw text into a standardized format for subsequent analysis. Typical text processing tasks include sentence segmentation, text normalization, and tokenization. (Cuantum Technologies LLC, 2025; Jurafsky & Martin, 2025)

Sentence segmentation splits text into sentences, using punctuation. While question mark and exclamation point are relatively clear markers of sentence boundaries, period is more ambiguous since it is used both in abbreviations and as sentence ending. Determining the period's role requires the use of abbreviation dictionaries. (Jurafsky & Martin, 2025)

Text normalization includes common tasks such as case folding, stemming, lemmatization, and text cleaning. Case folding transforms all characters into lower case. This technique is selectively used in modern systems, since preserving case is important for word meaning, and necessary for named entity labelling and recognition. Stemming reduces words to their root form; this technique is less commonly used in modern systems due to its errors of both over- and under-generalizing. Lemmatization converts words into their dictionary base form ("lemma"). Finally, text cleaning removes unwanted characters, symbols or stop words from the text. (Cuantum Technologies LLC, 2025; Jurafsky & Martin, 2025)

For the morphologically rich Finnish language, lemmatization is especially important. Finnish words which share the same surface should be put in context to identify the true lemma (Kanerva, 2024). For example: the lemma for the word "tuli" meaning "fire" is "tuli", whereas for "tuli" meaning "came", the lemma is the verb "tulla" ("to come").

Tokenization, a fundamental technique in text processing, breaks text into manageable, standardized units (the "tokens"). These tokens can be words, subword units, or even characters. Tokenization typically works in conjunction with sentence segmentation and text normalization, as these processing steps complement each other in identifying sentence boundaries, managing punctuation, and handling special characters effectively. By transforming raw text into such structured units, tokenization enables the extraction of meaningful features essential for downstream NLP tasks. (Cuantum Technologies LLC, 2025; Jurafsky & Martin, 2025)

Tokenization varies by language, for example Finnish uses a comma as the decimal marker while English use the period (Jurafsky & Martin, 2025). Importantly, general-purposed tokenizers handle Finnish significantly less efficiently than English in term of tokenization ratio (Chelombitko & Komissarov, 2024). To visualize how tokenization varies across languages, the sentence "I am a student" and its translations are tokenized using the google/gemma-7b model on the Tiktokenizer website (Tiktokenizer, n.d.). The token-to-word ratios, presented in Table 3, illustrate language-specific differences in vocabulary, morphology, and subword structure, where Finnish exhibits a higher token to word ratio compared to other languages in the example. This has important implications for Finnish-

language AI chatbot performance, including increased latency, as token count directly impacts processing time, and reduced effective context window usage, as more tokens required mean less semantic content can be used (Petrov et al., 2023). Tokenization ratio is also dependent on model and may vary across embedding and large language models. Finnish-efficient tokenization should be taken in considerations in developing language model applications (Chelombitko & Komissarov, 2024).

Table 3. Tokenization comparison across languages

	Text	Character count	Word count	Token count	Token to word ratio	Token IDs
en	I am a student	14	4	5	1.25	2, 235285, 1144, 476, 5913
vi	Tôi là sinh viên	17	4	5	1.25	2, 117709, 5536, 21693, 29883
fr	Je suis étudiant	16	3	4	1.33	2, 6151, 20154, 192063
fr	Je suis étudiante	17	3	5	1.67	2, 6151, 20154, 54421, 1994
fi	Minä olen opiskelija	20	3	7	2.33	2, 5204, 235383, 134127, 1200, 37103, 166399
fi	Olen opiskelija	15	2	6	3	2, 235302, 2597, 1200, 37103, 166399

Text must be transformed into a numerical representation before it can be processed by NLP models. Since linguistic meaning (semantics) is represented in a continuous, high-dimensional vector space, this process is often referred to as vector representation or vector semantics. In the context of modern embedding models, the vector that represents the text is called an “embedding” (Jurafsky & Martin, 2025). An example of such an embedding vector is illustrated in Figure 3, showing the first 12 dimensions of the 1024-dimensional vector representing the sentence “Hämeen linna on Hämeenlinnassa” (“Häme Castle is in Hämeenlinna”) using the snowflake-arctic-embed2 embedding model.

Figure 3. Vector representation for the phrase “Hämeen linna on Hämeenlinnassa”

```
[0.4029713571071625, -0.24617336690425873, 0.7695565223693848,
-0.8118278980255127, 0.3327203691005707, 0.5464510917663574,
-0.4217877984046936, 0.27082961797714233, 0.11984366178512573,
-0.6773697137832642, 0.0593334399163723, 0.23336103558540344,...
```

The underlying principle of embedding techniques is to derive the meaning of text units from their context: words or texts that appear in similar contexts tend to have similar meanings. In a continuous vector space, text units are mapped based on their distributional properties, using a co-occurrence matrix to represent how often words appear together (Jurafsky & Martin, 2025). Thanks to contextual modeling, the same word surface, such as “kuusi”, is represented differently depending on its context, for example as “six” or “spruce”. Similarly, words like “kieli” and “hiiri” are assigned distinct representations based on their context, distinguishing between their original senses (the “tongue” and the animal “mouse”, respectively) and their derived meanings (the “language” and the “computer mouse”, respectively).

Given the limited number of NLP models trained specifically for the Finnish language, a practical approach for this project is to utilize multilingual models. This strategy is particularly well-suited for this thesis, which focuses on developing an AI chatbot capable of understanding software documentation that frequently combines Finnish and English terminology.

Multilingual models rely on multilingual embeddings, which represent text from various languages in a shared vector space. Cross-language embeddings represent text from multiple languages within a shared vector space to enable semantic alignment across languages. This is achieved through two main approaches: implicit and explicit alignments. Implicit alignment leverages shared vocabularies, cross-lingual embeddings, and self-supervised learning on multilingual corpora to indirectly align languages without relying on direct translations. In contrast, explicit alignment uses pre-trained models or machine translation systems to directly align multilingual representations, employing techniques such as self-distillation, fine-tuning of query encoders, and monolingual similarity matrix distillation to improve performance, especially for low-resource languages. Together, these methods allow cross-language embeddings to effectively capture semantic relationships across languages for various multilingual tasks. (Tao et al., 2024)

For instance, embeddings for words or texts with similar contextual meaning in different languages are close to each other in the shared vector space (Meta Engineering, 2018). For example, “’äiti” in Finnish and “mother” in English would be close to each other, reflecting their semantic similarity despite the language difference. Multilingual embeddings mapped into a shared vector space theoretically enable multilingual information retrieval and conversations. This capability can be leveraged by instructing the model through a prompt template and query in one language (such as English) while requesting response in

another indicated language (Finnish in this thesis). This approach will be experimented in this chatbot development project.

2.2.3 Semantic similarity and information retrieval

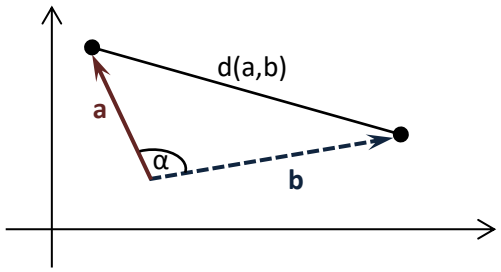
Contextual embeddings, as discussed in the previous section, are vectors that represent the meaning of text based on the context. These vectors can be used to measure the semantic similarity between text units, which is essential for semantic information retrieval, one of the two core components in a Retrieval-Augmented Generation (RAG) system. (Jurafsky & Martin, 2025)

Since text semantics is represented in the embedding vector space, semantic similarity is also called embedding similarity. To compare vectors, three metrics are commonly used, including the Euclidean distance, the dot product, and the cosine similarity (Chip, 2024).

These metrics and their adaptations are discussed below, based on two vectors $\vec{a}$ and $\vec{b}$ in an n-dimensional space, with $a = [a_1, a_2, \ldots, a_n]$ and $b = [b_1, b_2, \ldots, b_n]$. Illustrations are drawn in a 2-dimensional space for simplicity.

The Euclidean distance between the two mentioned vectors is the straight line between them. It is calculated as $d(a,b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \cdots + (a_n - b_n)^2}$ (Pinecone.io, n.d.-b). To simplify the calculation, the METRIC_L2 (IndexFlatL2) is used, as the square of the Euclidean distance, by avoiding the square root computation (Facebook Research, n.d.-b). Figure 4 illustrates the Euclidean distance in 2-dimensional space. By definition, texts of greater similarity tends to have their representing vectors closer to each other, i.e. the smaller Euclidean distance (or it squared metric, L2) between them. (Facebook Research, n.d.-b)

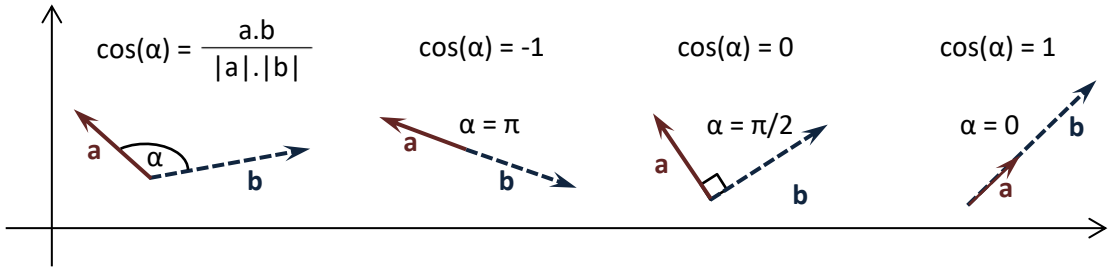
Figure 4. Euclidean distance between two vectors in 2-dimensional space



The dot product (scalar product) of two vectors mentioned above, is calculated as: $a.b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$ or $a.b = |a||b|cos(\alpha)$ where α is the angle between a and b. The larger the product, the more similar the two vectors are. (Pinecone.io, n.d.-b)

Cosine similarity is based on the trigonometric cosine of the angle formed by the two mentioned vectors. Cosine similarity is calculated as $\cos(\alpha) = \frac{a.b}{|a|.|b|}$. It ranges in value from -1 to +1, mathematically. The more similar in the direction, the greater cosine is (Facebook Research, n.d.-b). Figure 5 illustrates cosine similarity and it range of value.

Figure 5. Cosine similarity of two vectors in a 2-dimensional space



The discussed metrics are commonly applied in semantic search, an essential component of RAG systems. Semantic search is a form of information search. Information search, if limited to one database or system, is referred to as information retrieval. It is the core machinery of search engines, recommendation system, etc. and RAG. Basically, retrieval evaluates the relevance of the documents in a database versus the query, then ranks those documents accordingly. Two common types of retrieval are lexical retrieval - based on the keyword, and semantic retrieval - based on the embedding representation. (Chip, 2024)

Lexical retrieval assumes the relevance of a keyword (term) in a document is proportional to its occurrence, call the term frequency (TF). For example, if the count for the word “opinnäytetyö” (“thesis”) in a document is 99, its TF is 99 in that document. On the other hand, terms such as “että” (the conjunction “that”), “mutta” (“but”), “tai” (“or”), etc. tend to occur more frequently in many documents, but only play auxiliary roles. Their importance is inversely proportional to the number of documents containing them, is measure by the metric inverse document frequency (IDF). For example, in a collection of 10 documents, the word “Euclidean” occurs in 2 documents, its IDF is 10/2 = 5; the word “chatbot” occurs in all 10 documents, its IDF is 10/10 = 1. The higher the IDF, the more important it is to the retrieval mechanism. TF and IDF are combined to form the TF-IDF score. Based on it,

other metrics are developed such as the BM25 and its variances BM25+, BM25F. (Chip, 2024)

Semantic retrieval, based on vectors (embeddings) stored in a vector database. The naïve search uses one of the metrics discussed earlier (L2 metric / Euclidean distance, dot product, and cosine similarity) to compare the query embedding to all the embeddings in the vector database and find the k nearest neighbors (k-NN). This approach is precise, but slow and computationally costly, and appropriate for small datasets only. Vectors should be indexed and stored optimally to enable fast and efficient search. For larger datasets, embedding search typically relies on the approximate nearest neighbor (ANN) technique. Many search algorithms and libraries have been developed for vector search. The most popular ANN libraries include FAISS (Facebook AI Similarity Search), Google's ScaNN (Scalable Nearest Neighbors), Spotify's Annoy (Approximate Nearest Neighbors Oh Yeah), and HNSW (Hierarchical Navigable Small World). (Chip, 2024)

Semantic and lexical retrievals can be combined in hybrid search strategies. In a sequence hybrid approach, a simpler but faster lexical search first brings up a list of selected candidates; second, the slower semantic search, in a reranking step, finds the best group of candidates among those preselected documents. This reduces the number of vectors to be compared by the embedding search, and potentially improves the search latency. (Chip, 2024)

2.2.4 Large language models and prompt engineering for text generation

Large Language Models (LLMs) form a major progression in AI, especially in natural language processing. They enable fluent, human-like text generation for many applications including RAG. Their impressive language generation capability comes from deep learning, transformers, massive pretraining, and transfer learning. Modern LLMs rely on the transformer architecture, which uses parallel self-attention to process all input at once, capturing complex word relationships. Many LLMs use only the decoder stack, with masked self-attention for efficient, context-aware text generation. (Alammar & Grootendorst, 2024; Vaswani et al., 2017)

LLM training uses an autoregressive objective, predicting each next token given previous ones, optimizing for correct sequence probabilities. LLMs are pretrained on vast text corpora, exposing them to diverse language styles and structures for broad linguistic understanding. Pretraining is usually unsupervised, using self-supervised objectives, and

requires significant computational resources, but yields highly generalizable models. After pretraining, LLMs can be fine-tuned or prompted for specific tasks, allowing adaptation with minimal data and supporting few- or zero-shot learning. (Alammar & Grootendorst, 2024; Chip, 2024)

Most LLMs use subword tokenization (like BPE or Unigram) for efficient handling of rare words and better generative fluency. Tokenization has been discussed in detail in section 2.2.2 . Since there are fewer unique tokens than unique words, tokenization impacts both model vocabulary size and performance. (Alammar & Grootendorst, 2024; Chip, 2024)

LLMs face notable limitations, including the tendency to generate incorrect or fabricated information (“hallucination”). This can result in unreliable outputs, especially when models are asked about topics outside their training data or when there is ambiguity. Additionally, LLMs may struggle with context retention in long conversations, exhibit biases or toxicity from their training data, and have finite memory capacity due to fixed context windows, all of which impact their accuracy and trust in real-world applications. (Alammar & Grootendorst, 2024; Chip, 2024)

Recent LLM progress follows scaling laws: more parameters, data, and compute improve performance, but also bring inference, energy, and deployment challenges. As the field advances, improvements in training efficiency, interpretability, and ethics will guide future LLM development. (Alammar & Grootendorst, 2024; Chip, 2024)

As these resource demands grow, LLM innovation now prioritizes practical efficiency and accessibility. Models like Llama (Meta, n.d.-b), Gemma (Google AI for Developers, 2025), and Mistral (Mistral AI, n.d.-b) focus on efficiency, open access, and deployment, using novel architectures and quantization for resource-constrained setting. This project’s feasibility depends on these evolving trends.

Meta’s Llama is designed for public access and local deployment, especially in its smaller (7B, 13B) versions, using a decoder-only transformer with innovations like grouped-query attention. Trained on open datasets, Llama-2 outperforms earlier models when fine-tuned and its open weights enable wide research and use on consumer GPUs. While Llama-4 is the latest, only Llama 3.2 and earlier versions have quantized versions suitable for this project. (Meta, n.d.-b; Ollama, n.d.-a)

Gemma, by Google DeepMind, is a lightweight, open model family optimized for efficient, local deployment in sizes like 2B and 7B, supporting fast, safe generation via instruction tuning and RLHF. Compatible with popular frameworks, Gemma is ideal for edge and enterprise use; the latest Gemma 3 and its quantized variants are suitable for this project. (Google AI for Developers, 2025; Ollama, n.d.-a)

Mistral models, created by Mistral.ai, deliver high performance at small sizes by using advanced transformer variants. Key features include sliding window attention (reducing memory use while maintaining context) and grouped query attention for scalability. Mistral models are trained on high-quality, deduplicated data and often outperform larger models per FLOP, making them suitable for this project on-device and low-power inference. (Mistral AI, n.d.-b; Ollama, n.d.-a)

To use LLMs effectively, the input (prompt) should have clear instructions, context, and desired formats. Prompt engineering is the practice of designing and refining prompts to guide generative AI models to give relevant, accurate, and contextually appropriate responses. Prompt engineering is critically important in RAG, as the prompt is augmented with the context and instruction to aim for the desired response: relevant to the provided domain knowledge and reduced hallucinations. Effective prompt engineering leverages both creativity and technical understanding, often requiring iterative adjustments and experimentation to achieve optimal results. (Berryman & Ziegler, 2024).

Suitable models for this project such as Llama-2, Llama3 (Meta, n.d.-b), Gemma3 (Google AI for Developers, 2025), and Mistral (Mistral AI, n.d.-b) each have specific prompt formats recommended by their providers. These guidelines are carefully followed in this project to ensure compatibility and maximize performance (Program code 20).

2.3 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is an application that enhances language models by using information retrieval to inform language generation. This dual approach is particularly well-suited to the challenges identified in Finnish-language software documentation. RAG systems first use semantic search (often via a vector store) to retrieve the most relevant text chunks from a local document database. These retrieved contents are then passed, together with the user’s query and a customized instruction, to a large language model (LLM) to generate the response. (Chip, 2024)

2.3.1 Rationale for choosing RAG

Beside RAG, fine-tuning and Cache-Augmented Generation (CAG) are viable methods for adapting large language models to domain-specific tasks, they present notable trade-offs in the context of this project. Fine-tuning requires significant computational resources, large domain-specific datasets, labour and time (Anisuzzaman et al., 2025; Borek, 2024), making it impractical for this work's specific requirement of offline operation on legacy hardware.

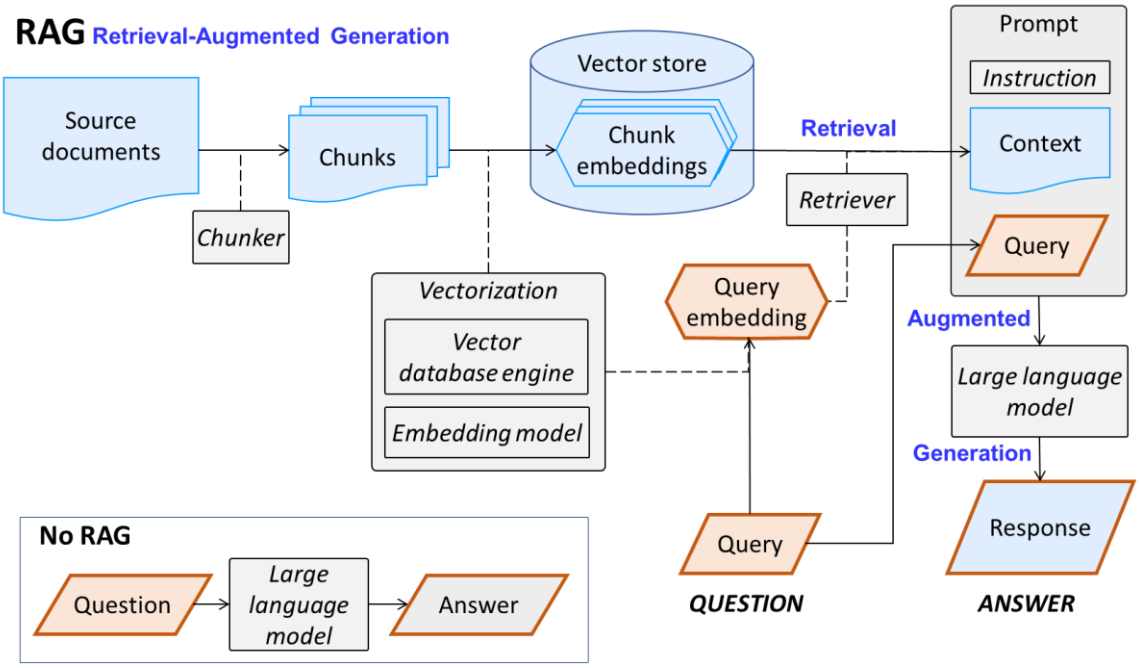
Cache-Augmented Generation, despite its speed and simplicity, requires high memory to preload the data to the cache. It also requires the whole data source to fit within the context windows (less suitable for resource-constraint devices), and is less effective compared to RAG for complex or highly specific queries. (Chan et al., 2024)

In contrast, RAG requires no training or fine-tuning yet provides real-time contextual responses by dynamically retrieving relevant text chunks. This makes it particularly suitable for evolving software documentation. Furthermore, RAG's modular design allows the document source to be flexibly updated or replaced, enabling quick adaptation to new information or different topics. (Zhao et al., 2024)

2.3.2 RAG architecture

Retrieval-Augmented Generation (RAG) architecture is built around two key components that operate sequentially: information retrieval and natural language generation. The output of the retrieval step is used to augment and inform the subsequent language generation process. (Zhao et al., 2024) This setup gives domain-specific information (the software documentation in this project) to the large language model for generating responses that are expected to be more accurate and contextually relevant. In a RAG system, the vector store is a database containing embeddings (vector representations) of text chunks built from the source data. Upon the user's query submission, the query is first embedded using the same embedding model that was used to build the vector store (database), resulting in a query embedding. This query embedding is compared to the chunk embeddings in the vector store, using semantic similarity search, to retrieve the most relevant chunks. The original text of these top-matched chunks is then restored as the context. The context is injected into a prompt template together with the original query. The augmented prompt is then passed to the large language model (LLM) to generate the response. (Chip, 2024) The data flow in a basic RAG system is illustrated in Figure 6.

