

Enabling SQL Server as a vector database for the embedding storage in the RAG pattern

Mickson Bonsu Bossman

Master's Thesis
2025



Tomas Bata University in Zlín
Faculty of Applied Informatics

Tomas Bata University in Zlín
Faculty of Applied Informatics
Department of Informatics and Artificial Intelligence
Academic year: 2024/2025

ASSIGNMENT OF DIPLOMA THESIS

(project, art work, art performance)

Name and surname:	Mickson Bonsu Bossman
Personal number:	A23575
Study programme:	N0613A140023 Information Technologies
Specialization:	Software Engineering
Type of Study:	Full-time
Work topic:	Enabling SQL Server as a Vector Database for the Embedding Storage in the RAG Pattern

Theses guidelines

1. Review the literature about Retrieval Augmented Generation (RAG) pattern and related concepts, such as Generative AI, Embeddings models, vector databases and existing documentation for storing embeddings in databases.
 2. Analyze the technical requirements, capabilities, and limitations of using SQL Server as a vector database.
 3. Implement stored procedures, triggers, and SQL code to handle the vectorization of unstructured content, including text files, PDF documents, and images and the storage of those vectors.
 4. Develop a smart chatbot agent using a framework, such as Semantic Kernel or Langchain, that handles conversations and can answer questions about provided datasources, whose semantic representation is stored in SQL Server.
 5. Evaluate the chatbot's answers by considering metrics such as F1-score, accuracy, truthfulness, latency, and compare them with other vector databases.
 6. Recommend the best practices for use of SQL Server as a vector store of embeddings in real-world applications.
-

Form processing of diploma thesis: **printed/electronic**
Language of elaboration: **English**

Recommended resources:

1. ROTHMAN, Denis.. RAG-DRIVEN GENERATIVE AI: Build Custom Retrieval Augmented Generation Pipelines with Lla-maindex, . . . Deep Lake, and Pinecone. Packt, 2024
2. CAMPESATO, Oswald. Natural Language Processing Fundamentals for Developers: A Practical Guide to Building NLP Applications. Mercury Learning and Information, 2024.
3. LOPES, Pedro, LAHOUD, Pam. Learn T-SQL Querying – second edition: A Guide to Developing Efficient and Elegant T-SQL Code. Packt, 2024
4. KANDARPA, Raghav, SAXENA, Shivangi. Data Wrangling with SQL: A Hands-on Guide to Manipulating, Wrangling, and Engineering Data Using SQL. Packt, 2023.
5. AUFFARTH, Ben. Generative AI with LangChain: Build Large Language Model (LLM) Apps with Python, ChatGPT and Other LLMs. , 2023
6. LANHAM, Micheal. AI Agents in Action. Manning, 2024.
7. INFANTE, Roberto. AI Apps with LangChain, Manning, 2024.
8. MEYER, Lucas A. Building AI Applications with Microsoft Semantic Kernel: Easily Integrate Generative AI Capabilities and Copilot Experiences Into Your Applications. Packt, 2024.

Supervisors of diploma thesis: **MSc. Luis Antonio Beltran Prieto**
Department of Informatics and Artificial Intelligence

Date of assignment of diploma thesis: **October 27, 2024**

Submission deadline of diploma thesis: **June 2, 2025**



L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. m.p.
Dean

prof. Mgr. Roman Jašek, Ph.D., DBA m.p.
Head of Department

In Zlín November 17, 2024

MASTER'S THESIS AUTHOR'S DECLARATION

I take cognizance of the fact that

- by submitting my Master's thesis, I give consent to the publication of my thesis in accordance with the Act No. 111/1998 Coll., as amended, regardless of the result of the thesis defence;
- the Master's thesis will be stored in electronic form in the university information system and will be available for consultation;
- one copy of the Master's thesis in hard copy form will be kept and stored by Tomas Bata University in Zlín;
- my Master's thesis is fully subject to the Act No. 121/2000 Coll. on Copyright, on Rights Related to Copyright and on Amendment to Certain Acts (Copyright Act), as amended, particularly to Section 35 Paragraph 3;
- according to Section 60 Paragraph 1 of the Copyright Act, Tomas Bata University in Zlín has the right to conclude a licence agreement on the utilization of a school work as is specified in Section 12 Paragraph 4 of the Copyright Act;
- according to Section 60 Paragraphs 2 and 3, I may utilize my school work – Master's thesis – or grant a licence to utilize it only with the prior written consent of Tomas Bata University in Zlín, which in such a case is entitled to require me to pay a reasonable contribution in order to cover the costs incurred by Tomas Bata University in Zlín for the preparation of the work (up to their full amount);
- if software provided by Tomas Bata University in Zlín or other entities was used for the preparation of the Master's thesis only for learning or research purposes (i.e. for non-commercial use), the conclusions of the Master's thesis cannot be used for commercial purposes;
- if the output of the Master's thesis is any type of software product, the source codes and/or files of which the project is comprised are also considered part of the thesis; failure to submit this part may result in a failure to defend the thesis.

I declare that

- I have worked on my Master's thesis independently and properly cited the literature used; in case of publication of the conclusions, I shall be listed as co-author;
- the submitted hard copy version of the Master's thesis and the electronic version uploaded to the IS/STAG are identical in content.

In Zlín on

.....

author's signature

ABSTRAKT

Systémy Retrieval-Augmented Generation (RAG) integrují velké jazykové modely s externím vyhledáváním informací za účelem zvýšení kvality odpovědí. Klíčovým prvkem RAG pipeline je vektorová databáze, která uchovává a umožňuje dotazování na vysoce dimenzionální textová vektorová znázornění (embeddingy). Existující řešení se často opírají o specializované vektorové databáze, ale mnoho organizací již provozuje relační databázové systémy jako je SQL Server. Tato práce se zaměřuje na možnost využití SQL Serveru jako efektivní vektorové databáze pro ukládání embeddingů v RAG systémech.

Cílem je demonstrovat, že SQL Server dokáže uchovávat embeddingy a provádět vyhledávání podobnosti s vysokou přesností a nízkou latencí. Byl nakonfigurován Azure SQL Server s columnstore indexy optimalizovanými pro ukládání vektorů. Text dokumentů byl rozdělen do menších bloků (chunků) a embeddingy byly generovány pomocí embedding API od OpenAI. Pro vložení a načítání embeddingů byl použit SQL konektor v rámci Microsoft Semantic Kernel. Webová služba postavená na FastAPI zpracovává uživatelské dotazy, transformuje vstupní text na embedding, vyhledává podobné záznamy pomocí kosinové podobnosti v SQL a poskytuje odpovědi s využitím jazykového modelu OpenAI.

Hodnocení na základě benchmarkových dat ukazuje efektivní sémantické vyhledávání, kvantifikované standardními metrikami jako je recall a precision. Výkonové testy ukazují latence dotazů pod jednu sekundu, což potvrzuje, že vyhledávání ve vektorech pomocí SQL může být velmi rychlé. Tyto výsledky potvrzují, že SQL Server může splnit výkonnostní požadavky RAG systémů.

Tato studie přináší příspěvek pro akademickou i průmyslovou sféru tím, že ověřuje nový způsob využití existující SQL infrastruktury v oblasti RAG. Nabízí kompletní architekturu kombinující SQL Server, Semantic Kernel, OpenAI a FastAPI, čímž ukazuje, že moderní AI aplikace lze provozovat na tradiční relační databázové infrastruktuře. Výsledky naznačují, že organizace mohou využít SQL Server pro ukládání embeddingů, což usnadňuje nasazení a snižuje režii při zachování vysokého výkonu.