Figure 6. RAG question answering mechanism



To build a local RAG chatbot prototype, a set of essential components is required. These components are illustrated in Figure 7, and their respective roles described in Table 4. This modular architecture ensures a flexibility for customization as each part of the system is clearly defined and can be reviewed and replaced independently as needed. (IBM, n.d.-b)

Figure 7. RAG system essential component architecture

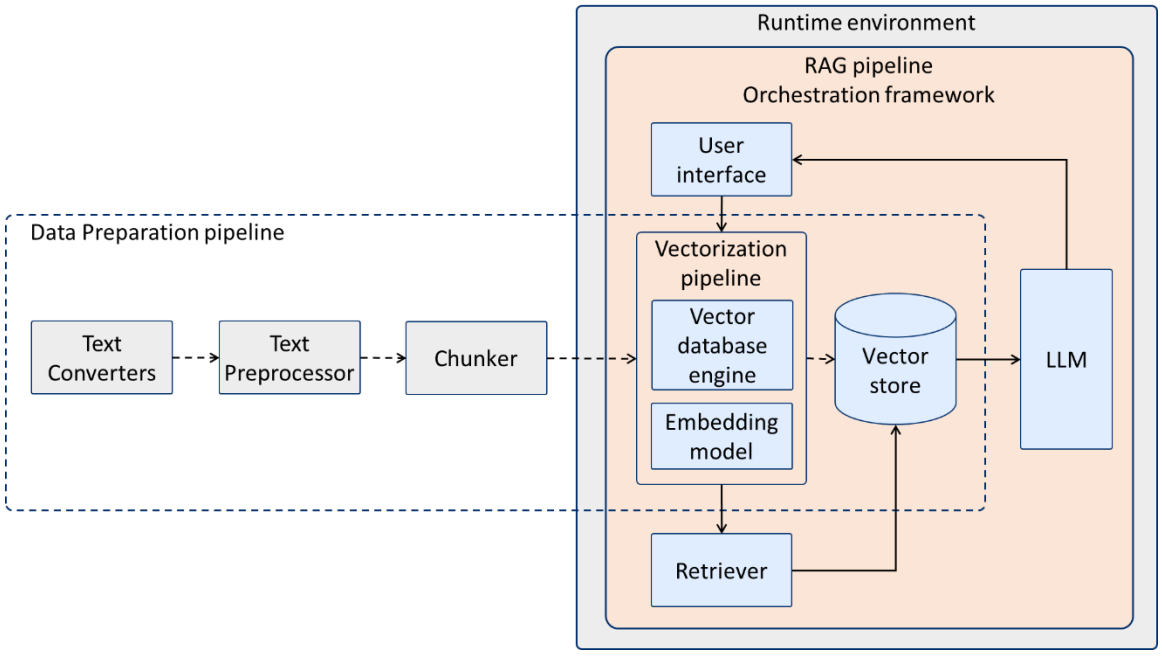


Table 4. RAG components and their roles

Component	Role
Runtime environment	Provides the infrastructure and computational resources to host and execute all system components.
Orchestration framework	Manages the workflow, integration, and communication between modules.
Data preparation pipeline	Coordinates the transformation of raw data into structured, chunked, and vectorized format suitable for retrieval and generation.
Text converter	Transforms various file formats such as PDF and DOCX into plain text for further processing.
Text preprocessor	Cleans and normalizes text by removing unwanted lines, special characters, or irrelevant content.
Chunker	Splits preprocessed text into smaller, manageable segments (chunks) to optimize embedding and retrieval.
Vector database engine	Stores, indexes, and enables efficient similarity search of vector embeddings.
Embedding model	Converts text chunks and queries into high-dimensional vector representations for similarity search.
Large language model	Generates contextually relevant natural language responses based on retrieved information and user queries.
Retriever	Searches the vector database for the most relevant chunks in response to a query embedding.
User interface	Enables users to submit queries and view generated responses, facilitating interaction with the system.

2.4 Ethical and sustainability considerations

The development of AI-powered chatbots must be guided by strong ethical principles and informed by advice from competent authorities. The European Union's guidelines on ethical AI outline seven key requirements for building trustworthy systems: human agency and oversight; robustness and safety; privacy and data governance; transparency; diversity, non-discrimination and fairness; societal and environmental well-being; and accountability. (European Parliamentary Research Service, 2019)

Sustainability is another essential consideration in AI development. Large language models are known for their substantial energy demands, raising concern over their environmental impact. Recommended strategies to mitigate this include: algorithm optimization, hardware

optimization, data center optimization, and pragmatic scaling factor reductions. (Rohde et al., 2024)

3 Methodology

Developing and evaluating an effective AI chatbot involves not only technical implementation but also planning. This section presents the guiding requirements, the approach taken in system development, and the methods used to evaluate the prototype's performance. The goal is to ensure the solution is both functional and reliable, meeting the intended use cases and quality standards.

3.1 Functional and non-functional requirements

The commissioning company seeks an offline AI chatbot capable of delivering fast and accurate answers about their Tweb REST API, based on the documentation in Finnish. The goal is to develop and evaluate an AI chatbot that can operate without exposing proprietary information to external servers or cloud services. To support development, the company has supplied an 80-page Tweb REST API PDF file, under a nondisclosure agreement (NDA)

The chatbot must fulfill both functional and non-functional requirements to address stakeholder's expectations. These requirements also ensure the tool aligns with the defined project constraints, are presented in Table 5.

Table 5. Functional and non-functional requirements

Functional requirements
1. The system must accurately answer user queries based on the provided Tweb REST API documentation. 2. The chatbot must support Finnish-language input and output, while also correctly handling embedded English IT terminology present in documentation and developer queries. 3. The system must provide a command-line interface (CLI) to integrate into existing developer workflows and allow future extension to graphical or web-based frontends..

Functional requirements
4. The chatbot must display a disclaimer indicating that responses are AI-generated and should be verified against the official documentation, and warn users about potentially destructive commands (e.g., DELETE, POST, UPDATE). 5. The system must operate entirely on local machines, without exchanging data with external servers. 6. The chatbot must support flexible configuration, including easy updates to the source software documentation. 7. The system must allow for flexible experimentation with different RAG system configurations to improve answer accuracy and responsiveness.
Non-functional requirements
1. The chatbot must deliver fast response times (<3 seconds), even when running on legacy or resource-limited hardware. 2. The project focuses on developing a prototype-level solution, with an emphasis on configurability, evaluation, and performance tuning, rather than production deployment. 3. The interface must be implemented as a CLI, with future expandability toward graphical or web-based frontends. 4. The system design should enable extraction of practical guidelines for deploying RAG-based chatbots in privacy-sensitive, low-resource environments. 5. The entire solution should serve as a proof of concept demonstrating the feasibility of local AI-assisted documentation support for Finnish-language IT projects.

3.2 Prototype development

The development process of this work follows the Design Science Research Process, applying an iterative cycle of design, development, demonstration, evaluation, and refinement guided by feedback (Sahadevan et al., 2025). In this project, weekly meetings and as-needed information exchanges with the commissioning company enable continuous feedback, clarification of requirements, and alignment on evolving priorities. These regular check-ins support agile responses to challenges and help maintain transparency throughout the project. To support this, documentation is maintained throughout the project. Data used in development is handled locally with regular backups, in line with the project's privacy requirements.

3.2.1 Technology and RAG component selection and parameter configuration

The development of a local RAG tool in this project involves selecting from a variety of open-source components, each responsible for a specific function in the pipeline. Table 6 presents notable options for each component, all of which are currently available and widely applied by many developers. Due to time constraints, not all candidates were tested; instead, selections for experimentation are informed by recent publications and practical evaluations during the development process to align with the project objectives.

Table 6. Technology and RAG components

Component	**Options (not exhaustive)**
Programming language and platform	Python (Python, n.d.), Java (Java, n.d.), Javascript (Mozilla, 2025) Low-code/no-code : n8n (n8n, n.d.), RAGARrch (AI-ANK, n.d.), Langflow (Langflow AI, n.d.)
Integrated Development Environments (IDE)	VSCode (Microsoft, n.d.). JupyterLab (JupyterLab, n.d.), PyCharm (JetBrains, n.d.)
Runtime environment	Ollama (Ollama, 2023/2025), llama.cpp (Llamacpp, n.d.), gpt4all (NomicAI, 2023/2025), vLLM (Vllm project, 2023/2025), LMDeploy (InternLM, 2023/2025), Haystack (DeepsetAI, n.d.)
Orchestration framework (RAG pipeline)	LangChain (LangChain, n.d.-b), LlamaIndex (LlamaIndex, n.d.-a), Haystack (Haystack, n.d.), Ragas (Ragas, n.d.), Dspy (Stanfordnlp, 2023/2025)
Vector database engine	FAISS (Jegou et al., 2017), Chroma (Chroma, n.d.), Milvus (Milvus, n.d.), Qdrant (Qdrant, n.d.), Annoy (Spotify, 2013/2025), Weaviate (Weaviate, n.d.)
Embedding model	bge-m3 (Chen et al., 2024), snowflake-arctic-embed2 (Yu et al., 2024), paraphrase-multilingual-MiniLM-L12-v2 (HuggingFace, n.d.), distiluse-base-multilingual (Hugging Face, n.d.)
Large language model (LLM)	Llama (Meta, n.d.-b), Gemma (Google AI for Developers, n.d.-b), Mistral (Mistral AI, n.d.-b)
Retriever	VectorIndexRetriever (LlamaIndex, n.d.-d), BM25Retriever (LangChain, n.d.-a), CustomRetriever (LlamaIndex, n.d.-c)
Text preprocessing, Chunker	Python scripts to adapt to the specific needs of the project.
User interface	CLI as defined in the functional requirements.

Architecture decisions and considerations are based on guidance from trustworthy sources. The publication sources includes universities and notable organizations. (IBM, n.d.-c) (IBM AI Academy, 2024). Developer familiarity with the technology is also taken into consideration for quick prototyping in this project.

Computers used in the development: To support the development of an AI chatbot suitable for low-end hardware environments, a virtual machine was created using VirtualBox (Oracle, n.d.). The virtual machine was configured with 4 CPUs, 8 GB of RAM, and no GPU, representing a constrained hardware setup. Ubuntu Server was chosen as the operating system due to its open-source nature, compatibility with the selected project technologies, and suitability for limited-resource environments (Ubuntu, n.d.). For performance comparison, the GPU-based counterpart was the developer's own computer, equipped with a GPU that has CUDA Compute Capability 7.5.

Selection of programming language and IDE: Python is the leading programming language in the field of NLP (Cuantum Technologies LLC, 2025), supported by powerful frameworks such as Hugging Face Transformers, LangChain, LlamaIndex, and TensorFlow.

Conda (Conda, n.d.) is used as environment and package management tool in this project. Its powerful CLI use is available on Linux, Windows, and macOS, making it particularly suitable for this project for deploying on low-end hardware, using Linux server distributions.

Visual Studio Code (VSCode) offers the flexibility to integrate multiple languages, libraries, and tools within a single workspace. Its integrated terminal is particularly useful for running scripts, managing environments, and interacting with vector databases or LLM APIs during RAG system development. Combined with extensive extension support and built-in Jupyter Notebook capabilities, VSCode provides a lightweight yet powerful IDE tailored to iterative and modular AI workflows (Microsoft, n.d.).

Selection of runtime environment: In this project, the selection of Ollama as the runtime environment was motivated by recent publications, particularly the article in which a group of experts at Tampere University evaluate RAG system (Khan et al., 2024) and IBM Developer guidance (Goodhart, n.d.; Jeya, n.d.). It offers quick prototyping and efficient deployment of large language models (LLMs) in resource-constrained settings. Ollama's ability to run LLMs locally with minimal setup is particularly beneficial for low-performance environments, aligning with the company's demand for on-premises, resource-efficient

solutions. Ollama is also Docker ready, allowing efficient delivery via containerization. (Ollama, 2023/2025)

Selection of orchestration framework: LlamaIndex is selected for implementing Retrieval-Augmented Generation (RAG) due to its streamlined focus on data indexing and efficient retrieval, which are critical for our application's performance. LlamaIndex excels in creating and managing indexes of large datasets, offering robust search and retrieval capabilities tailored for question-answering and knowledge base applications. Its core strength lies in providing efficient data ingestion and querying, making it well-suited for document-centric RAG systems. This focus allows for rapid prototyping and deployment of RAG applications with minimal complexity. (Belcic & Stryker, n.d.)

Selection of retriever: with the choice of Llamaindex, there are several options for the retrievers (LlamaIndex, n.d.-b). In this project, the default retriever (VectorIndexRetriever) is used for the prototype.

Selection of vector database engine: FAISS was chosen as the vector database for Retrieval-Augmented Generation (RAG) due to its proven efficiency and scalability in handling large-scale semantic search tasks. FAISS (Facebook AI Similarity Search) is optimized for high-dimensional vector search, making it ideal for applications that require fast and accurate retrieval of relevant information from extensive datasets. (Facebook Research, n.d.-a; Jegou et al., 2017)

Selection of embedding model: Available embedding models compatible with Ollama and particularly suitable for Finnish language tasks include snowflake-arctic-embed2 (Yu et al., 2024), and bge-m3 (Chen et al., 2024). Both models generate 1024-dimensional embeddings, support an 8,000-token context window, and have a similar model size of approximately 1.2 GB.

Selection of large language model: language models are planned for development and evaluation. The set of LLMs described in Table 7, are selected according to the following criteria: quantized versions with fewer parameters suitable for low-end hardware; multiple versions of the same model for comparison; instruct-tuned variants suitable for chatbot use; and larger models from alternative providers with different architectures to serve as AI judges. The prototyping models chosen include gemma3:1b-it-qat and gemma3:4b-it-qat (Google AI for Developers, 2025). For the AI judge role, mistral:7b-instruct-v0.3-q4_1

(Mistral AI, n.d.-a), which is based on the llama architecture, is preferred over llama3.1:8b-instruct-q4_0 (Meta, n.d.-a) for its relative smaller size and fewer parameters.

Table 7. LLMs pre-selection for the RAG prototype

Model	Architecture	Parameters	Quantization	Context window (tokens)	Size (GB)
gemma3:1b-it-qat	gemma3	1000M	Q4_0	32K	1
gemma3:4b-it-qat	gemma3	4.3B	Q4_0	128K	2
mistral:7b-instruct-v0.3-q4_1	llama	7.25B	Q4_1	32K	4.6
llama3.1:8b-instruct-q4_0	llama	8.03B	Q4_0	128K	4.7

Parameter adjustment: Key model parameters are systematically adjusted to assess their impact on response time and output quality, as summarized in Table 8. These include the context window size (num_ctx) for the embedding model, and num_ctx, maximum new tokens (max_new_token), and temperature for the large language model (LLM). Parameters are applied across three stages: data processing, retrieval-augmented generation (RAG), and AI-based evaluation. For the embedding model, num_ctx is set to 2048 during data processing to capture the whole chunk, then reduced to 512 in RAG due to typical shorter queries. LLM parameters vary: num_ctx ranges from 4096 in RAG to cover the instruction, retrieved context, and query, to 32,768 in the AI judge to include the instruction, query, context, and RAG-generated answer. The max_new_token spans 512 in RAG for concise responses and 1024 in AI judge for thorough reasoning. The batch inference module use the same configuration as the RAG pipeline to reflect actual RAG performance. Temperature is set to 0.1 in RAG for varied responses when the user repeat the same query, and 0 in AI judge for a consistent evaluation. All the configurations are set in a centralized configuration file (Program code 3). Section 3.2.4 provides rationale supporting these parameters, based on the chunk sizes of the documents used.

Table 8. Key system parameter configuration

	Data processing	RAG pipeline, Batch inference module	AI judge
Embed model num_ctx	2048	512	N/A
LLM num_ctx	N/A	4096	32,768
LLM max_new_token	N/A	512	1024
LLM temperature	N/A	0.1	0

3.2.2 Environment, project directory, and configuration settings

A modular directory structure is adopted in this project, separating code, data, configuration, and evaluation, making it straightforward for new users to navigate and for developers to extend. The directory structure is illustrated in Figure 8. The ".gitignore" file in the project root directory also serves the role of marker for flexible future adjustments of the directory structure.

By developing a modular approach and leveraging these existing tools, the project is able to focus on customization, evaluation, and language-specific tuning, rather than reinventing low-level infrastructure. This approach ensured efficient development while maintaining alignment with the project's goals of modularity, offline compatibility, and strict privacy compliance.

Figure 8. Project directory structure for the RAG system

```
rag/
├── config/
│   └── config.yml
├── data/
│   ├── chunk/
│   ├── evaluation/
│   │   ├── batch_output/
│   │   └── ai_judge_output/
│   ├── source/
│   ├── txt_processed/
│   ├── txt_raw/
│   └── vector_store/
├── notebooks/
├── src/
│   └── rag_app.py
├── .gitignore
├── environment.yml
├── Dockerfile
├── LICENCE
└── README.md
```

To ensure reproducibility and easy setup, an “environment.yml” file is created in the project root (Figure 9). This file lists the environment name, channels, and required dependencies, enabling users to replicate the exact environment on other machines, using a simple Conda command: “conda env create -f environment.yml”.

Figure 9. Environment settings

```yaml
name: rag
channels:
  - pytorch
  - conda-forge
  - defaults
dependencies:
  - python=3.10
  - pyyaml # for yaml
  - cpuonly
  - pytorch
  - pytorch::faiss-cpu
  - conda-forge::pandas
  - conda-forge::pywin32
  - conda-forge::python-docx
  - pip
  - pip:
      - ollama
      - pymupdf #for fitz
      - llama-index.core
      - llama_index.vector_stores.faiss
      - llama_index.llms.ollama
      - llama_index.embeddings.ollama
```

A “config.yml” file is used to centralize key settings and parameters - such as file paths, embedding and language model names, and runtime configurations. It also defines the prompt template, exit command, and other behavioral settings. This modular, model-agnostic approach not only allows for easy switching between language models and embedding methods, but also facilitates adapting the RAG pipeline to different use cases - such as the batch inference module and the AI judge used during the evaluation phase of this project – with no or minimal modification the core codebase. Program code 3 presents details of the configuration file.

To apply the settings, Python scripts in the notebooks automatically locate and load the configuration config.yml file using the .gitignore as the unique marker in the project root directory, as shown in Program code 1. This ensures all parameters are accessible within the code, keeping the workflow consistent and easily configurable. Necessary imports and Classes are presented in Program code 4

Program code 1. Python scripts to load the project root directory and configuration

```
class ConfigManager:
    def __init__(self, marker=".gitignore",
config_subpath="config/config.yml"):
```

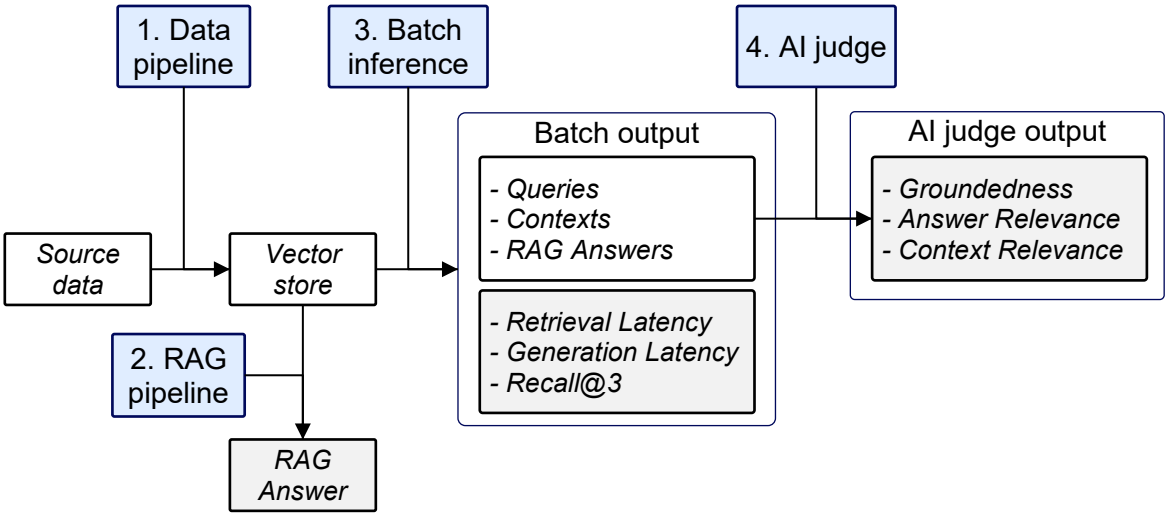
Ollama, the runtime environment for the prototype, is downloaded using the appropriate version according to the operating system. Guidance to install and run Ollama on the local computer is available on the Ollama documentation website (Ollama, n.d.-b).

3.2.3 Jupyter notebooks and python files

Jupyter notebooks serve as the primary environment for developing both the data processing pipeline and the retrieval-augmented generation (RAG) prototype. To ensure modularity and flexibility, classes and functions are organized into separate, clearly structured cells, facilitating iterative development and debugging. Additionally, the RAG notebooks are intentionally written in a format that supports straightforward conversion into standalone Python scripts, enabling easy deployment. Deployment on low-end hardware can be simplify by using bash script to activate the environment automatically and use an alias to call the rag.py function located in the “src” subdirectory.

For evaluation, the original RAG notebook is adapted into two modified files. The first handles batch inference, automatically passing JSON-stored queries to the model and saving the responses. The second, the AI judge, uses an LLM to qualitatively evaluate those responses based on predefined criteria, storing reasoning and scores back in the same JSON file. The roles of the four files are described in Figure 10, with their outputs.

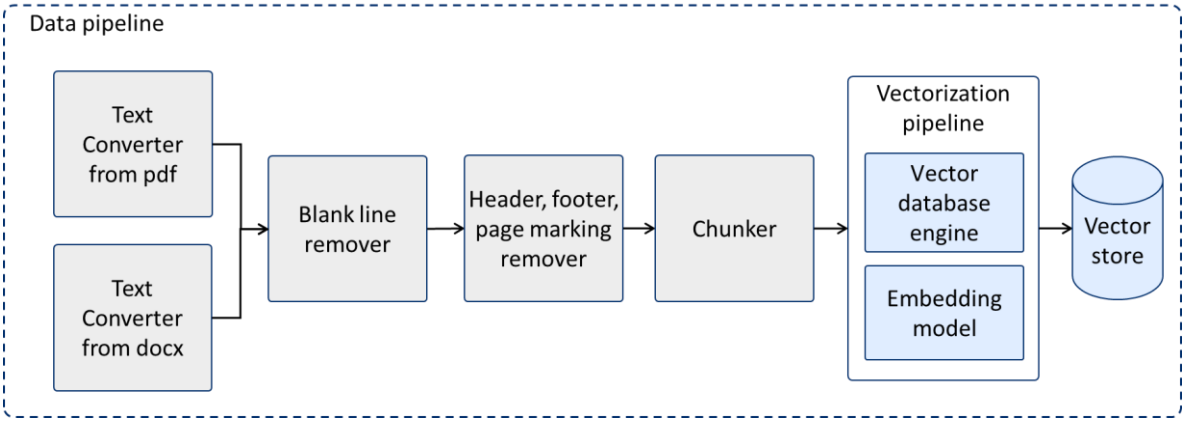
Figure 10. The four notebooks, their inputs and outputs



3.2.4 Data pipeline notebook

The data pipeline notebook is organized into 6 functional steps, from text conversion to vector store building. Figure 11 describes those steps with their functions in sequence.

Figure 11. Steps in data preparation



All the data in the preparation steps are stored in the data directory and its subdirectories, as outlined earlier in Figure 8. The source subdirectory store the source PDF and word files. Raw text converted from PDF and DOCX are stored and further processed in the txt_raw directory (Program code 5 and Program code 6). The fully processed text files ready for chunking and vectorization are stored in the “txt_processed” directory. The vectorization engine used in this project perform chunking and embedding as integrated in

the same operation, the chunk directory store the chunked documents in a separate process to review and adjust the code if needed. Finally, the vector database and relevant JSON files are stored in the “vector_store” directory.

When converting text from PDF, PDF files stored in data/source are converted to TXT file with the name of the original files and the .txt file extension. Similarly, when converting from DOCX to TXT, the naming adopts the same patterns. To avoid file overriding, attention is required to name the PDF and DOCX files differently. Newly created TXT files are store in “data/txt_raw”.

The next step involves automatic removal of blank lines in all files in the data/txt_raw directory. This editing does not change the file names or locations (Program code 7).

Removing of header, footer, and page markers is semi-automatic with human oversight. Each source file may have different header and footer and page number pattern, which needs human review to decide the number of lines relative to the page marker to remove. Undesirable text such as the table of contents should be removed by human. After execution of the code, the files are converted to save in the txt_processed directory with a new name incorporated the old name with the suffix “_processed” (Program code 8).

Chunking in this project is not based on length of text, but based on the section, each being a unit of information useful for the reader. The Chunker, using regular expressions (regrex) identify the heading pattern of each section with number such as “2.1.3” to split the document in to chunks. The chunks, named ‘documents’ is stored as a Document object in the program to pass to the vectorization engine (Program code 9). The chunks are saved both as a JSON file and a text file in the data/chunk directory review to adjust the code and the document content, improving the RAG outcome (Program code 10). The chunk in text format help visualize the content, while the JSON file is loaded to a pandas dataframe to perform basic statistic of word and character counts to guide RAG setting. For this particular document, the mean word count is 41, while the 75 centile has 48 words, the three longest chunks have respectively 362, 248, and 239 words. With top 3 retrieved chunks are planned to serve as the context for LLM generation, longest theorical context has 849 words. With a tokenization ratio of approximately 3.5 for Finnish in the selected LLM models in this project (Brack, 2024; Chelombitko & Komissarov, 2024), the context requires no less than 2550 tokens. With this information, the number of tokens num_ctx is set to 4096 to cover the whole prompt.

The vector store is built in several steps: compute the embed model's vector dimension (Program code 11), create the FAISS index, build and save the index in the vector store (Program code 11 and Program code 12). In this process, which uses abstracted functions from Ollama embedding and FAISS, the vector store (called "index" in FAISS terminology) is built from the processed document, involving the customized chunker in the process. The "vectors.bin" and relevant JSON files are stored in the "data/vector_store" directory, ready for querying by the RAG pipeline. A text file containing the name of the embedding model should be created in this directory for verification when deploying the RAG pipeline, since the RAG pipeline must use the same embedding model used to create the vector store.

3.2.5 RAG pipeline notebook

The RAG pipeline notebook shares most of the configurations and settings with the Data pipeline notebook. For simplicity, both pipelines share the same config file (Program code 3) and configuration scripts in the project (Program code 4).

A function is dedicated to check the status of the local Ollama server, the runtime environment (Program code 13). OllamaEmbedding and Ollama are use to set the embedding and language models with centralized parameters, respectively (Program code 15).

The vector store needs to be loaded for information retrieval (Program code 16). The embedding model and its dimensions used to build the vector store is checked against the embedding model set for RAG for matching (Program code 17). This is critical since semantic retrieval requires the same embedding model and vector dimension are used for the vector store building and query embedding. During vector store building, a json file (_model_info.json) is created in the vector store directory, supporting automatic checking.

For information retrieval, the query engine is built (Program code 18) and used by the chunk retriever (Program code 19). Both the top k number of retrieved chunks and the number of chunks to display to the user are set in the configuration file.

The prompt format is dynamically customized according to the LLM used (Program code 20). Each LLM has each own prompt guidance (Google AI for Developers, n.d.-a; Meta, n.d.-a; Mistral AI Large Language Models, n.d.). The prompt instruction and the organization of the context and query are set in the configuration file (Program code 3). For multilingual LLM, prompt instruction can be in the same language or another one. To save

the number of tokens for important information, the prompt uses placeholder to answer in case the context does not support a grounded answer. The placeholder is replaced upon answering. For the same reason, the prompt does not guide the model to generate a warning message in case of potential data loss risk. Instead, disclaimer and warning messages are displayed systematically for all queries.

The context is constructed from the retrieved chunks, using regular expressions. At this point, chunk metadata can be added if needed (Program code 21).

The LLM response is generated using the customized prompt, and return a response string (Program code 22). The main function orchestrate all the step including a user interaction to get the query input and display the response (Program code 23).

3.3 Prototype evaluation

Evaluation of this project's RAG-based AI chatbot prototype involves assessing system latency and evaluating system overall quality. To ensure a reliable and focused evaluation of the system's latency, all unnecessary applications and background processes on the evaluation device are closed, and system resources are monitored to prevent unexpected load spikes. The device is disconnected from all network interfaces: Ethernet, Wi-Fi and Bluetooth. These precautions aimed to eliminate potential sources of latency fluctuations, allowing the inference performance of the chatbot to be measured as accurately and consistently as possible.

RAG quality evaluation is usually classified into assessment of the Retrieval and Generation components (Brehme et al., 2025; IBM, n.d.-c). However, the quality of the retrieved context significantly impacts the performance of the language model's responses. Therefore, the interpretation of a RAG system evaluation should consider the quality metrics holistically, reflecting the integrated performance of both components. (Chip, 2024; Mistral AI, 2025; Pinecone.io, n.d.-a)

A set of 60 questions were developed in collaboration with the commissioning company to evaluate the RAG (Table 9). They cover different RAG quality dimensions, assessing the chatbot's capabilities to generate relevant answers and avoid errors, especially hallucinations. Evaluation is implemented with an AI judge with human review.

Table 9. Questionnaire for RAG-based AI chatbot evaluation

Question	Association to source document	Purpose and expected RAG behaviors
1 – 50 Aligned QnA	Labelled to a chunk, answerable from the chunk	Expect grounded correct answers from relevant chunks. RAG should answer accurately with high context relevance
51 – 52 Misaligned context	Labelled to a chunk, but the question is unrelated to the chunk	Test system's robustness to misaligned context. RAG should return "Not found" or abstain from answering
53 – 56 Lexical trap	Lexically related, but semantically unrelated	Test susceptibility to shallow lexical matches. RAG should not be misled by lexical cues
57 – 60 Off-topic	Off-topic	Test hallucination and noise tolerance. RAG should give robust rejection or neutral output

3.3.1 Evaluation metrics and analysis

Common RAG metrics include Precision, Recall, F1-score, and Retrieval Time for evaluating the Retriever. To evaluate the Generator, common metrics include Accuracy, F1-score, Recall, and Precision (Brehme et al., 2025; IBM, n.d.-c). In this project, seven metrics are used (Table 10) in line with the research questions. One of them is for evaluating the latency of the vector database building process. The other six metrics evaluate the RAG pipeline, with two of them are for latency assessment.

The other four metrics evaluate RAG quality, including Recall@3 and three of the RAG Triad. The RAG Triad metrics are evaluated by an AI judge with human oversight to evaluate the RAG performance. They are adapted from the RAG evaluation Triad (TruLens, n.d.). The evaluation logic is adapted from the approach published in a Mistral website with categorical score, each from 0 to 3 (Mistral AI, n.d.-a). Definitions, evaluation criteria, and scoring logic of each metric are provided in Program code 25.

Table 10. Metrics for RAG-based AI chatbot evaluation

Metric	Description	Method
Vector Store Building Latency	The time required to embed and index the preprocessed documents into a functional vector database.	Observation
Semantic Retrieval Latency	The time elapsed from receiving the user input to retrieving the relevant chunks based on semantic similarity.	Python script

Metric	Description	Method
Response Generation Latency	The time elapsed between injecting the retrieved chunks into the language model and receiving the generated response.	
Recall@3	The percentage of cases in which the expected chunk appears among the top three retrieved chunks.	
Groundedness	The factual consistency between the generated response and the retrieved context.	AI tool with human oversight on a subset of questions
Answer Relevance	The relevance of the generated response to the user's query.	
Context Relevance	The relevance of the retrieved context to the user's query.	

RAG system evaluation metrics should be interpreted holistically, reflecting the integrated performance of both information retrieval and response generation. (Chip, 2024; Mistral AI, 2025; Pinecone.io, n.d.-a). The three metrics are generated on a configuration using snowflake-arctic-embed2 as embedding model, as guided by the Recall@3 results; and using the gemma3:1b-it-qat as LLM, since this 1B model size demonstrates a preferable latency on low-end hardware, as informed by the results in section 4.2.1. The metrics, including Groundedness, Answer Relevance, and Context Relevance, are all evaluated by an AI judge as described in section 3.3. Metric data ("recall" and the Triad's "scores") are also extracted, as discussed in section 0. The final JSON file records all the input and output data generated by the batch inference module, the AI judge, and the postprocessing script, available for analysis. The categorical variables used in analysis are described briefly in Table 11. Detailed description of the Triad metrics is included in Program code 25.

Table 11. Variables used in quality analysis

Variable	Value	Description
recall	0 – 1	Matching between the labelled chunk and the top 3 retrieved chunks. 0: Not matched 1: Matched Used to evaluate Recall@3 and classify RAG outcome.
groundedness_score (abbreviated as G_score)	0 – 3	Answer vs Context: 0: Not grounded 1: Low groundedness 2: Medium groundedness

Variable	Value	Description
		3: High groundedness Used to evaluate Groundedness and classify RAG outcome.
answer_relevance_score (abbreviated as AR_score)	0 – 3	Answer vs query 0: No relevance 1: Low relevance 2: Medium relevance 3: High relevance Used to evaluate Answer Relevance and classify RAG outcome.
context_relevance_score (abbreviated as CR_score)	0 – 3	Context vs query 0: No relevance 1: Low relevance 2: Medium relevance 3: High relevance Used to evaluate Context Relevance and classify RAG outcome.

There are different points of failure in RAG systems. Authors from the Arizona State University (USA) categorized them in to main areas: Reasoning Failures and Structural Limitation, which are further divided into eight error types (Agrawal et al., 2024). Researchers from the Applied Artificial Intelligence Institute (Australia) pointed out seven failure points (Barnett et al., 2024). These failure points are presented in Table 12.

Table 12. Failure points related to Retrieval-Augmented Generation

Eight points of failure (Agrawal et al., 2024)	**The seven failure points** (Barnett et al., 2024)
Reasoning Failures Misinterpretation of Question's Context Incorrect Relation Mapping Ambiguity in Question or Data Specificity or Precision Errors Constraint Identification Error **Structural Limitation** Encoding Issues Inappropriate Evaluation Limited Query Processing	FP1 Missing Content FP2 Missed the Top Ranked Documents FP3 Not in Context - Consolidation strategy Limitations FP4 Not Extracted FP5 Wrong Format FP6 Incorrect Specificity FP7 Incomplete

There is no available guidance to use the metrics from the RAG Triad (TruLens, n.d.) adapted to match the metrics into common failure points. Proprietary or online tools for RAG Triad interpretation are out of scope of the project. A heuristic adaptation is used to classify RAG outcomes into 5 categories based on the RAG Triad (Table 13).

Table 13. RAG outcome heuristic categories and criteria

Category	**G_score**	**CR_score**	**AR_score**	**Description**
Correct answer	≥ 2	≥ 2	≥ 2	Well-grounded, relevant, on-topic
Hallucination	< 2	any	≥ 2	Not grounded, but answer relevant
Context error	any	< 2	any	Irrelevant context used
Answer error	any	any	<2	Answer not relevant to question
Other	any	any	any	Unclassified

3.3.2 Batch inference module and AI judge notebooks

Two modified versions of the RAG pipeline notebook are developed, as discussed in section 3.2.3 (Figure 10). In the batch inference module, the user interface loop is replaced by an automated loop that processes queries from a JSON file. For each query, it records the list of retrieved chunk headings, the context, the generated response, and latency metrics, and writes the results to an output JSON file in the data/evaluation/batch_output directory. Code modifications involve the only the main function, shown in Program code 24 replacing lines 25 – 52 of the RAG pipeline code (Program code 23). This setup ensures the RAG scripts remain unchanged, and the batch inference module operation reflects RAG behaviors as during normal user interaction.

The AI judge is a further adaptation of the RAG pipeline code, involving the “model setter” (Program code 25), the prompt customizing function (Program code 26), the response generating function (Program code 27), and the main function (Program code 28). The AI judge only uses the modified Generation part and not the Retrieval part from the RAG pipeline. Three addtitional versions of prompt instruction, one for each metric, are included in the config.yaml. The new context builder includes: the query, the context, the rag answer - all generated by the batch inference module. The LLM and its configuration are adjusted

as discussed in section 3.2.2. With this modification, both the prompt customizing function (Program code 26) and the response generating function (Program code 27) take a new argument: the rag_answer, while the logic of each of the response generator remains unchanged.

The lines from 16 - 52 in the RAG pipeline main function is replaced by the AI judge code (Program code 28). This modified function reads the stored query, context, RAG answer in the JSON file generated earlier by the batch inference module. It follows the respective instructions for evaluating Groundedness, Context Relevance, and Answer Relevance to evaluate and generating the respective responses, also saved to a JSON file in the “data/evaluation/ai_judge_output” directory. The resulted file conserves all the original data, the generated data by the batch inference module, and adding new data generated by the AI judge. It is further processed by python scripts (Program code 29) to prepare the target data for analysis. The scripts include function to record the Recall and extract the scores from the relevant string. Since the generated response may display variations of the score, such as “**Score:**” instead of the guided pattern “Score: n”, human verification is necessary when the return value is “null”.

4 Result and Discussion

Findings from the development and evaluation of the RAG-based AI chatbot are discussed in detail in this section. The discussion is centered around the four research questions guiding this study. It begins by the key considerations in applying rapid prototyping to the chatbot’s development, followed by an exploration of data preparation functions tailored for Finnish-language software documentation. The next part focuses on evaluating the prototype’s performance in terms of latency and response quality. The discussion concludes with an overview of the system’s limitations and potential directions for improvement.

4.1 Development of the RAG-based AI chatbot

The development phase focuses on building a functional prototype capable of understanding query and generating responses in Finnish, based on domain-specific content. Emphasis is placed on iterative design, enabling quick experimentation and refinement. Special attention is also given to handling language-specific challenges, particularly those related to processing Finnish-language documentation.

4.1.1 Key considerations in applying rapid prototyping for chatbot development and evaluation (Research question 1)

The work results in a complete set of deliverables, including a functional and testable RAG system with its data pipeline, RAG pipeline, batch inference module, and AI judge. This also includes the full project structure, configuration files, environment setup, assets, and documentation, supporting deployment readiness. The project structure and configurations are detailed in section 3.2 and the all source code is provided in **Appendix 2**.

In the rapid prototyping process used in this project, decisions are primarily guided by a combination of insights from existing publications, the developer's familiarity with the chosen technologies, and practical factors such as time and constraints. Publications from IBM sources (Goodhart, n.d.; Jeya, n.d.) provided concrete examples of building RAG systems using the selected technologies, offering valuable implementation guidance. Particularly, the experience report by authors from Tampere University (Khan et al., 2024) described the development of an offline RAG-based system built from PDFs and evaluated by eight domain experts, contributing to the methodological foundation of this work. A detailed discussion of the technology selection rationale is presented in section 3.2.1.

The project directory and development notebooks are organized modularly to support scalability and flexible, incremental development. Thanks to this modular architecture, both the batch inference module and AI judge are adapted directly from the core RAG pipelines, enabling streamlined automatic testing and evaluation.

The development process followed the Design Science Research Process (DSRP), applying an iterative cycle of design, implementation, demonstration, evaluation, and refinement (Sahadevan et al., 2025). This cycle is strongly informed by early and regular client feedback, which played a central role in shaping both functionality and usability.

Guardrail rules set in the prompt template serve a double role: to guide the LLM answer in context, and to further prevent potential harmful and toxic content. Disclaimers and warnings in the output provide user-centric transparency. AI-generated contents are monitored by human, to implement necessary adjustments for quality improvement, and also to detect any harmful behaviors. The project adheres to AI ethic guidance by implementing those measures. The effort to build a chatbot operating on low-end device reflects sustainability practices of the project.

These considerations, from technology selection and modular design to iterative user-centered refinement and AI ethic and sustainable awareness, collectively define the practical and methodological foundations of rapid prototyping in this project. They not only accelerate development but also ensure the system could be effectively evaluated and improved in successive iterations.

4.1.2 Data preparation for Finnish-language software documentation in the RAG prototype (Research question 2)

Data preparation is critically important to provide a well-prepared text structure for the RAG system. During the RAG prototype development, key considerations are documented, related to preparation of source files, chunking strategy, and Finnish language encoding.