Klíčová slova: Retrieval-Augmented Generation, SQL Server, vektorová databáze, embeddingy, sémantické vyhledávání, OpenAI, Semantic Kernel

ABSTRACT

Retrieval-Augmented Generation (RAG) systems integrate large language models with external information retrieval to enhance answer quality. A key component of a RAG pipeline is a vector database to store and query high-dimensional text embeddings. Existing solutions often rely on specialized vector databases, but many organizations already maintain relational databases like SQL Server. This research addresses the problem of enabling SQL Server to function as an efficient vector database for embedding storage in RAG systems.

The objective is to demonstrate that SQL Server can store embeddings and perform similarity search with high accuracy and low latency. We configure an Azure SQL Database with columnstore indexes optimized for vector storage. Document text is segmented into chunks, and embeddings are generated via the OpenAI embedding API. Microsoft's Semantic Kernel SQL connector is used to insert and retrieve vectors in SQL Server. A FastAPI-based web service handles user queries, transforming input text into embeddings, retrieving similar results via SQL cosine similarity, and passing the retrieved context to an OpenAI language model for answer generation.

Evaluation on benchmark datasets shows effective semantic retrieval, quantified by standard metrics such as recall and precision. Performance tests report sub-second query latencies, confirming that SQL-based vector search can be extremely fast. These results validate that SQL Server can meet the performance demands of RAG systems.

This study contributes to both academia and industry by validating a novel use of existing SQL infrastructure for RAG. It provides a complete architecture combining SQL Server, Semantic Kernel, OpenAI, and FastAPI. In doing so, it shows that existing relational infrastructure can support modern AI workloads. Our findings suggest that organizations can leverage SQL Server for embedding storage, simplifying deployment and reducing overhead while maintaining retrieval performance.

Keywords: Retrieval-Augmented Generation, SQL Server, vector database, embeddings, semantic search, OpenAI, Semantic Kernel

ACKNOWLEDGEMENT

I am deeply grateful to my thesis supervisor, Ing. Luis Antonio Beltran Prieto for his guidance and support throughout this research. Their expertise in database systems and artificial intelligence greatly shaped this work. I also thank the faculty of Applied Informatics at Univerzita Tomáše Bati ve Zlíně for their encouragement and for fostering an academic environment conducive to research.

My sincere thanks go to the technical community and tool providers that supported this research. The open-source Semantic Kernel framework and FastAPI library were integral to the project architecture. I appreciate OpenAI for access to its language model APIs that made the generation of embeddings and responses possible. I also acknowledge Microsoft and the SQL Server team for providing the database platform and vector search capabilities used in this thesis. I am also thankful to the broader open-source community, whose shared knowledge and tools greatly facilitated this project.

Finally, I thank my friends and family for their patience and encouragement. Their unwavering support motivated me through challenges and helped me complete this work. I could not have completed this thesis without their support.

CONTENTS

INTRODUCTION	15
I THEORETICAL PART	18
1 LITERATURE REVIEW	19
1.1 RAG	19
1.2 TECHNIQUE DESCRIPTION	19
1.3 IMPORTANCE OF RAG BEFORE FINE-TUNING	19
1.3.1 Applications of RAG.....	20
1.4 EMBEDDINGS IN AI AND INFORMATION RETRIEVAL.....	21
1.4.1 Definition and Properties of Embeddings	21
1.4.2 Popular Embedding Models	22
1.4.3 Applications of Embeddings	22
1.5 VECTOR DATABASES.....	23
1.5.1 The Advantages for Vector Databases:.....	23
1.5.2 Prominent Vector Databases	23
1.5.3 Research on Vector Databases	24
1.6 SQL SERVER IN DATA MANAGEMENT	24
1.6.1 Extensibility Features	24
1.6.2 Limitations in Handling Vectors	25
1.6.3 Existing Adaptations	25
1.7 GAP IN LITERATURE	26
II PRACTICAL PART	27
2 METHODOLOGY	28
2.1 DATA COLLECTION AND EMBEDDING GENERATION	28
2.1.1 Data Source	28
2.1.2 Data Usage	28
2.2 EMBEDDING MODEL.....	29
2.2.1 Embedding Generation Process	29
2.3 DATA PREPROCESSING	30
2.4 CHUNKING	30
2.4.1 Importance of Chunking	30
2.4.2 Chunking Methods	30
2.5 SQL SERVER ADAPTATION FOR EMBEDDINGS	31
2.5.1 Schema Design.....	31
2.5.2 Indexing and Optimization.....	33
2.6 RAG-BASED CHATBOT DEVELOPMENT	34
2.6.1 Chatbot Architecture	34
2.6.2 Integration	34