Source file preparation: Converting PDF files to TXT has potentially side effects of undesirable line breaking. Automatic correction, based on identifying the lower-case word at the beginning of lines, has limitation in restoring the original text, especially for software documentation, often containing variable name in lower case at the beginning of line. Moreover, when a camel case variable such as “documentId” accidentally broken to line with the capital “I” at the beginning of the next line, the correction mechanic would ignore it. Since most of documentation PDF files are converted from DOCX file, it is preferable to use the DOCX file as source documents for the RAG tool than using the PDF format.

Chunking strategy: Chunking can critically impact RAG performance. There is no universal best chunk strategy, and chunking should be approach creatively and flexibly based on the structure of the source document. Chunk size matters regardless of the chunk strategy, taking into considerations about the information it conveys, and the capacity of the context windows. (Chip, 2024)

The above recommendations are applied in this project. The source document is a software documentation with numbered section, each typically discusses a unit of guidance. Chunking approach is based on these sections. The chunk size are examined to calculated the possible longest context combined from 3 chunks, guiding the number of tokens setting for the embedding model to vectorize the complete chunk, and the prompt to be completely provided to the LLM with the instruction, context, and user query, avoiding the error of missing context. These practices are discussed in detail in section 3.2.4. The heading numbers can facilitate recall matching, they should be checked to be unique values in the documents.

Ideally, chunk metadata and body text should be kept separately for flexible use. During development, when testing different vector database engine, some does not accept an empty string body. Other engines such as the one currently used in this prototype, automatically take the body text to embed with no option to include the title for a richer context building. In fact, titles convey important information for semantic search. To overcome the limitations of both cases, in this project, the chunk title is added as the 1st line of the modified body text, while still preserved as the metadata title. Later on, when constructing the answer, the title is easily taken back out.

Encoding considerations for handling Finnish language in RAG system: When developing a RAG system that processes Finnish-language software documentation, special attention must be given to text encoding. Finnish uses the extended Latin, non-ASCII letters ä, ö, and å. These characters can cause issues during file I/O operations if proper encoding is not set, leading to corrupted text, decoding errors, or data loss, disrupting the RAG performance. In Python, the standard and recommended encoding for handling such characters is UTF-8. UTF-8 is widely supported across platforms and is fully capable of representing all characters in the Finnish alphabet. To preserve the original encoding, specific flags such as [encoding='utf-8'] and [ensure_ascii=False] should be used, as examples in Program code 2. This practice is systematically applied in this project (Program code 5, Program code 6, Program code 7, Program code 8, Program code 9, Program code 10, Program code 24, Program code 28, and Program code 29).

Program code 2. Flags to preserve Finnish character encoding

```
# The flag encoding='utf-8'
# using with the python built-in open() function
with open('file.txt', 'r', encoding='utf-8') as f:
    text = f.read()

# using with CSV / pandas
import pandas as pd
df = pd.read_csv('file.csv', encoding='utf-8')
df.to_csv('output.csv', encoding='utf-8', index=False)
#
# using with JSON
df = pd.read_json('file.json', encoding='utf-8')

# The flags encoding='utf-8'and ensure_ascii=False
# using JSON
import json
with open('data.json', 'w', encoding='utf-8') as f:
    json.dump(data, f, ensure_ascii=False)
#
df.to_json('output.json', force_ascii=False)
```

4.2 Evaluation of the RAG-based AI chatbot

This section focuses on the systematic evaluation of the developed AI chatbot to understand its practical capabilities and limitations. The assessment is divided into two key areas: the system's response time, reflecting its operational performance, and the quality of the generated outputs, reflecting its effectiveness in producing useful and accurate answers. By analyzing both dimensions, the evaluation provides insights into the strengths of the current implementation and guides further refinement.

4.2.1 Evaluating the latency of the RAG prototype (Research question 3)

Typical RAG latency evaluation include Retrieval and Generation components. This report extends the scope to inform about the preparation step: building the vector database.

Vector Store Building Latency: While not a primary focus of the study, the latency involved in building the vector store was recorded as a supporting observation to better understand practical performance differences across hardware. This metric refers to the time required to embed, and index preprocessed documents into a usable vector database. The source document used for this measurement contains 80 pages, with the converted and preprocessed text file comprising 87,878 characters (8,556 words), the resulting binary vector store size is 825 KB, all the generated binary and json files summed up to 1.98 MB. On a CPU-based system, the vector store construction takes approximately 30 minutes, whereas the same process on a GPU-enabled machine (CUDA Compute Capability 7.5) typically completes in under 2 minutes. The embedding models used - snowflake-arctic-embed2 and bge-m3 -yielded comparable vector store build times, likely due to their similar architecture (both producing 1024-dimensional embeddings and having similar model sizes around 1.2 GB).

Semantic Retrieval Latency: Retrieval can significantly contribute to the overall RAG latency, motivating the application of Cache-Augmented Generation (CAG) (Chan et al., 2024). However, the main latency in a RAG system remains in the generation part; and the added latency due to information retrieval may impact user experience (Chip, 2024). Retrieval latency was measured across four configurations combining two embedding

models (snowflake-arctic-embed2 and bge-m3) with CPU and GPU execution. The results are summarized in Table 14, and statistical comparisons are provided in Table 15.

Hardware effect (same model, different hardware): For both embedding models, using a GPU significantly reduced retrieval latency compared to CPU. With snowflake-arctic-embed2, the median latency dropped from 0.720 s on CPU to 0.085 s on GPU. Similarly, bge-m3 saw a reduction from 0.735 s to 0.086 s. These differences were statistically significant (Wilcoxon p = 0.000 for both cases; Table 15), confirming the strong impact of hardware acceleration. Figure 12 presents the Semantic Retrieval Latency comparison between CPU and GPU using snowflake-arctic-embed2 model.

Model effect (same hardware, different model): On the same hardware, both models performed similarly. On CPU, the median latencies were 0.720 s (snowflake-arctic-embed2) and 0.735 s (bge-m3); on GPU, 0.085 s and 0.086 s, respectively. The differences were not statistically significant (Wilcoxon p = 0.622 on CPU, p = 0.098 on GPU; Table 15), suggesting minimal impact from model choice on retrieval latency.

Table 14 confirms that median retrieval times remained below one second even in the low-end CPU setting. This allowed for rapid display of relevant document chunks for user review, supporting interactive performance.

Table 14. Semantic Retrieval Latency measurements across four configurations

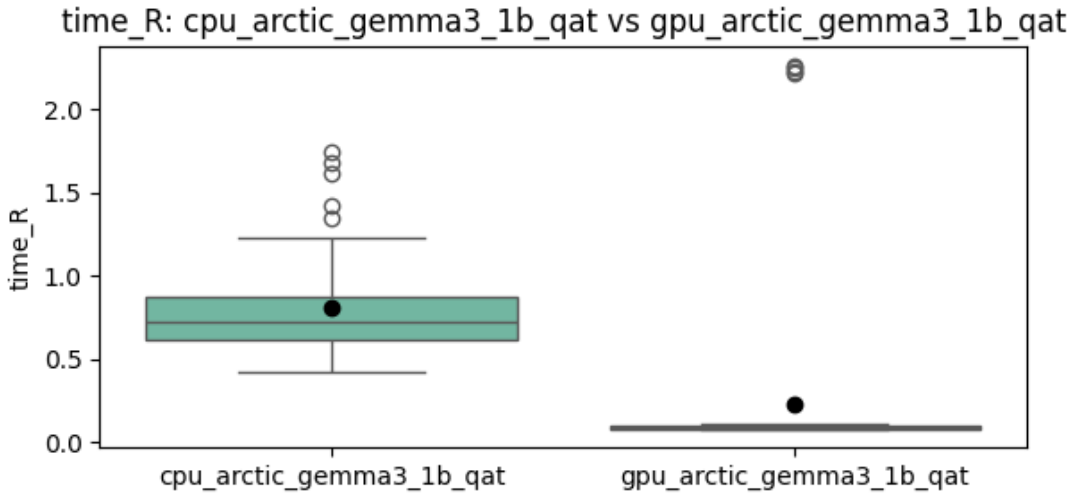
	snowflake-arctic-embed2 (n = 60)	**bge-m3** (n = 60)
CPU	Mean [95%CI]: 0.811 [0.733, 0.888] Min = 0.422, Max = 1.743 Median [95%CI]: 0.720 [0.675, 0.785]	Mean [95%CI]: 0.867 [0.756, 0.979] Min = 0.365, Max = 2.774 Median [95%CI]: 0.735 [0.674, 0.776]
GPU	Mean [95%CI]: 0.232 [0.095, 0.369] Min = 0.080, Max = 2.269 Median [95%CI]: 0.085 [0.084, 0.086]	Mean [95%CI]: 0.305 [0.139, 0.471] Min = 0.081, Max = 2.312 Median [95%CI]: 0.086 [0.085, 0.088]

Table 15. Semantic Retrieval Latency comparison across four configurations

Common settings	**Settings to compare**		**Comparison**
CPU	snowflake-arctic-embed2	bge-m3	Normality: False Wilcoxon test p-value: 0.622
GPU	snowflake-arctic-embed2	bge-m3	Normality: False Wilcoxon test p-value: 0.098
snowflake-arctic-embed2	CPU	GPU	Normality: False Wilcoxon test p-value: **0.000** *

Common settings	Settings to compare		Comparison
bge-m3	CPU	GPU	Normality: False Wilcoxon test p-value: **0.000 ***

Figure 12. Semantic Retrieval Latency comparison between CPU and GPU



Response Generation Latency, the main contributor to RAG latency, was evaluated across five configurations, involving two embedding models (snowflake-arctic-embed2 and bge-m3), two hardware types (CPU and GPU), and two LLM sizes (gemma3-qt-qat Q4_0 1B and 4B). Summary statistics are presented in Table 16, and statistical comparisons in Table 17.

Hardware effect (same model & LLM, different hardware): For both embedding models using the 1B LLM, GPU use led to a significant reduction in latency compared to CPU. Median latency dropped from approximately 23 seconds on CPU to approximately 2.4 seconds on GPU. These differences were statistically significant (Wilcoxon $p < 0.001$ for both models; Table 17, Figure 13), demonstrating the clear performance advantage of GPU acceleration.

Model effect (same hardware & LLM, different embedding model): When using the same hardware and LLM (1B), the choice of embedding model had no significant impact on latency. Median values remained near 23 seconds on CPU and 2.4 seconds on GPU, with no statistically significant difference (Wilcoxon $p = 0.173$ on CPU, $p = 0.317$ on GPU; Table 17).

LLM size effect (same hardware & embedding model, different LLM): A marked increase in latency was observed when increasing the LLM size from 1B to 4B using snowflake-arctic-embed2 on CPU. The median latency rose from approximately 23 seconds to nearly 87 seconds. This increase was statistically significant (Wilcoxon $p = 0.000$; Table 17 and Figure 14), highlighting the substantial trade-off between model size and latency on CPU.

These findings confirm that hardware is the dominant factor in generation speed. While embedding model choice has little impact, LLM size notably affects latency, especially on low-end hardware.

Table 16. Generation Response Latency measurements across five configurations

	snowflake-arctic-embed2 (n = 60)	**bge-m3** (n = 60)
CPU	LLM: cpu_arctic_gemma3_1b-qt-qat Mean [95%CI]: 26.571 [23.725, 29.417] Min = 4.566, Max = 55.457 Median [95%CI]: 23.663 [21.337, 25.328]	LLM: cpu_arctic_gemma3_1b-qt-qat Mean [95%CI]: 30.254 [25.715, 34.793] Min = 11.451, Max = 83.961 Median [95%CI]: 23.203 [20.82, 28.955]
GPU	LLM: cpu_arctic_gemma3_1b-qt-qat Mean [95%CI]: 2.687 [2.330, 3.043] Min = 1.033, Max = 8.243 Median [95%CI]: 2.354 [1.92, 2.762]	LLM: cpu_arctic_gemma3_1b-qt-qat Mean [95% CI]: 2.867 [2.442, 3.293] Min = 1.049, Max = 9.761 Median [95% CI]: 2.395 [2.111, 2.758]
CPU	LLM: cpu_arctic_gemma3_4b-qt-qat Mean [95%CI]: 93.405 [82.906, 103.903] Min = 15.272, Max = 197.777 Median [95%CI]: 86.734 [78.403, 99.325]	-

Table 17. Generation Response Latency comparison across configurations

Common settings	**Settings to compare**		**Comparison**
Hardware=CPU, LLM= gemma3:1b-it-qat	snowflake-arctic-embed2	bge-m3	Normality: False Wilcoxon test p-value: 0.173
Hardware=GPU, LLM= gemma3:1b-it-qat	snowflake-arctic-embed2	bge-m3	Normality: False Wilcoxon test p-value: 0.317
Embed model=snowflake-arctic-embed2, LLM= gemma3:1b-it-qat	CPU	GPU	Normality: False Wilcoxon test p-value: **0.000 ***
Embed model=bge-m3, LLM= gemma3:1b-it-qat	CPU	GPU	Normality: False Wilcoxon test p-value: **0.000 ***
CPU, Embed model=snowflake-arctic-embed2	LLM= gemma3:1b-it-qat	LLM= gemma3:4b-it-qat	Normality: False Wilcoxon test p-value: **0.000 ***

Figure 13. Response Generation Latency comparison between CPU and GPU

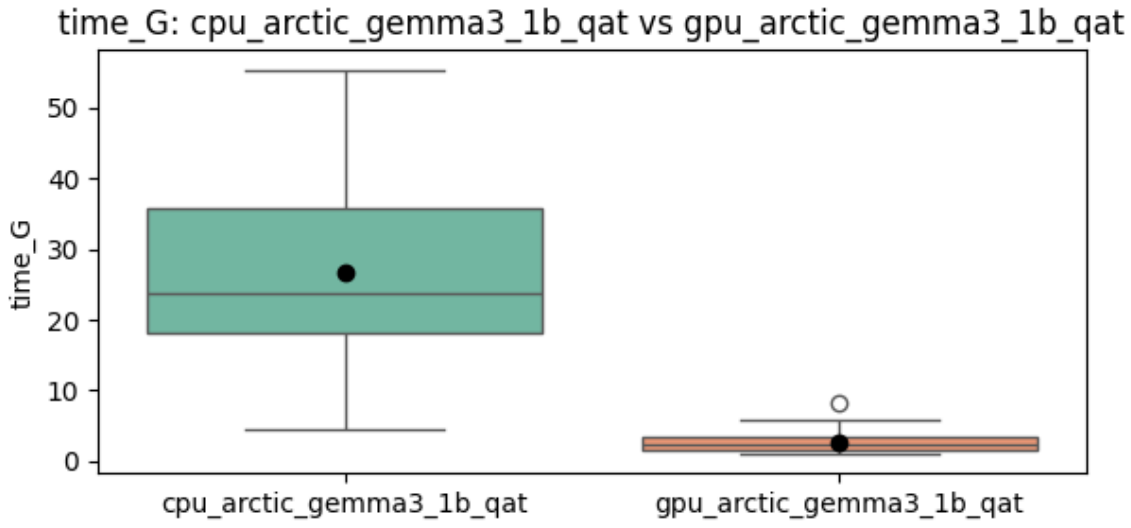
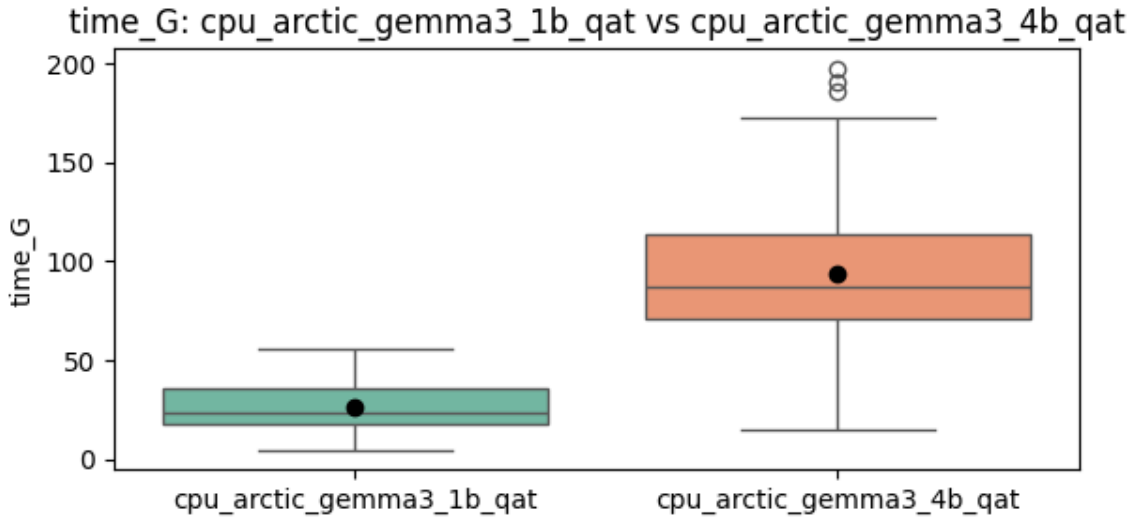


Figure 14. Effect of LLM size on Response Generation Latency (CPU)



4.2.2 Evaluating the quality of the RAG prototype (Research question 4)

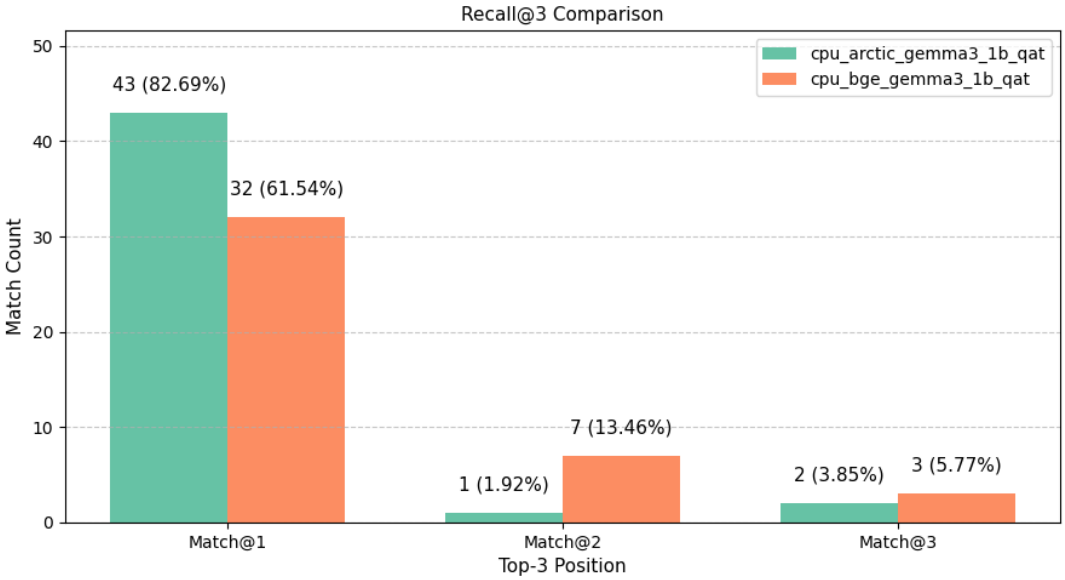
Based on the designed questionnaire (Table 9), data analysis is planned to use relevant subsets of questions for each of the metrics. Recall@3, based on labelled questions, is analyzed based on questions 1 to 52. Other quality metrics are performed on all the questions.

Recall@3: The results show 88.46% for snowflake-arctic-embed2 and 80.77% for bge-m3, the difference is not statistically significant. Further analysis reveal that snowflake-arctic-embed2 achieves a statistically significantly higher top 1 matching (also known as Recall@1) compared to bge-m3. This value also exceeds bge-m3's Recall@3 numerically (Table 18 and Figure 15). These findings contribute to the decision to select snowflake-arctic-embed2 for the subsequent qualitative analysis.

Table 18. Recall@3 for the two embedding models: snowflake-arctic-embed2 and bge-m3

	snowflake-arctic-embed2	**bge-m3**	**McNemar test**
Recall@3	46/52 (88.46%)	42/52 (80.77%)	Contingency table: [[0, 4], [0, 0]] p-value: 0.125
Recall@1	43/52 (82.69%)	32/52 (61.54%)	Contingency table: [[0, 12], [1, 0]] p-value: **0.003**

Figure 15. Recall@3 for the two embedding models: snowflake-arctic-embed2 and bge-m3



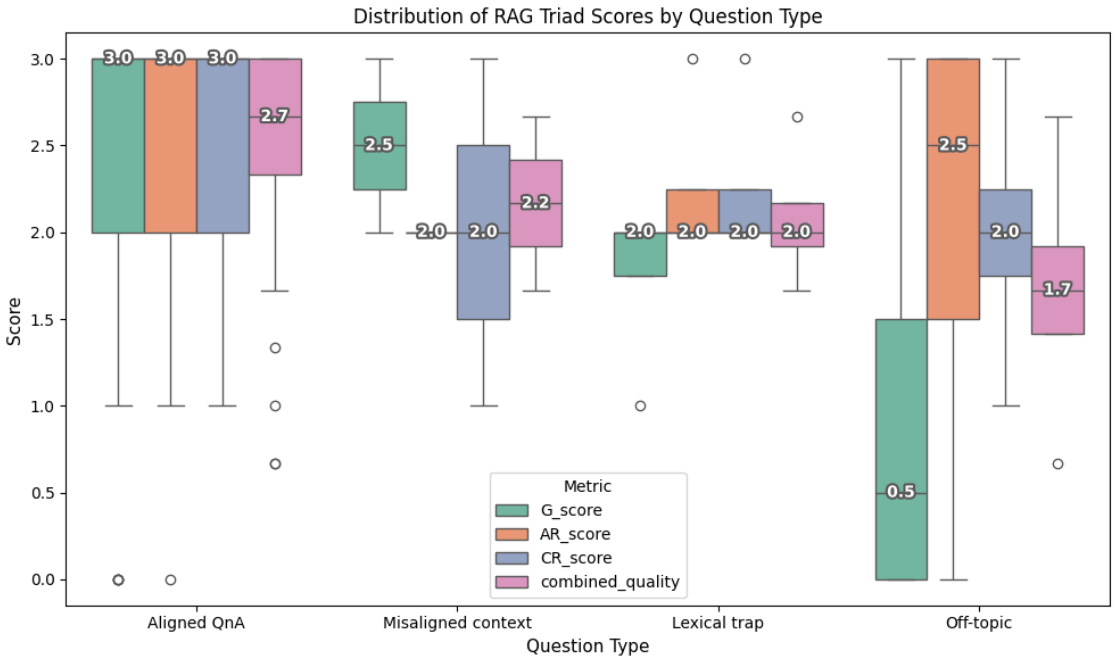
While Recall@3 is a useful retrieval metric, there are several potential limitations to consider. If recall = 0 (the labelled chunk does not match the top 3 retrieved chunks), it does not necessarily mean a retrieval failure. Investigation should be done to verify any knowledge gap during labelling, where better matching chunks were ignored. On the other hand, a successful recall = 1 does not guarantee a good context provided to the LLM. There might be other better chunks, but the retrieved one is not the best. The retrieved chunk may be diluted in the context built from different chunks; or the chunk is not sufficiently clear or specific, leading to confusion for the LLM. (Pinecone.io, n.d.-a)

During experimentation, a query is observed to produce different chunk rankings when run on CPU versus GPU systems. Specifically, for question #43 in the study, the IndexFlatL2 scores for chunk A and chunk B are 100.24 and 100.25 on one system, but they are 100.54 and 100.49 on the other machine, causing their ranking positions to swap. Although this change did not affect the Recall@3 since both chunks remained within the top k (k=3), similar small variations in floating-point computations could impact retrieval accuracy if they occurred between ranks k and k+1. This highlights the sensitivity of nearest-neighbor search to floating-point non-determinism across hardware platforms.

RAG outcome evaluation based on the RAG Triad: The evaluation questionnaire composing of 60 questions are presented in section 3.3. It includes 50 questions categorized as “Aligned QnA”, 2 as “Misaligned context”, 4 as “Lexical trap”, and 4 as “Off-topic”. Quality analysis is performed by category to evaluate the RAG system appropriately. Recorded RAG answers are classified into 5 categories following the criteria described in Table 13. All the categories are based solely on metric generated by the batch inference module and the AI judge. Another quality of RAG outcome, the “Abstention”, requires human review and is discussed separately.

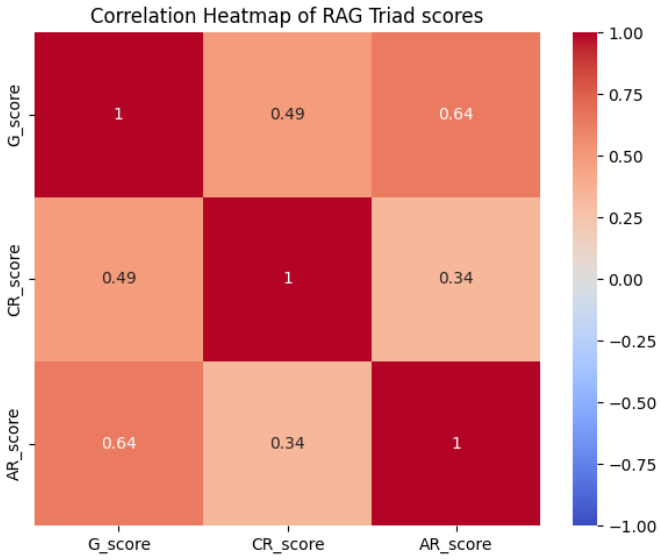
RAG Triad Distribution: The boxplot analysis (Figure 16) shows clear trends in system performance across question types. For aligned, in-scope queries, all three metrics (Groundedness, Answer Relevance, and Context Relevance) are high, indicating accurate and contextually appropriate answers. In contrast, misaligned and lexical trap questions show lower scores across the board, revealing difficulties with misleading or mismatched contexts. Off-topic queries, while maintaining moderate relevance scores, suffer from remarkably low Groundedness, pointing to a risk of hallucination despite surface-level relevance. These patterns highlight the need for better context handling and Abstention strategies.

Figure 16. Distribution of RAG Triad scores



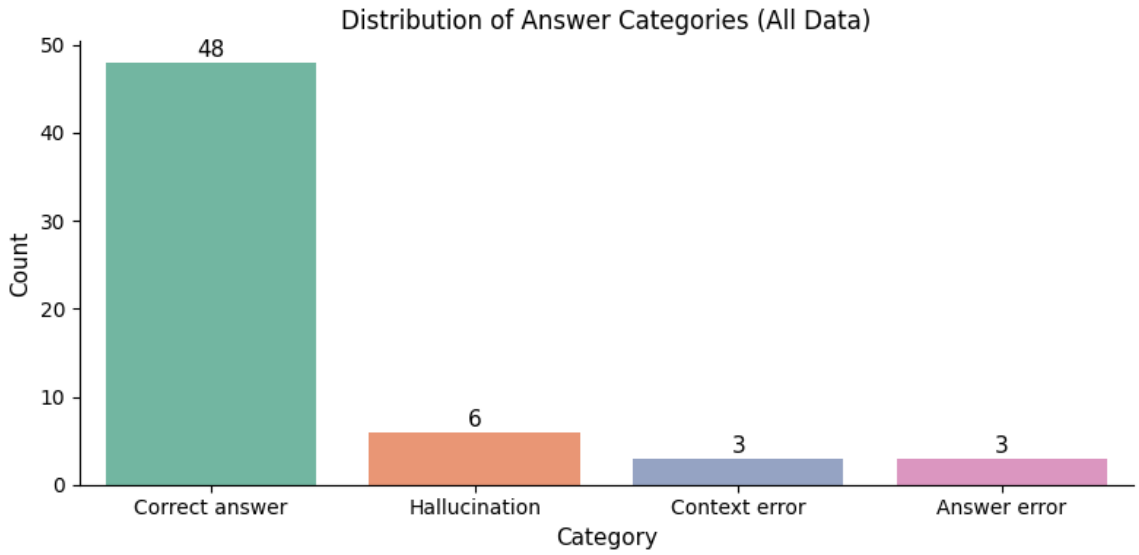
RAG Triad scores correlation: The correlation heatmap (Figure 17) reveals moderate-to-strong links between the metrics. Groundedness correlates well with Answer Relevance (r = 0.64) and moderately with Context Relevance (r = 0.49). The weakest link is between Context and Answer Relevance (r = 0.34), suggesting they assess distinct aspects of performance. This underscores the importance of using all three metrics for a comprehensive evaluation.

Figure 17. Correlation heatmap of the RAG Triad scores



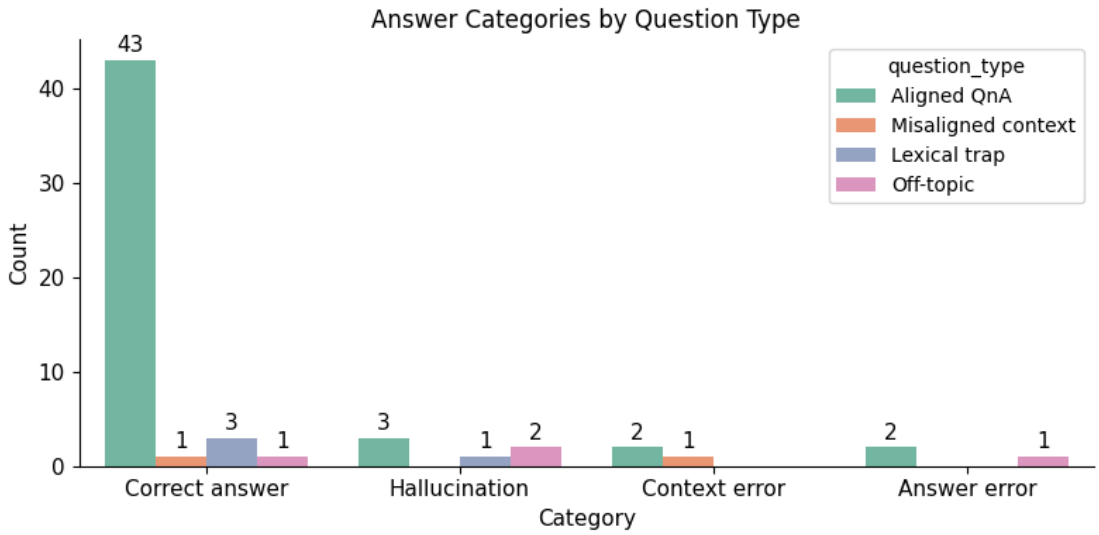
RAG output quality evaluated using the RAG Triad: To evaluate the quality of RAG outcomes, the RAG Triad scores is used, as described in section 3.2.1, to classify the outcome into 5 categories. The majority of responses (80%) were correct, aligning well with high Triad scores, while a smaller portion exhibited issues: 10% showed hallucinations, 5% had context errors, and 5% had answer errors (Figure 18). These error categories, by criteria, correspond with lower Groundedness or Relevance scores, demonstrating that the RAG Triad effectively captures different failure modes and provides a reliable framework for assessing RAG answer quality.

Figure 18. Distribution of RAG outcome categories as classified using RAG Triad scores



When analyzed by question type, the error distribution further illustrates how specific query characteristics influence RAG performance. Aligned QnA gets the highest correctness rate (86%), with minimal errors, confirming strong system performance on well-formed, in-scope queries. Lexical trap questions see a slight drop, with 25% hallucinations, indicating vulnerability to misleading phrasing. Misaligned context questions show an even split between correct answers and context errors, underscoring the impact of retrieval mismatches. Off-topic queries perform worst, with 50% hallucinations and only 25% correctness, highlighting the risk of ungrounded responses when faced with out-of-scope inputs. These patterns reinforce the importance of careful context handling and robust safeguards against irrelevant or deceptive queries (Figure 19).

Figure 19. Distribution of RAG output categories by question type



The batched answers analyzed in this study reveal hallucinations in response to tricky or deceptive questions and show no instances of abstention, as verified through human review, highlighting limitations in the quality and control mechanisms of the 1B-parameter LLM used. However, during system development, the same model occasionally did exhibit abstention behavior, suggesting a degree of non-determinism in generation. Larger models, such as 4B or 7B variants, were observed to handle abstention more consistently and reliably, indicating that model scale plays a key role in managing uncertainty and producing safer, more grounded outputs.

This evaluation highlights both strengths and limitations of the RAG system. The pipeline performs reliably on well-formed, aligned questions, achieving 80% correct answers and high RAG Triad scores across Groundedness, Answer Relevance, and Context Relevance, mainly contributed by the dominant subset of Aligned questions. The Triad metrics show meaningful correlations and effectively differentiate between correct and erroneous responses, confirming their value as evaluation tools. Retrieval performance is strong overall, with snowflake-arctic-embed2 outperforming bge-m3 on Recall@1 significantly, justifying its use in the qualitative phase.

However, weaknesses emerge in handling edge cases. Tricky, off-topic, or misaligned queries often result in hallucinations or context errors. Notably, the 1B LLM used in the final batched run produced no abstentions, as verified by human review, despite earlier development observations showing occasional abstention behavior. During development, repeatedly entering the same tricky query yielded both abstentions and erroneous answers,

underscoring model instability on difficult inputs. This inconsistency reflects both the non-deterministic nature of LLM and the limited ability of small language models to manage uncertainty. In contrast, larger models (4B and 7B) demonstrated more reliable abstention behavior, suggesting that scale improves safety and response control.

Additionally, this highlights the significant sensitivity of LLMs to prompt phrasing. Small changes in prompt wording can markedly affect output quality, making prompt adjustment, systematic testing, and query paraphrasing critical strategies to improve response accuracy and reduce hallucinations. Iteratively refining prompts and exploring alternative formulations can better align the model's understanding with user intent, particularly for challenging or ambiguous queries.

To address these challenges, improvements are needed in several areas. Retrieval could benefit from semantic chunking and better context filtering to reduce dilution and irrelevance. Generation quality could be enhanced with stronger abstention logic and uncertainty-aware mechanisms. Finally, evaluation would be strengthened by incorporating human review and abstention detection alongside automated metrics, aligning with recent best practices in RAG evaluation.

4.3 Limitations and improvement options

The current prototype has several limitations. It is developed as a proof of concept for further development and provides a framework for flexible evaluation of different components such as the embedding and the language models. The runtime environment has been restricted to Ollama, and LlamaIndex has been the sole orchestration framework used. Future work should involve testing across different environments, such as LangChain or Haystack and other components (Table 6) to assess portability, performance, and compatibility with enterprise-grade tools.

The development and evaluation are centered around a single source document, albeit a challenging API software documentation combining Finnish language with English technical terminology. While this setup served well for initial experimentation, additional adaptation and testing are required to enable the chatbot to handle a wider variety of documents, particularly those with different structures, domains, or multilingual content.

The current pipeline lacks comprehensive error handling and robustness under failure conditions. For example, it does not yet manage retrieval failures gracefully. Fallback

strategies instructed in the prompt template is effective with larger models but suboptimal with small LLM chosen for low-end hardware. On real-world usage, adding user feedback mechanisms would bring opportunities for iterative improvement.

The evaluation performed is limited in the diversity of the questionnaire, the number of models and datasets evaluated. The use of minimal number of challenging questions effectively revealed the weakness of smaller language model. However, more robust testing and adjustment would bring more insight about the RAG prototype. Another limitation lies in the limited diversity of evaluation metrics. While the RAG Triad provides useful insight into grounding and relevance, it does not fully capture user satisfaction, factual correctness at a more fine-grained level, or the effectiveness of abstention behavior. Human-in-the-loop review is implemented in the project, should continue, together with task success metrics, and longitudinal usage data would provide a more holistic picture of performance.

Prompt engineering remains underdeveloped, though some adjustments were made during testing. Given the observed sensitivity of small LLM responses to prompt phrasing, especially in ambiguous cases, integrating prompt tuning, automatic paraphrasing, and A/B testing could meaningfully enhance reliability and reduce hallucination rates.

Finally, while using an AI judge offers a practical and scalable solution for evaluating system responses, the AI may introduce evaluation errors or biases, particularly in edge cases or nuanced contexts. Additionally, the developer’s limited fluency in Finnish, which was openly acknowledged with the commissioning company at the outset, may have affected some language-specific aspects of implementation or evaluation. These factors highlight the need for future improvements, such as incorporating human-in-the-loop validation and involving native speakers in critical evaluation phases.

5 Conclusions

This thesis set out to design, implement, and evaluate an offline, open-source RAG-based AI chatbot for Finnish-language software documentation on low-end hardware. The primary goal was to improve the usability and accessibility of technical documentation for developers, specifically within the context of Triplan Oy’s Tweb REST API manual. The research demonstrated that a local, privacy-conscious chatbot, leveraging Retrieval-

Augmented Generation (RAG) and offline large language models, can provide accurate, context-aware support in Finnish, even when constrained by modest hardware resources.

The developed model-agnostic prototype successfully integrated key components: Ollama for local inference, FAISS for efficient vector search, and llama-index for managing retrieval workflows. The system emphasized modularity and language-specific optimization, delivered through a command-line interface to maximize accessibility and performance on low-end devices. Iterative development and user-centered design ensured that the solution aligned with real-world needs, while prompt engineering and prompt fine-tuning addressed the unique challenges of Finnish-language processing.

Evaluation results confirmed the feasibility of the approach. The chatbot demonstrated acceptable latency and response quality, with performance metrics indicating that local, offline operation is practical for supporting Finnish-language software documentation. The system maintains data security and privacy by processing all information locally, in compliance with ethical and organizational requirements.

The current prototype, developed as a proof of concept, offers a flexible framework for evaluating RAG components but has several limitations. Evaluation was constrained by a limited range of test queries, models, and datasets, and focused on a single, complex source document, which restricts generalizability. The system is tied to a specific runtime (Ollama) and orchestration framework (LlamaIndex), lacks comprehensive error handling, and omits user feedback loops. Evaluation metrics are limited, and prompt engineering remains basic despite its critical impact on small LLM performance. AI-based evaluation, while scalable, may introduce errors, and the developer’s limited Finnish fluency could influence language-specific implementation. Future work should address these issues through broader testing, improved robustness, human-in-the-loop feedback, and enhanced evaluation strategies.

In conclusion, this thesis provides a practical foundation for secure, offline AI chatbots serving documentation in Finnish and other languages supported by the selected embedding models and LLMs. The prototype demonstrates the feasibility of deploying advanced language technologies in resource-constrained, privacy-sensitive environments, while also highlighting areas for future research and development.

6 Summary

This thesis successfully addressed the usability challenges of traditional software documentation by designing and evaluating an offline, Finnish-language RAG-based chatbot in collaboration with Triplan Oy. Through a rapid prototyping approach, the development process emphasized modularity, language-specific configuration, and adaptability to low-end hardware. Key considerations included selecting lightweight components, such as Ollama, FAISS, and LlamaIndex, and maintaining system transparency and reusability, aligning with the project's confidentiality and sustainability goals.

Critical data preparation steps were identified, particularly for handling Finnish-language documentation. These included managing special character encodings and applying section-based chunking strategies to maintain contextual coherence in retrieval and generation. The evaluation of system performance demonstrated efficient processing across core metrics: vector store building completed quickly, semantic retrieval achieved sub-second latency on CPU, and response generation latency varied significantly with hardware and model size, with GPU providing a substantial performance advantage.

Output quality, measured using Recall@3 and an AI judge scoring groundedness, context relevance, and answer relevance, confirmed the chatbot's effectiveness for clear, well-aligned queries. However, smaller language models exhibited limitations in handling ambiguity and maintaining reliable abstention behavior, occasionally resulting in hallucinated responses. Despite these limitations, the prototype demonstrated the feasibility of secure, offline AI support for localized documentation, offering a functional starting point for broader deployment.

Overall, the findings confirm that privacy-preserving, local AI chatbots are a viable solution for enhancing Finnish-language software documentation in resource-constrained, privacy-sensitive environments. The model-agnostic prototype lays a strong foundation for future research, with opportunities to improve prompt engineering, expand usability, integrate user feedback and learning capabilities, and validate the approach across more diverse evaluation settings and documentation contexts.

References

Agrawal, G., Kumarage, T., Alghamdi, Z., & Liu, H. (2024). Mindful-RAG: A Study of Points of Failure in Retrieval Augmented Generation (arXiv:2407.12216; Version 2). arXiv. https://doi.org/10.48550/arXiv.2407.12216

AI-ANK. (n.d.). RAGArch. https://github.com/AI-ANK/RAGArch

Alammar, J., & Grootendorst, M. (2024). Hands-On Large Language Models. https://learning.oreilly.com/library/view/hands-on-large-language/9781098150952/

Anisuzzaman, D. M., Malins, J. G., Friedman, P. A., & Attia, Z. I. (2025). Fine-Tuning Large Language Models for Specialized Use Cases. Mayo Clinic Proceedings: Digital Health, 3(1), 100184. https://doi.org/10.1016/j.mcpdig.2024.11.005

Barnett, S., Kurniawan, S., Thudumu, S., Brannelly, Z., & Abdelrazek, M. (2024). Seven Failure Points When Engineering a Retrieval Augmented Generation System (arXiv:2401.05856). arXiv. https://doi.org/10.48550/arXiv.2401.05856

Belcic, I., & Stryker, C. (n.d.). Llamaindex vs Langchain: What's the difference? | IBM. Retrieved May 16, 2025, from https://www.ibm.com/think/topics/llamaindex-vs-langchain

Berryman, J., & Ziegler, A. (2024, November). Prompt Engineering for LLMs: The Art and Science of Building Large Language Model-Based Applications. https://learning.oreilly.com/library/view/prompt-engineering-for/9781098156145/

Borek, C. (2024). Comparative Evaluation of LLM-Based Approaches to Chatbot Creation: Implementing a Death Doula Chatbot. https://trepo.tuni.fi/handle/10024/154995

Brack, M. (2024, March 11). Tokenizer Evaluation on European Languages. https://occiglot.eu/posts/eu_tokenizer_perfomance/

Brehme, L., Ströhle, T., & Breu, R. (2025). Can LLMs Be Trusted for Evaluating RAG Systems? A Survey of Methods and Datasets (arXiv:2504.20119; Version 2). arXiv. https://doi.org/10.48550/arXiv.2504.20119

Chan, B. J., Chen, C.-T., Cheng, J.-H., & Huang, H.-H. (2024). Don't Do RAG: When Cache-Augmented Generation is All You Need for Knowledge Tasks. https://doi.org/10.1145/3701716.3715490

Chelombitko, I., & Komissarov, A. (2024). Specialized Monolingual BPE Tokenizers for Uralic Languages Representation in Large Language Models. In M. Hämäläinen, F. Pirinen, M. Macias, & M. Crespo Avila (Eds.), Proceedings of the 9th International Workshop on Computational Linguistics for Uralic Languages (pp. 89–95). Association for Computational Linguistics. https://aclanthology.org/2024.iwclul-1.11/

Chen, J., Xiao, S., Zhang, P., Luo, K., Lian, D., & Liu, Z. (2024). BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation (arXiv:2402.03216). arXiv. https://doi.org/10.48550/arXiv.2402.03216

Chip, H. (2024). AI Engineering. https://learning.oreilly.com/library/view/ai-engineering/9781098166298/

Chroma. (n.d.). Chroma Docs. Chroma Docs. Retrieved May 31, 2025, from https://docs.trychroma.com

Conda. (n.d.). Conda Documentation—Conda-docs documentation. Retrieved May 31, 2025, from https://docs.conda.io/en/latest/

Cuantum Technologies LLC. (2025). Natural Language Processing with Python. https://learning.oreilly.com/library/view/natural-language-processing/9781837021635/

DeepsetAI. (n.d.). Deepset-ai/haystack: AI orchestration framework to build customizable, production-ready LLM applications. Connect components (models, vector DBs, file converters) to pipelines or agents that can interact with your data. With advanced retrieval methods, it's best suited for building RAG, question answering, semantic search or conversational agent chatbots. Retrieved May 31, 2025, from https://github.com/deepset-ai/haystack

Doner, J. (2025). The Linguistic Analysis of Word and Sentence Structures. University of Manitoba. https://pressbooks.openedmb.ca/wordandsentencestructures/

European Parliamentary Research Service. (2019, September). EU guidelines on ethics in artificial intelligence: Context and implementation. EPRS_BRI(2019)640163

Facebook Research. (n.d.-a). Guidelines to choose an index. GitHub. Retrieved May 16, 2025, from https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index

Facebook Research. (n.d.-b). MetricType and distances. GitHub. Retrieved May 27, 2025, from https://github.com/facebookresearch/faiss/wiki/MetricType-and-distances

GitHub. (n.d.). GitHub. GitHub. Retrieved May 23, 2025, from https://github.com

Goodhart, G. (n.d.). Build a local AI co-pilot using IBM Granite Code, Ollama, and Continue. IBM Developer. Retrieved May 16, 2025, from https://developer.ibm.com/tutorials/awb-local-ai-copilot-ibm-granite-code-ollama-continue/

Google AI for Developers. (n.d.-a). Gemma formatting and system instructions. Google AI for Developers. Retrieved May 16, 2025, from https://ai.google.dev/gemma/docs/core/prompt-structure

Google AI for Developers. (n.d.-b). Gemma models overview. Google AI for Developers. Retrieved May 28, 2025, from https://ai.google.dev/gemma/docs

Google AI for Developers. (2025). Google AI Gemma open models | Google for Developers. Google AI for Developers. https://ai.google.dev/gemma

Google NotebookLM. (n.d.). Google NotebookLM. Retrieved May 23, 2025, from https://notebooklm.google.com/

Haystack. (n.d.). Introduction to Haystack. Retrieved May 31, 2025, from https://docs.haystack.deepset.ai/docs/intro

Hugging Face. (n.d.). Sentence-transformers/distiluse-base-multilingual-cased-v2 · Hugging Face. Retrieved May 31, 2025, from https://huggingface.co/sentence-transformers/distiluse-base-multilingual-cased-v2

HuggingFace. (n.d.). Sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2. Retrieved May 31, 2025, from https://huggingface.co/sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2

IBM. (n.d.-a). NLP vs. NLU vs. NLG: What's the Difference? Retrieved May 2, 2025, from https://www.ibm.com/think/topics/nlp-vs-nlu-vs-nlg

IBM. (n.d.-b). RAG Cookbook. https://www.ibm.com/architectures/papers/rag-cookbook?utm_source=ibm_developer&utm_content=in_content_link&utm_id=blogs_awb-introduci%E2%80%A6

IBM. (n.d.-c). Retrieval Augmented Generation. Retrieved May 16, 2025, from https://www.ibm.com/architectures/patterns/genai-rag

IBM AI Academy. (2024, May 15). Choose AI Model For Use Case. https://www.ibm.com/think/videos/ai-academy/choose-ai-model-for-use-case

InternLM. (2025). InternLM/lmdeploy [Python]. InternLM. https://github.com/InternLM/lmdeploy (Original work published 2023)

Java. (n.d.). Java Documentation. Oracle Help Center. Retrieved May 31, 2025, from https://docs.oracle.com/en/java/

Jegou, H., Douze, M., & Johnson, J. (2017, March 29). Faiss: A library for efficient similarity search. Engineering at Meta. https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/

JetBrains. (n.d.). PyCharm Documentation. PyCharm Help. Retrieved May 31, 2025, from https://www.jetbrains.com/help/pycharm/getting-started.html

Jeya, G. R. M. (n.d.). Accessing IBM Granite LLMs in Jupyter Notebook through Ollama. IBM Developer. Retrieved May 16, 2025, from https://developer.ibm.com/tutorials/awb-accessing-llms-jupyter-notebook-ollama/

JupyterLab. (n.d.). JupyterLab Documentation—JupyterLab 4.5.0a0 documentation. Retrieved May 31, 2025, from https://jupyterlab.readthedocs.io/en/latest/

Jurafsky, D., & Martin, J. H. (2025). Speech and Language Processing An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models (3rd ed.). https://web.stanford.edu/~jurafsky/slp3/

Kanerva, J. (2024). Understanding the structure and meaning of Finnish texts: From corpus creation to deep language modelling. University of Turku.

Khan, A. A., Hasan, M. T., Kemell, K. K., Rasku, J., & Abrahamsson, P. (2024). Developing Retrieval Augmented Generation (RAG) based LLM Systems from PDFs: An Experience Report (arXiv:2410.15944). arXiv. https://doi.org/10.48550/arXiv.2410.15944

LangChain. (n.d.-a). BM25. Retrieved May 31, 2025, from https://python.langchain.com/docs/integrations/retrievers/bm25/

LangChain. (n.d.-b). LangChain Introduction. Retrieved May 31, 2025, from https://python.langchain.com/docs/introduction/

Langflow AI. (n.d.). langflow-ai/langflow: Langflow is a powerful tool for building and deploying AI-powered agents and workflows. Retrieved May 31, 2025, from https://github.com/langflow-ai/langflow/tree/main

Llamacpp. (n.d.). Getting Started—Llama-cpp-python. Retrieved May 31, 2025, from https://llama-cpp-python.readthedocs.io/en/latest/

LlamaIndex. (n.d.-a). LlamaIndex—LlamaIndex. Retrieved May 31, 2025, from https://docs.llamaindex.ai/en/stable/

LlamaIndex. (n.d.-b). Retriever Modules—LlamaIndex. Retrieved May 30, 2025, from https://docs.llamaindex.ai/en/stable/module_guides/querying/retriever/retrievers/

LlamaIndex. (n.d.-c). Retriever Query Engine with Custom Retrievers—Simple Hybrid Search [Dataset]. Retrieved May 31, 2025, from https://docs.llamaindex.ai/en/stable/examples/query_engine/CustomRetrievers/

LlamaIndex. (n.d.-d). Vector Index Retriever. Retrieved May 31, 2025, from https://docs.llamaindex.ai/en/stable/api_reference/retrievers/vector/#llama_index.core.retrievers.VectorIndexRetriever

Meng, M., Steinhardt, S., & Schubert, A. (2018). Application Programming Interface Documentation: What Do Software Developers Want? Journal of Technical Writing and Communication, 48(3), 295–330. https://doi.org/10.1177/0047281617721853

Meng, M., Steinhardt, S., & Schubert, A. (2019). How developers use API documentation: An observation study. Commun. Des. Q. Rev, 7(2), 40–49. https://doi.org/10.1145/3358931.3358937

Meta. (n.d.-a). Llama 3.1 | Model Cards and Prompt formats. Retrieved May 16, 2025, from https://www.llama.com/docs/model-cards-and-prompt-formats/

Meta. (n.d.-b). Llama Documentation. Retrieved May 28, 2025, from https://www.llama.com/docs/overview/

Meta Engineering. (2018, January 24). Under the hood: Multilingual embeddings. Engineering at Meta. https://engineering.fb.com/2018/01/24/ml-applications/under-the-hood-multilingual-embeddings/

Microsoft. (n.d.). Documentation for Visual Studio Code. Retrieved May 30, 2025, from https://code.visualstudio.com/docs

Microsoft Copilot. (n.d.). Microsoft Copilot: Your AI companion. Microsoft Copilot: Your AI Companion. Retrieved May 23, 2025, from https://copilot.microsoft.com

Milvus. (n.d.). Milvus vector database documentation. Retrieved May 31, 2025, from https://milvus.io/docs

Mistral AI. (n.d.-a). Frontier AI LLMs, assistants, agents, services. Retrieved May 16, 2025, from https://mistral.ai/

Mistral AI. (n.d.-b). Mistral AI Documentation | Mistral AI Large Language Models. Retrieved May 28, 2025, from https://docs.mistral.ai/

Mistral AI. (2025, April 9). Evaluating RAG with LLM as a Judge. https://mistral.ai/news/llm-as-rag-judge

Mistral AI Large Language Models. (n.d.). Prompting capabilities. Retrieved May 16, 2025, from https://docs.mistral.ai/guides/prompting_capabilities/

Mozilla. (2025, April 3). JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript

n8n. (n.d.). Explore n8n Docs: Your Resource for Workflow Automation and Integrations | n8n Docs. Retrieved May 31, 2025, from https://docs.n8n.io/

Nassif, M., & Robillard, M. P. (2023). A Field Study of Developer Documentation Format. Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems, 1–7. https://doi.org/10.1145/3544549.3585767

NomicAI. (2025). Nomic-ai/gpt4all [C++]. Nomic AI. https://github.com/nomic-ai/gpt4all (Original work published 2023)

Ollama. (n.d.-a). Ollama Models. Retrieved May 28, 2025, from https://www.ollama.com/search

Ollama. (n.d.-b). Ollama/docs. GitHub. Retrieved May 31, 2025, from https://github.com/ollama/ollama/tree/main/docs

Ollama. (2025). Ollama GitHub page [Go]. Ollama. https://github.com/ollama/ollama (Original work published 2023)

Oracle. (n.d.). Oracle VirtualBox. Oracle Help Center. Retrieved May 31, 2025, from https://docs.oracle.com/en/virtualization/virtualbox/

Petrov, A., Malfa, E. L., Torr, P. H. S., & Bibi, A. (2023). Language Model Tokenizers Introduce Unfairness Between Languages (arXiv:2305.15425). arXiv. https://doi.org/10.48550/arXiv.2305.15425

Pinecone.io. (n.d.-a). Evaluation Measures in Information Retrieval. Retrieved May 30, 2025, from https://www.pinecone.io/learn/offline-evaluation/

Pinecone.io. (n.d.-b). Vector Similarity Explained. Retrieved May 27, 2025, from https://www.pinecone.io/learn/vector-similarity/

Python. (n.d.). Python 3.13 documentation. Python Documentation. Retrieved May 31, 2025, from https://docs.python.org/3/

Qdrant. (n.d.). Home—Qdrant. Retrieved May 31, 2025, from https://qdrant.tech/documentation/

Ragas. (n.d.). explodinggradients/ragas: Supercharge Your LLM Application Evaluations 🚀. Retrieved May 31, 2025, from https://github.com/explodinggradients/ragas

Raglianti, M., Nagy, C., Minelli, R., Lin, B., & Lanza, M. (2023). On the Rise of Modern Software Documentation (Pearl/Brave New Idea) [Application/pdf]. LIPIcs, Volume 263, ECOOP 2023, 263, 43:1-43:24. https://doi.org/10.4230/LIPICS.ECOOP.2023.43

Rohde, F., Wagner, J., Meyer, A., Reinhard, P., Voss, M., Petschow, U., & Mollen, A. (2024). Broadening the perspective for sustainable artificial intelligence: Sustainability criteria and indicators for Artificial Intelligence systems. Current Opinion in Environmental Sustainability, 66, 101411. https://doi.org/10.1016/j.cosust.2023.101411

Sahadevan, V., Joshi, R., Borg, K., Singh, V., Singh, A. R., Muhammed, B., Beemaraj, S. B., & Joshi, A. (2025). Knowledge augmented generalizer specializer: A framework for early stage design exploration. Advanced Engineering Informatics, 65, 103141. https://doi.org/10.1016/j.aei.2025.103141

Smith, J. (2022). Human-Computer Interaction: Enhancing User Experience and Productivity. International Multidisciplinary Journal Of Science, Technology & Business, 1(4), Article 4.

Spotify. (2025). Spotify/annoy [C++]. Spotify. https://github.com/spotify/annoy (Original work published 2013)

Stackoverflow. (n.d.). Stackoverflow. Stack Overflow. Retrieved May 23, 2025, from https://stackoverflow.com/questions

Stanfordnlp. (2025). Stanfordnlp/dspy [Python]. Stanford NLP. https://github.com/stanfordnlp/dspy (Original work published 2023)

Tao, C., Shen, T., Gao, S., Zhang, J., Li, Z., Tao, Z., & Ma, S. (2024). LLMs are Also Effective Embedding Models: An In-depth Overview (arXiv:2412.12591; Version 1). arXiv. https://doi.org/10.48550/arXiv.2412.12591

Tiktokenizer. (n.d.). Tiktokenizer. Retrieved May 24, 2025, from https://tiktokenizer.vercel.app/?model=google%2Fgemma-7b

Triplan Oy. (n.d.). Triplan Oy Github Page. GitHub. Retrieved May 23, 2025, from https://github.com/triplanoy

TruLens. (n.d.). RAG Triad—TruLens. Retrieved May 31, 2025, from https://www.trulens.org/getting_started/core_concepts/rag_triad/

Ubuntu. (n.d.). Official Ubuntu Documentation. Retrieved May 31, 2025, from https://help.ubuntu.com/

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. ukasz, & Polosukhin, I. (2017). Attention is All you Need. Advances in Neural Information Processing Systems, 30.

Vllm project. (2025). Vllm-project/vllm [Python]. vLLM. https://github.com/vllm-project/vllm (Original work published 2023)

Weaviate. (n.d.). Weaviate Docs. Retrieved May 31, 2025, from https://weaviate.io/developers/weaviate

Yu, P., Merrick, L., Nuti, G., & Campos, D. (2024). Arctic-Embed 2.0: Multilingual Retrieval Without Compromise (arXiv:2412.04506). arXiv. https://doi.org/10.48550/arXiv.2412.04506

Zhao, S., Yang, Y., Wang, Z., He, Z., Qiu, L. K., & Qiu, L. (2024). Retrieval Augmented Generation (RAG) and Beyond: A Comprehensive Survey on How to Make your LLMs use External Data More Wisely (arXiv:2409.14924). arXiv. https://doi.org/10.48550/arXiv.2409.14924

Appendix 1. Data management plan

This project involves managing various types of materials and tools:

1. Source Materials: Tweb REST API documentation (Word and PDF formats) provided by Triplan, protected under a Non-Disclosure Agreement (NDA).

2. Software and Environment:

Python 3.10 for Retrieval-Augmented Generation (RAG) development.
Python 3.12 for evaluation tasks.
Anaconda and Miniconda used for environment management.
Ubuntu server used for hosting the virtual machine.

3. Project Files: A structured project directory containing Jupyter notebooks (data_pipeline, rag_pipeline, batch_inference_module, ai_judge), README.md, configuration files (config.yml, environment.yml), the vector store, and .gitignore.

4. Documentation: Includes research notes, meeting minutes, thesis drafts (Word), and presentation slides (PPTX) used throughout the project and for the final defense.

5. Communication and Collaboration: Team interactions, decisions, and action items are tracked via Microsoft Teams meetings and group chat discussions.

6. Project Tracking and Metrics: Key performance indicators, outcomes, and achievements are systematically recorded to monitor and demonstrate project success.

All project data are organized in a dedicated directory on the author's computer (drive F), which includes source documents, code, configuration files, notebooks, and documentation. This directory is regularly backed up to two external hard drives to ensure data security and redundancy. Version control is maintained using Git, and environment dependencies are managed with Anaconda and Miniconda. Communication records and decisions are documented via Microsoft Teams. Sensitive materials, such as the Tweb REST API documentation, are protected under an NDA and handled accordingly.

Appendix 2. Program codes

Program code 3. Configuration file

```
1.
2.  retrieval_top_k: 3
3.  display_limit: 3
4.  body_preview_length: 100
5.
6.  exit_command: "poistu"
7.  rephrase_message: "En löytänyt suoraa vastausta tästä Triplan Oy:n dokumentista.
    Haluatko tarkentaa kysymystäsi?"
8.  prompt_control_command: "okhgn_hte_art_iol" # Intentionally nonsensical
9.
10. prompt_instruction: >
11.   Follow these rules:
12.     1. Base your answer strictly on the provided context to answer the question.
13.     2. If no relevant information is found: {prompt_control_command}.
14.     3. Answer in Finnish.
15.
16. prompt_instruction_g: >
17.  You are a judge for evaluating a RAG system.
18.  Groundedness is defined as the factual accuracy and reliability of the
    generated answer as the answer is grounded in the retrieved context.
19.  Provide both a Reasoning and a Score.
20.   Reasoning: step-by-step explaining how faithful the generated answer is to the
    retrieved context.
21.   Score as a string between '0' and '3':
22.    "0: Not grounded - The answer is completely unrelated or not based on the
    context."
23.    "1: Low groundedness - The answer has minimal grounding on the context."
24.    "2: Medium groundedness - The answer is somewhat grounded on the context."
25.    "3: High groundedness - The answer is highly grounded on the context"
26.
27. prompt_instruction_ar: >
28.  You are a judge for evaluating a RAG system.
29.  Answer Relevance is defined the helpfulness and on-point nature of the answer,
    aligning with the user's intent and providing valuable insights.
30.  Provide both a Reasoning and a Score.
31.   Reasoning: step-by-step explaining how well the generated answer addresses the
    user's original query.
32.   Score as a string between '0' and '3':
33.    "0: No relevance - The generated answer is completely unrelated to the
    query."
34.    "1: Low relevance - The generated answer has minimal relevance to the query."
35.    "2: Medium relevance - The generated answer is somewhat relevant to the
    query."
36.    "3: High relevance - The generated answer is highly relevant to the query."
37.
38. prompt_instruction_cr: >
39.  You are a judge for evaluating a RAG system.
40.  Context Relevance is defined as the relevance of the retrieved context to the
    query's intent and the appropriateness of the retrieved context in providing a
    coherent and useful response.
41.  Provide both a Reasoning and a Score.
```

```
  Reasoning: step-by-step explaining how the retrieved context aligns with the user's query.
  Score as a string between '0' and '3':
   "0: No relevance - The retrieved context is completely unrelated to the query."
   "1: Low relevance - The retrieved context has minimal relevance to the query."
   "2: Medium relevance - The retrieved context is somewhat relevant to the query."
   "3: High relevance - The retrieved context is highly relevant to the query."

prompt_context_and_query: >
  ---KONTEXTI---:\n{context}\n\n---KYSYMYS---:\n{query}

prompt_context_query_answer_ai_judge: >
  ---RETRIEVED CONTEXT---:\n{context}\n\n---QUERY---:\n{query}\n\n---GENERATED ANSWER---:\n{rag_answer}

source_dir: "data/source"
txt_raw_dir: "data/txt_raw"
txt_processed_dir: "data/txt_processed"
chunk_dir: "data/chunk"
vector_store_dir: "data/vector_store"
evaluation_dir: "data/evaluation"

base_url: "http://localhost:11434" # Base URL for the Ollama server

llm:
  model_name_rag: "gemma3:1b-it-qat" # "gemma3:4-it-qat" #
  model_name_judge: "mistral:7b-instruct-v0.3-q4_1"
  request_timeout: 300.0
  model_kwargs:
    stream: false
    num_ctx_rag: 4096
    num_ctx_judge: 32768
    max_new_tokens_rag: 512
    max_new_tokens_judge: 1024
    temperature_rag: 0.1
    temperature_judge: 0
    top_p: 0.95
    top_k: 50
    repetition_penalty: 1.1
    num_batch: 1
    parallel: false

embedding:
  model_name: "snowflake-arctic-embed2:latest" # "bge-m3:latest#
  model_kwargs:
    pooling: "mean"
    num_ctx_build: 2045
    num_ctx_query: 512
    embed_batch_size: 8
    # normalize_embeddings: true
    # precision: "fp16"
```

Program code 4. Importing libraries and setting configuration

```
import os
os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"
from IPython.display import display, Markdown
display(Markdown("⚠️ **Restart kernel after environment changes**"))
from llama_index.llms.ollama import Ollama
from llama_index.embeddings.ollama import OllamaEmbedding
from llama_index.vector_stores.faiss import FaissVectorStore
from llama_index.core import (
    Document,
    VectorStoreIndex,
    Settings,
    StorageContext,
    load_index_from_storage,
)
import faiss # 'conda install pytorch::faiss-cpu' for vector search
from pathlib import Path
from typing import List, Union
import pandas as pd
import re
import logging
import yaml
import json
import faiss
import requests
import time
from types import SimpleNamespace

import torch
import win32com.client # for docx
import win32clipboard # for docx copying to txt
import fitz # 'pip install PyMuPDF' for reading PDFs
from collections import namedtuple

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s] %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S',
    force=True
)
#
class ProjectYmlConfigLoader:
    def __init__(self, marker: str = ".gitignore", config_subpath: str =
"config/config.yml"):
        self.marker = marker
        self.config_subpath = config_subpath
        self.working_dir = Path(__file__).resolve().parent if '__file__' in
globals() else Path.cwd()
        self._project_root = None
        self._config_dict = None

    def _find_project_root(self, current_path: Path, marker: str) -> Path:
        current_path = current_path.resolve()
        for parent in [current_path] + list(current_path.parents):
            if (parent / marker).exists():
                return parent
```

```
54.         raise FileNotFoundError(f"💔 Could not find project root with marker
    '{marker}'.")
55.
56.     def _load_config_dict(self):
57.         config_path = self.project_root / self.config_subpath
58.         if not config_path.exists():
59.             raise FileNotFoundError(f"❌ Config file not found at {config_path}")
60.         with open(config_path, "r", encoding="utf-8") as f:
61.             config_dict = yaml.safe_load(f)
62.         logging.info(f"✅ Config loaded from
    {config_path.relative_to(self.project_root)}")
63.         logging.info(f"✅ Current working directory:
    {self.working_dir.relative_to(self.project_root)}")
64.         return config_dict
65.
66.     @property
67.     def project_root(self):
68.         if self._project_root is None:
69.             self._project_root = self._find_project_root(self.working_dir,
    self.marker)
70.         return self._project_root
71.
72.     @property
73.     def config_dict(self):
74.         if self._config_dict is None:
75.             self._config_dict = self._load_config_dict()
76.         return self._config_dict
77. #
78. class ConfigNamespaceBuilder:
79.     def __init__(self, project_root: Path):
80.         self.project_root = project_root
81.         self.config_namespace = SimpleNamespace()
82.
83.     def _is_url(self, value):
84.         return isinstance(value, str) and re.match(r'^https?://', value)
85.
86.     def build_config_namespace(self, config_dict, parent_key=''):
87.         for key, value in config_dict.items():
88.             var_name = f"{parent_key}_{key}" if parent_key else key
89.             var_name = var_name.upper()
90.
91.             if isinstance(value, dict):
92.                 self.build_config_namespace(value, var_name)
93.             elif isinstance(value, str):
94.                 if self._is_url(value):
95.                     setattr(self.config_namespace, var_name, value)
96.                     logging.info(f"[URL] {var_name}: {value}")
97.                 elif key.endswith('_dir') or key.endswith('_path'):
98.                     abs_path = (self.project_root / value).resolve()
99.                     setattr(self.config_namespace, var_name, abs_path)
100.                    rel_path = abs_path.relative_to(self.project_root)
101.                    logging.info(f"[REL] {var_name}: {rel_path}")
102.                else:
103.                    setattr(self.config_namespace, var_name, value)
104.                    logging.info(f"[STR] {var_name}: {value}")
105.            else:
106.                setattr(self.config_namespace, var_name, value)
107.                logging.info(f"[VAL] {var_name}: {value}")
```

```
108.
109.         return self.config_namespace
110.#
111.class ConfigManager:
112.    def __init__(self, marker=".gitignore", config_subpath="config/config.yml"):
113.        self.loader = ProjectYmlConfigLoader(marker, config_subpath)
114.        self.builder = ConfigNamespaceBuilder(self.loader.project_root)
115.        self._config = None
116.
117.    @property
118.    def config(self):
119.        if self._config is None:
120.            self._config =
    self.builder.build_config_namespace(self.loader.config_dict)
121.        return self._config
122.
123.project_root = ConfigManager().loader.project_root
124.if project_root: logging.info(f"✅ Project root: {project_root}")
125.config = ConfigManager().config
```

Program code 5. PDF to TXT converter

```
1.  def convert_pdfs_to_text_with_page_separation(pdf_dir: Path, text_dir: Path) ->
    List[str]:
2.      pdf_dir = Path(pdf_dir)
3.      text_dir = Path(text_dir)
4.
5.      converted_files = []
6.      for file_path in pdf_dir.iterdir():
7.          if file_path.suffix.lower() == ".pdf" and file_path.is_file():
8.              text_file_name = file_path.stem + ".txt"
9.              text_path = text_dir / text_file_name
10.             try:
11.                 with fitz.open(file_path) as doc:
12.                     text = ""
13.                     for i, page in enumerate(doc, start=1):
14.                         page_text = page.get_text()
15.                         text += f"\n=== Page {i} ===\n{page_text}"
16.                 with open(text_path, "w", encoding="utf-8") as text_file:
17.                     text_file.write(text)
18.                 logging.info(f"Converted [{file_path.name}] to
    [{text_file_name}]")
19.                 converted_files.append(text_file_name)
20.             except Exception as e:
21.                 logging.error(f"Failed to convert {file_path.name}: {e}")
22.     if converted_files:
23.         logging.info(f"PDF conversion complete: {len(converted_files)} file(s)
    converted.")
24.     else:
25.         logging.info("PDF conversion complete: No PDF files were converted.")
26.     return converted_files
27. source_dir = Path(config.SOURCE_DIR)
28. output_dir = Path(config.TXT_RAW_DIR)
29. output_dir.mkdir(parents=True, exist_ok=True)
30.
31. convert_pdfs_to_text_with_page_separation(source_dir, output_dir)
```

Program code 6. DOCX to TXT converter

```
word = win32com.client.Dispatch("Word.Application")
word.Visible = False  # Run Word in background

def get_clipboard_text(retries=5, delay=0.3) -> str:
    for attempt in range(retries):
        try:
            win32clipboard.OpenClipboard()
            data = win32clipboard.GetClipboardData()
            win32clipboard.CloseClipboard()
            return data
        except Exception as e:
            if attempt == retries - 1:
                raise e
            time.sleep(delay)
    return ""


def convert_docxs_to_text(docx_dir: Path, text_dir: Path) -> List[str]:
    docx_dir = Path(docx_dir)
    text_dir = Path(text_dir)
    text_dir.mkdir(parents=True, exist_ok=True)

    converted_files = []

    for file_path in docx_dir.iterdir():
        if file_path.name.startswith('~$'):
            continue  # Skip Word temp files

        if file_path.suffix.lower() == ".docx" and file_path.is_file():
            text_file_name = file_path.stem + ".txt"
            text_path = text_dir / text_file_name

            doc = None
            try:
                logging.info(f"Opening file: {file_path.name}")
                doc = word.Documents.Open(str(file_path), ReadOnly=True)

                # Use clipboard to preserve formatting (numbering, bullets, etc.)
                doc.Content.Copy()
                time.sleep(0.3)  # Ensure clipboard is ready
                text_data = get_clipboard_text()

                with open(text_path, "w", encoding="utf-8") as f:
                    f.write(text_data)

                logging.info(f"Converted [{file_path.name}] → [{text_file_name}]")
                converted_files.append(text_file_name)

            except Exception as e:
                logging.error(f"Failed to convert {file_path.name}: {e}")
            finally:
                try:
                    if doc is not None:
                        doc.Close(False)
                except Exception as e:
                    logging.warning(f"Failed to close document {file_path.name}:
{e}")
```

```
57.
58.     if converted_files:
59.         logging.info(f"✅ Docx conversion complete: {len(converted_files)} file(s)
    converted.")
60.     else:
61.         logging.warning("⚠️ Docx conversion: No docx files were converted.")
62.
63.     return converted_files
64.
65.
66. # Run the conversion function if this script is executed directly
67. source_dir = Path(config.SOURCE_DIR)
68. output_dir = Path(config.TXT_RAW_DIR)
69. output_dir.mkdir(parents=True, exist_ok=True)
70.
71. try:
72.     converted_files = convert_docxs_to_text(source_dir, output_dir)
73. finally:
74.     word.Quit()
```

Program code 7. Blank lines remover

```
1.  def remove_blank_lines_from_file(file_path: Union[str, Path])-> None:
2.      lines = Path(file_path).read_text(encoding="utf-8").splitlines()
3.      logging.info(f"[{file_path.name}]: [{len(lines)} lines] before")
4.
5.      filtered = [line for line in lines if line.strip()]
6.
7.      num_blank_lines_removed = len(lines) - len(filtered)
8.      logging.info(f"[{file_path.name}]: [-{num_blank_lines_removed} blank lines]")
9.
10.     Path(file_path).write_text('\n'.join(filtered), encoding="utf-8")
11.     lines = Path(file_path).read_text(encoding="utf-8").splitlines()
12.     logging.info(f"[{file_path.name}]: [{len(lines)} lines] after")
13.
14. def process_all_txt_files_in_dir(directory: Union[str, Path])-> None:
15.     for file in Path(directory).glob("*.txt"):
16.         remove_blank_lines_from_file(file)
17.
18. process_all_txt_files_in_dir(config.TXT_RAW_DIR)
```

Program code 8. Page separator, header, and footer remover

```
1.  def read_lines(input_path: Union[str, Path]) -> list:
2.      with open(input_path, "r", encoding="utf-8") as infile:
3.          return infile.readlines()
4.
5.  def skip_blocks_around_separator(lines: list,
6.                                   separator: str,
7.                                   num_lines_to_skip_above: int,
8.                                   num_lines_to_skip_below: int
9.                                   ):
10.     output_lines = []
11.     i = 0
12.     total = len(lines)
13.     while i < total:
```

```
        if lines[i].startswith(separator):
            skip_start_index = max(0, len(output_lines) - num_lines_to_skip_above)
            output_lines = output_lines[:skip_start_index]
            skip_end_index = min(total, i + num_lines_to_skip_below + 1)  # +1 to
include separator line
            i = skip_end_index
        else:
            output_lines.append(lines[i])
            i += 1
    return output_lines

def write_lines(output_path: Union[str, Path],
                lines: list):
    with open(output_path, "w", encoding="utf-8") as outfile:
        outfile.writelines(lines)

def get_output_path(input_path: Union[str, Path],
                    output_dir: Union[str, Path],
                    suffix="_processed.txt"):
    input_path = Path(input_path)
    return Path(output_dir) / (input_path.stem + suffix)

def clean_file_with_separator(input_path: Path,
                              output_dir: Path,
                              n_above: int,
                              m_below: int,
                              separator):
    lines = read_lines(input_path)
    cleaned_lines = skip_blocks_around_separator(lines, separator, n_above,
m_below)
    output_path = get_output_path(input_path, output_dir)
    write_lines(output_path, cleaned_lines)
    logging.info(f"Output written to: {output_path.relative_to(project_root)}")

# Process the text files with the specified separator and CUSTOMIZED n_above and
m_below values
output_dir = Path(config.TXT_PROCESSED_DIR)
output_dir.mkdir(parents=True, exist_ok=True)

separator = "=== Page"

input_file_1 = Path(config.TXT_RAW_DIR) / "raw_text.txt"
clean_file_with_separator(input_path = input_file_1, output_dir = output_dir,
n_above = 0, m_below = 0, separator = separator)
```

Program code 9. Chunker with regular expressions

```
Section = namedtuple("Section", ["chunk_id", "heading", "title", "body"])

def chunk_document_by_section(text: str) -> List[Section]:
    pattern = re.compile(r'^(\d+\.\d+(?:\.\d+)*|\d+\.)\s+([^\n]+)', re.MULTILINE)

    matches = list(pattern.finditer(text))

    sections = []
    for i, match in enumerate(matches):
        chunk_id = i + 1
```

```
        heading = match.group(1)
        title = match.group(2).strip()
        start_idx = match.end()
        end_idx = matches[i + 1].start() if i + 1 < len(matches) else len(text)
        body = text[start_idx:end_idx].strip()
        if body == "":
            body = title

        sections.append(Section(chunk_id, heading, title, body))
    return sections
#
def load_and_chunk_documents(directory_path: Path):
    documents = []
    if not directory_path.exists():
        raise FileNotFoundError(f"📂 Directory not found: {directory_path}")

    for text_file in directory_path.glob("*.txt"):
        try:
            with text_file.open('r', encoding='utf-8') as infile:
                file_content = infile.read()
                chunks = chunk_document_by_section(file_content)
                for chunk in chunks:
                    documents.append(
                        Document(
                        #    text=chunk.body,
                        # Modify to include title and body in the text field
                            text=f"{chunk.title}\n{chunk.body}",
                            metadata={'chunk_id':chunk.chunk_id, 'heading':
chunk.heading, 'title': chunk.title}
                        )
                    )
            logging.info(f"Loading and chunking: {text_file.name}")
        except Exception as e:
            logging.error(f"❌ Error reading {text_file.name}: {str(e)}")

    return documents

input_dir = Path(config.TXT_PROCESSED_DIR)

chunked_documents = load_and_chunk_documents(input_dir)
logging.info(f"Total number of chunked documents: {len(chunked_documents)}")
```

Program code 10. Functions to save chunks to TXT and JSON for review

```
def save_documents_txt(
        documents: List[Document],
        output_txt_file: Union[str, Path]
        ):
    with open(output_txt_file, 'w', encoding='utf-8') as file:
        for idx, doc in enumerate(documents, start=1):
            file.write(f"chunk_id: {doc.metadata.get('chunk_id', '(no chunk
id)')}\n")
            file.write(f"Heading: {doc.metadata.get('heading', '(no
heading)')}\n")
            file.write(f"Title: {doc.metadata.get('title', '(no title)')}\n")
            file.write(f"Text: {doc.text}\n")
            file.write("-" * 50 + "\n")
```

```
12.
13. def save_documents_jsonl(
14.         documents: List[Document],
15.         output_jsonl_file: Union[str, Path]):
16.     with open(output_jsonl_file, 'w', encoding='utf-8') as file:
17.         for doc in documents:
18.             json_line = json.dumps({"text": doc.text, "metadata": doc.metadata},
    ensure_ascii=False)
19.             file.write(json_line + '\n')
20.
21. document_path = Path(config.TXT_PROCESSED_DIR)
22. documents = chunked_documents
23.
24. output_dir = Path(config.CHUNK_DIR)
25. output_dir.mkdir(parents=True, exist_ok=True)
26.
27. output_txt_file = Path(output_dir) / "chunked_documents.txt"
28. output_jsonl_file = Path(output_dir) / "chunked_documents.jsonl"
29.
30. save_documents_txt(documents, output_txt_file)
31. logging.info(f"Chunked documents saved to:
    {output_txt_file.relative_to(project_root)}")
32. save_documents_jsonl(documents, output_jsonl_file)
33. logging.info(f"Chunked documents saved to:
    {output_jsonl_file.relative_to(project_root)}")
34.
35. # Verify the JSONL file and get stats, especially the max num of word_count to set
    the num_ctx in the config
36. df = pd.read_json(output_jsonl_file, lines=True, encoding='utf-8')
37. df["word_count"] = df["text"].str.split().str.len()
38. df["char_count"] = df["text"].str.len()
39.
40. # Get stats and convert to integers
41. stats = df[["word_count", "char_count"]].describe().round(0).astype(int)
42. print(stats)
43.
44. # Extract chunk_id from nested metadata
45. df["chunk_id"] = df["metadata"].apply(lambda x: x.get("chunk_id", "unknown"))
46.
47. # Show top chunks by word count
48. top_chunks = df[["chunk_id", "word_count"]].sort_values(by="word_count",
    ascending=False).head(15)
49. print(top_chunks)
```

Program code 11. Embedding vector dimension computing

```
1.  def compute_embedding_dimension(
2.      embed_model,
3.      test_input: str = "any text"
4.  ) -> int:
5.      # Compute dimension
6.      try:
7.          actual_dimension = len(embed_model.get_text_embedding(test_input))
8.          logging.info(f"🔢 Computed embedding dimension: {actual_dimension} for
    model: {getattr(embed_model, 'model_name', 'unknown')}")
9.          return actual_dimension
10.     except Exception as e:
```

```
        logging.error(f"❌ Failed to compute embedding dimension: {e}")
        raise)
```

Program code 12. Functions for embedding and saving FAISS index to vector store

```
def create_faiss_index(dimension: int):
    """Create a FAISS IndexFlatL2 index with the given dimension."""
    faiss_index = faiss.IndexFlatL2(dimension)
    logging.info(f"🧩 FAISS index created with dimension: {dimension}")
    return faiss_index

def save_faiss_index(faiss_index, vector_store_dir: Path):
    """Save the FAISS index to a binary file."""
    file_name = "vectors.bin"
    index_path = vector_store_dir / str(file_name)
    faiss.write_index(faiss_index, str(index_path))
    logging.info(f"✅ FAISS index vector database saved to [{vector_store_dir.relative_to(project_root)}]\[{file_name}]. Ready for retrieval!")

def build_and_save_index(
    documents: list[Document],
    vector_store_dir: Path
):
    """Build and persist the vector index using the embedding dimension from cache."""
    # Validate documents before proceeding
    if not documents:
        raise ValueError("⚠️ No documents available for embedding.")

    # Ensure the directory exists for saving the index (create it if it doesn't exist)
    vector_store_dir = Path(vector_store_dir)
    vector_store_dir.mkdir(parents=True, exist_ok=True)

    # Save embedding model metadata
    metadata = {
        "embedding_model": config.EMBEDDING_MODEL_NAME,
        "embedding_model_kwargs": {
            "num_ctx": int(config.EMBEDDING_MODEL_KWARGS_NUM_CTX_BUILD),
            "embed_batch_size": int(config.EMBEDDING_MODEL_KWARGS_EMBED_BATCH_SIZE)
        }
    }
    with open(vector_store_dir / "_model_info.json", "w") as f:
        json.dump(metadata, f, indent=2)

    # Set up embedding model
    embed_model = OllamaEmbedding(
        model_name=config.EMBEDDING_MODEL_NAME,
        base_url=config.BASE_URL,
        ollama_additional_kwargs={
            "num_ctx": int(config.EMBEDDING_MODEL_KWARGS_NUM_CTX_BUILD),
            "embed_batch_size": int(config.EMBEDDING_MODEL_KWARGS_EMBED_BATCH_SIZE)
            }
```

```
    )

    # Compute embedding dimension
    embedding_dimension = compute_embedding_dimension(embed_model)

    # Create FAISS index with the cached dimension
    faiss_index = create_faiss_index(embedding_dimension)

    # Wrap the FAISS index in a LlamaIndex FAISS vector store
    vector_store = FaissVectorStore(faiss_index=faiss_index)

    # Force a fresh start for StorageContext
    storage_context = StorageContext.from_defaults(vector_store=vector_store)

    # Build and persist the index
    logging.info("🧠 Building and saving index with FAISS vector store...")
    index = VectorStoreIndex.from_documents(documents, embed_model=embed_model,
storage_context=storage_context)
    index.storage_context.persist(str(vector_store_dir))

    # Save FAISS index
    save_faiss_index(faiss_index, vector_store_dir)
```

Program code 13. Local Ollama server status checker

```
def check_ollama_server():
    url = f"{config.BASE_URL}/api/tags"
    while True:
        try:
            ollama_response = requests.get(url, timeout=3)
            if ollama_response.status_code == 200:
                logging.info(f"✅ Ollama status: {ollama_response.status_code}
(running)")
                return  # Ready to continue
            else:
                logging.info(f"👉 Ollama responded with status:
{ollama_response.status_code}, waiting...")
        except requests.exceptions.ConnectionError:
            logging.info("👉 Ollama is not running. Please start Ollama to
continue...")
        except requests.exceptions.RequestException as e:
            logging.info(f"💔 Error contacting Ollama: {e}")
        time.sleep(10)  # Wait before retrying

```

Program code 14. Main function to build vector store

```
def main():

    # Disable unnecessary logging for cleaner output
    logging.getLogger("llama_index").setLevel(logging.WARNING)
    logging.getLogger("httpx").setLevel(logging.WARNING)
    logging.getLogger("requests").setLevel(logging.WARNING)

    # Check Ollama server status
    check_ollama_server()
    documents = chunked_documents
    vector_store_dir = Path(config.VECTOR_STORE_DIR)
    vector_store_dir.mkdir(parents=True, exist_ok=True)
    build_and_save_index(
        documents=documents,
        vector_store_dir=vector_store_dir
    )
```

Program code 15. Embedding and large language models setter

```
def set_models():

    Settings.embed_model = OllamaEmbedding(
        model_name=config.EMBEDDING_MODEL_NAME,
        base_url=config.BASE_URL,
        model_kwargs={
            "num_ctx": int(config.EMBEDDING_MODEL_KWARGS_NUM_CTX_QUERY),
            "pooling": str(config.EMBEDDING_MODEL_KWARGS_POOLING)
        }
    )

    Settings.llm = Ollama(
        base_url=config.BASE_URL,
        model=config.LLM_MODEL_NAME_RAG,
        request_timeout=300.0,  #  minute(s) timeout
        model_kwargs={
            "num_ctx": int(config.LLM_MODEL_KWARGS_NUM_CTX_RAG),
            "max_new_tokens": int(config.LLM_MODEL_KWARGS_MAX_NEW_TOKENS_RAG),
            "temperature": float(config.LLM_MODEL_KWARGS_TEMPERATURE_RAG),
        }
    ))
```

Program code 16. Vector store loader

```
# Load the vector store (FAISS index) from storage to use in the RAG app
def load_vector_store(
    vector_store_path: Path):
    try:
        if not vector_store_path.exists():
            raise FileNotFoundError(f"❌ FAISS index file not found at:
{vector_store_path}")

        # Load FAISS index
        faiss_index = faiss.read_index(str(vector_store_path))
```

```
        logging.info(f"FAISS index class: {type(faiss_index)}")
        vector_store = FaissVectorStore(faiss_index=faiss_index)

        # Use parent directory of index file as vector_store_dir
        vector_store_dir = vector_store_path.parent
        storage_context = StorageContext.from_defaults(
            persist_dir=str(vector_store_dir),
            vector_store=vector_store
        )
        return load_index_from_storage(storage_context)
    except Exception as e:
        logging.info(f"❌ Error loading FAISS index: {e}")
        return None
```

Program code 17. Checking the embedding model and dimension matches

```
def check_embedding_and_dimension_match(index) -> bool:
    if index is None:
        logging.info("⚠️ FAISS index could not be loaded. Exiting...")
        return False

    meta_path = Path(config.VECTOR_STORE_DIR) / "embedding_metadata.json"
    if not meta_path.exists():
        logging.info("❌ Missing embedding metadata file.")
        return False

    with open(meta_path) as f:
        meta = json.load(f)

    embedding_model_vector_store = meta.get("embedding_model")
    if embedding_model_vector_store != config.EMBEDDING_MODEL_NAME:
        logging.info(f"❌ Embedding model mismatch: FAISS index
({embedding_model_vector_store}) vs Config ({config.EMBEDDING_MODEL_NAME})")
        return False
    logging.info(f"✅ Embedding model matched: {embedding_model_vector_store}")

    # Compute query embedding dimension
    try:
        test_embedding = Settings.embed_model.get_text_embedding("embed this text,
please")
        query_dim = len(test_embedding)
        logging.info(f"Dimension: {query_dim} (query embedding)")
    except Exception as e:
        logging.info(f"❌ Failed to compute test embedding: {e}")
        return False

    # FAISS index dimension
    faiss_index_dim = index._vector_store._faiss_index.d
    logging.info(f"Dimension: {faiss_index_dim} (FAISS index vector store)")

    if faiss_index_dim != query_dim:
        logging.info(f"❌ Embedding dimension mismatch: FAISS index
({faiss_index_dim}) vs Query embedding ({query_dim})")
        return False
    else:
```

```
        logging.info(f"✅ Embedding dimension match: {faiss_index_dim} (FAISS index) == {query_dim} (query embedding)")

    logging.info(f"✅ Embedding dimensions matched")
    return True
```

Program code 18. Query engine builder

```
def build_query_engine(index):
    query_engine = index.as_query_engine(
        similarity_top_k=config.RETRIEVAL_TOP_K,
        response_mode="compact"
    )
    return query_engine
```

Program code 19. Chunk Retriever and displayer class

```
class ChunkRetriever:
    def __init__(self, query, query_engine):
        self.query = query
        self.query_engine = query_engine

    @staticmethod
    def extract_body(text: str) -> str:
        return text.split("\n", 1)[1].strip() if "\n" in text else ""

    def retrieve_chunks(self, retrieve_limit: int = 5):
        retriever = self.query_engine._retriever
        retrieved_chunks = retriever.retrieve(str(self.query))

        if not retrieved_chunks:
            logging.info("⚠️ No relevant chunks found!")
            return []

        return retrieved_chunks[:retrieve_limit]

    def display_chunks(self, retrieved_chunks, display_limit, body_preview_length: int):
        print(f"Kysymyksesi (Your question):\n\"\n{self.query}\n\"", flush=True)
        print(f"{len(retrieved_chunks)} asiakirjaa haettu (documents retrieved). Näyttöraja (Diplaying limit): {display_limit}: \n", flush=True)

        for i, chunk in enumerate(retrieved_chunks[:display_limit]):
            body_preview_length = int(body_preview_length)
            chunk_id = chunk.metadata.get("chunk_id", f"Document {i+1} has no Chunk ID")
            heading = chunk.metadata.get("heading", f"Document {i+1} has no Heading")
            title = chunk.metadata.get("title", f"Document {i+1} has no Title")
            body_text = self.extract_body(chunk.text)
            content = f"{body_text[:body_preview_length]}" if body_text else ""
            score = getattr(chunk, "score", None)
            print(
                f"📌 {i+1} | ID: {chunk_id} | Pisteet (Score) = {score:.2f} | Osio (Heading): {heading}\n"
```

```
                f"Otsikko (Title): {title}\nSisältö (Content): {content} ...\n",
    flush=True
            )
```

Program code 20. Prompt customization function

```
# Customize the prompt based on the prompt guidance LLM model, with specific
instructions for RAG
def customize_prompt(query: str, context: str, prompt_instruction) -> str:

    model_name = config.LLM_MODEL_NAME_RAG.lower()

    # Set the instruction based on the model name
    instruction =
prompt_instruction.format(prompt_control_command=config.PROMPT_CONTROL_COMMAND)

    # Context and query string
    context_and_query_string = config.PROMPT_CONTEXT_AND_QUERY
    context_and_query = context_and_query_string.format(query=query,
context=context)

    # Model-specific prompt templates
    if "llama2" in model_name:
        return (f"<s>[INST] <<SYS>>
{instruction}\n<</SYS>>\n{context_and_query}[/INST]")
    elif "llama3" in model_name:
        return
(f"<|begin_of_text|><|start_header_id|>system<|end_header_id|>\n{instruction}\n"

f"<|eot_id|><|start_header_id|>user<|end_header_id|>\n{context_and_query}"
                f"<|eot_id|><|start_header_id|>assistant<|end_header_id|>")
    elif "gemma3" in model_name:
        return
(f"<start_of_turn>user\n{instruction}\n{context_and_query}\n<end_of_turn>\n<start_o
f_turn>model")
    elif "mistral" in model_name:
        return (f"<s>[INST] {instruction}\n{context_and_query}[/INST]")
    else:
        return (f"{instruction}\n{context_and_query}\n")
```

Program code 21. Building the context from retrieved chunks

```
def build_context(retrieved_chunks):
    if not retrieved_chunks:
        return "No relevant documents found for this query."
    context = "\n\n".join(f"[Osio {chunk.metadata['heading']}] {chunk.text}" for
chunk in retrieved_chunks)
    return context

```

Program code 22. Generating the LLM response

```
def generate_response(query, context, prompt_instruction):
    custom_prompt = customize_prompt(context, query, prompt_instruction)

```

```
    try:
        response = Settings.llm.complete(custom_prompt)
        return str(response)

    except Exception as e:
        logging.info(f"💔 Error generating response: {e}")
        return "❌ An error occurred while processing the query."
```

Program code 23. Main RAG function

```
def main():
    # Configure logging
    logging.basicConfig(level=logging.INFO)

    # Inform user about the project root
    project_root = ConfigManager().loader.project_root
    if project_root: logging.info(f"✅ Project root: {project_root}")

    # Build config from config.yml
    config = ConfigManager().config
    if config: logging.info("✅ Config globals set.")

    # Check Ollama server status
    check_ollama_server()
    logging.info("ℹ️ Ollama automatically uses the GPU if available. Ollama käyttää automaattisesti GPU:ta, jos se on saatavilla.")

    # Set embedding and large language models
    set_models()
    logging.info("✅ Models set.")

    # Load vector store (FAISS index)
    vector_store_path = Path(config.VECTOR_STORE_DIR).resolve() / "vectors.bin"
    index = load_vector_store(Path(vector_store_path))
    if index: logging.info(f"✅ Vector store (FAISS index) loaded from {vector_store_path.relative_to(project_root)}")

    # Check embedding and dimension match
    if not check_embedding_and_dimension_match(index):
        logging.info("❌ Incompatible vector store. Exiting main().")
        return

    # Build query engine
    query_engine = build_query_engine(index)
    if query_engine: logging.info("✅ Query engine built.\n✅✅✅ System ready. Kyselyä odotellessa... (Waiting for question...)")

    # Main loop for user interaction
    while True:
        try:
            user_query = input(f"💬 Kirjoita kyselysi. Lopeta kirjoittamalla '{config.EXIT_COMMAND}' (Enter your query. To quit, type '{config.EXIT_COMMAND}')\n>>> ").strip()
            if not user_query:
                # Check if the input is empty or contains only whitespace
```

```
                print(f"⚠️ Anna kelvollinen kysely. Lopeta kirjoittamalla '{config.EXIT_COMMAND}' (Please enter a valid query. To quit, type: '{config.EXIT_COMMAND}')", flush=True)
                continue
            if user_query.lower() == config.EXIT_COMMAND:
                print(f"👋 Kirjoitit (You typed) '{config.EXIT_COMMAND}'. Toivomme, että RAG-chat oli hyödyllinen sinulle. Nähdään pian. (We hope that you found the RAG chat useful. See you soon.)", flush=True)
                break
        except KeyboardInterrupt:
            logging.info("🛑 [Session interrupted by user. Exiting...] Käyttäjä keskeytti istunnon. Poistutaan...")
            break

        logging.info(f"Kysely vastaanotettu. (Query received)")
        # Take the user valid query
        query = user_query
        # Define the retriever
        retriever = ChunkRetriever(query, query_engine)
        # Retrieve the top K chunks, as a list of Document objects
        retrieved_chunks = retriever.retrieve_chunks(int(config.RETRIEVAL_TOP_K))

        # Context display option 1, intended for developer to view details including the score
        retriever.display_chunks(retrieved_chunks, config.DISPLAY_LIMIT, config.BODY_PREVIEW_LENGTH)
        # Build the context from the retrieved chunks for generating the response, and optionally display it

        print(f"Vastauksen luominen (Generating the answer)...\n", flush=True)

        context = build_context(retrieved_chunks)
        # Generate the response
        response = generate_response(query, context, config.PROMPT_INSTRUCTION)

        mahdoton_variants = [config.PROMPT_CONTROL_COMMAND, "khong_the_tra_loi"]
        normalized_response = response.strip().lower()
        if any(mahdoton_variant in normalized_response for mahdoton_variant in mahdoton_variants):
            # Found a variant of 'mahdo_ton'
            response = str(config.REPHRASE_MESSAGE)

        # Print the response
        print(f"\n🤖 Tekoälyn vastaus (AI's answer):\n\n{response}\n", flush=True)

        # Print the disclaimer and warning
        print(f"ℹ️ VASTUUVAPAUSLAUSEKE: Tämä vastaus on tekoälyn luoma, ja se voi olla väärä.\n (DISCLAIMER: This response is generated by AI, and can be wrong.)\n"
              f"⚠️ VAROITUS: Jotkin toiminnot voivat aiheuttaa tietojen menetyksenb, esim. DELETE, POST, UPDATE.\n (WARNING: Some operations may cause data loss, e.g. DELETE, POST, UPDATE).\n"
              f"👉 Lue aina oikeat ohjeet virallisesta dokumentaatiosta.\n(Always read the official documentation for the correct instructions.)\n\n"
              f"Vastauksen loppu (End of the answer).\n\nUutta kyselyä odotellessa (Waiting for new query)...\n", flush=True)
```

Program code 24. Main function adaptation for the batch inference module

```
    with open("../data/evaluation/60_questions.json", "r", encoding="utf-8") as f:
        data = json.load(f)

    for idx, item in enumerate(data):
        query = item.get("fi", "")
        start_time = time.time()

        retriever = ChunkRetriever(query, query_engine)
        retrieved_chunks = retriever.retrieve_chunks(int(config.RETRIEVAL_TOP_K))
        retrieval_time = time.time() - start_time

        context = build_context(retrieved_chunks)
        response = generate_response(query, context, config.PROMPT_INSTRUCTION)

        mahdoton_variants = [config.PROMPT_CONTROL_COMMAND, "En osaa vastata."]
        normalized_response = response.strip().lower()
        if any(mahdoton_variant in normalized_response for mahdoton_variant in
mahdoton_variants):
            response = str(config.REPHRASE_MESSAGE)

        rag_time_including_retrieval_and_generation = time.time() - start_time
        generation_time_alone = rag_time_including_retrieval_and_generation -
retrieval_time

        provided_heading = item.get("dokumentaatio", "").strip().split(" ")[0]
        item["heading"] = provided_heading
        item["retrieved_headings"] = [chunk.metadata.get("heading") for chunk in
retrieved_chunks]
        item["rag_answer"] = response
        item["time_R"] = retrieval_time
        item["time_G"] = generation_time_alone
        item["context"] = context

        with open("../data/evaluation/batch_output/yymmdd_batch_rag_inference.json",
"w", encoding="utf-8") as f:
            json.dump(data, f, ensure_ascii=False, indent=2)

        logging.info(f"Processed item {idx+1}/{len(data)} and saved progress.")

    print("Batch output saved!")
```

Program code 25. Modified model setter for the AI judge

```
# Set up the LLM and embed models
def set_llm_model_judge():
    Settings.llm = Ollama(
        base_url=config.BASE_URL,
        model=config.LLM_MODEL_NAME_JUDGE,
        request_timeout=300.0,  #  minute(s) timeout
        model_kwargs={
            "num_ctx": int(config.LLM_MODEL_KWARGS_NUM_CTX_JUDGE),
            "max_new_tokens": int(config.LLM_MODEL_KWARGS_MAX_NEW_TOKENS_JUDGE),
            "temperature": float(config.LLM_MODEL_KWARGS_TEMPERATURE_JUDGE),
        }
    )
```

Program code 26. Modified prompt customizing function for the AI judge

```
# Customize the prompt based on the prompt guidance LLM model, with specific instructions for RAG
def customize_prompt_judge(query: str, context: str, rag_answer: str, prompt_instruction) -> str:

    model_name = config.LLM_MODEL_NAME_JUDGE.lower()

    instruction = prompt_instruction.format(prompt_control_command=config.PROMPT_CONTROL_COMMAND)

    # Context and query string
    context_and_query_string = config.PROMPT_CONTEXT_QUERY_ANSWER_AI_JUDGE
    context_and_query = context_and_query_string.format(query=query, context=context, rag_answer=rag_answer)

    # Model-specific prompt templates
    if "llama2" in model_name:
        return (f"<s>[INST] <<SYS>>{instruction}\n<</SYS>>\n{context_and_query}[/INST]")
    elif "llama3" in model_name:
        return (f"<|begin_of_text|><|start_header_id|>system<|end_header_id|>\n{instruction}\n"
                f"<|eot_id|><|start_header_id|>user<|end_header_id|>\n{context_and_query}"
                f"<|eot_id|><|start_header_id|>assistant<|end_header_id|>")
    elif "gemma3" in model_name:
        return (f"<start_of_turn>user\n{instruction}\n{context_and_query}\n<end_of_turn>\n<start_of_turn>model")
    elif "mistral" in model_name:
        return (f"<s>[INST] {instruction}\n{context_and_query}[/INST]")
    else:
        return (f"{instruction}\n{context_and_query}\n")
```

Program code 27. Modified response generating function for the AI judge

```
# Generate response using LLM
def generate_response_judge(query: str, context: str, rag_answer: str, prompt_instruction) -> str:
    custom_prompt = customize_prompt_judge(query, context, rag_answer, prompt_instruction)

    try:
        response = Settings.llm.complete(custom_prompt)
        return str(response)

    except Exception as e:
        logging.info(f"💔 Error generating response: {e}")
        return "❌ An error occurred while processing the query."
```

Program code 28. Main function adaptation for the AI judge

```
    # Set large language model
    set_llm_model_judge()
    logging.info("✅ Model set.")

    # Main loop for judge
    with open("../data/evaluation/batch_output/yymmdd_batch_rag_inference.json", "r", encoding="utf-8") as f:
        data = json.load(f)

    for idx, item in enumerate(data):
        # Take the Finnish question from the JSON file
        query = item.get("fi", "")
        context = item.get("context", "")
        rag_answer = item.get("rag_answer")
        logging.info(f"query, context, and rag_answer {idx +1} received. AI judge starting...")

        start_moment = time.time()
        judge_response_G = generate_response_judge(query, context, rag_answer, config.PROMPT_INSTRUCTION_G)
        item["Groundedness"] = judge_response_G
        groundedness_moment = time.time()
        groundedness_time = groundedness_moment - start_moment
        item["Groundedness_time"] = groundedness_time

        judge_response_AR = generate_response_judge(query, context, rag_answer, config.PROMPT_INSTRUCTION_AR)
        item["Answer_Relevance"] = judge_response_AR
        AR_moment = time.time()
        answer_relevance_time = AR_moment - groundedness_moment
        item["Answer_Relevance_time"] = answer_relevance_time

        judge_response_CR = generate_response_judge(query, context, rag_answer, config.PROMPT_INSTRUCTION_CR)
        item["Context_Relevance"] = judge_response_CR
        CR_moment = time.time()
        context_relevance_time = CR_moment - AR_moment
        item["Context_Relevance_time"] = context_relevance_time

        # SAVE ON THE FLY
        with open("../data/evaluation/ai_judge_output/yymmdd_rag_ai_judge.json", "w", encoding="utf-8") as f:
            json.dump(data, f, ensure_ascii=False, indent=2)

        logging.info(f"Processed item {idx+1}/{len(data)} and saved progress.")

    print("All done for Judge!")
```

Program code 29. RAG evaluation postprocessing functions

```
import pandas as pd
import re
import numpy as np
df = pd.read_json('../data/evaluation/ai_judge_output/file_name.json', encoding='utf-8')
```

```
5.
6.  def check_recall(row):
7.      heading = row['heading']
8.      retrieved_list = row['retrieved_headings']
9.      return int(heading in retrieved_list)
10.
11. df['recall'] = df.apply(check_recall, axis=1)
12.
13. def extract_score(text):
14.     match = re.search(r'Score:\s*(\d+)', text)
15.     if match:
16.         return int(match.group(1))  # return integer directly
17.     else:
18.         return np.nan
19.
20. df['groundedness_score'] = df['Groundedness'].apply(extract_score)
21. df['answer_relevance_score'] = df['Answer_Relevance'].apply(extract_score)
22. df['context_relevance_score'] = df['Context_Relevance'].apply(extract_score)
23.
24. df['groundedness_score'] = df['groundedness_score'].astype('Int64')
25. df['answer_relevance_score'] = df['answer_relevance_score'].astype('Int64')
26. df['context_relevance_score'] = df['context_relevance_score'].astype('Int64')
27.
28. df.to_json('../data/evaluation/ai_judge_output/yymmdd_rag_analysis.json',
    orient='records', indent=2, force_ascii=False)
```