2.6.3	Tools and Frameworks	35
3	SYSTEM ARCHITECTURE AND IMPLEMENTATION	36
3.1	SYSTEM ARCHITECTURE OVERVIEW	36
3.2	BACKEND IMPLEMENTATION AND ENDPOINTS	37
3.3	SEMANTIC KERNEL INTEGRATION	39
3.4	DOCUMENT CHUNKING AND EMBEDDING PROCESS	40
3.5	VECTOR SIMILARITY SEARCH IN SQL SERVER (STORED PROCEDURE)	45
3.6	END-TO-END SYSTEM WORKFLOW	49
3.7	ILLUSTRATIVE EXAMPLE OF SYSTEM OUTPUT (SAMPLE SCENARIO)	52
3.8	TECHNOLOGIES AND TOOLS USED	54
4	PERFORMANCE EVALUATION.....	56
4.1	EXPERIMENTAL DESIGN AND METHODOLOGY	56
4.1.1	Document Corpus.....	56
4.1.2	Query Design.....	57
4.1.3	Annotation Protocol and Scoring Rubric	58
4.1.4	Latency Measurement	59
4.2	QUANTITATIVE RESULTS.....	59
4.3	ANALYSIS AND INTERPRETATION	61
4.3.1	Semantic Retrieval Efficacy	61
4.3.2	Language model Influence	62
4.3.3	Retrieval Performance and Scalability	63
4.4	THREATS TO VALIDITY	63
4.5	COMPARISON OF LOCAL SQL SERVER AND AZURE AI SEARCH	64
5	CONCLUSION FUTURE WORK.....	70
5.1	SUMMARY OF CONTRIBUTIONS.....	70
5.2	KEY FINDINGS.....	70
5.3	LIMITATIONS	71
5.4	DIRECTIONS FOR FUTURE WORK.....	72
5.5	PRACTICAL RECOMMENDATIONS	74
5.5.1	Use SQL Server for Prototyping and Internal Deployments.....	74
5.5.2	Implement Preprocessing and Index Optimization	74
5.5.3	Avoid SQL Server for High-Scale Real-Time Applications.....	75
5.5.4	Hybrid Architecture for Scalability and Flexibility	75
5.5.5	Use Azure AI Search When Deploying in Cloud-Centric Workflows	75
5.5.6	Monitor and Benchmark Continuously	76
5.5.7	Security and Compliance Considerations.....	76
5.5.8	Educate Teams on Embedding Semantics and RAG Design	76
	LIST OF FIGURES	79
	LIST OF TABLES	80

LIST OF ABBREVIATIONS81

APPENDICES.....82

APPENDIX P I: FIRST APPENDIX TITLE.....83

INTRODUCTION

Generative AI technologies, such as OpenAI's GPT models, have advanced the way machines process and generate human-like language. However, these models often lack the ability to generate factually grounded responses due to their reliance on static training data. Retrieval-Augmented Generation (RAG) addresses this limitation by combining generative models with retrieval mechanisms that leverage external knowledge sources.

At the heart of the RAG pattern lies the concept of embeddings. These embeddings are high-dimensional vector representations of text, images, or other unstructured data that capture their semantic meaning. For instance, similar text snippets are represented by vectors that are geometrically close in an embedding space. This enables semantic search, a capability that surpasses traditional keyword-based search.

Problem Statement

Despite the importance of embeddings, most organizations rely on relational databases such as Microsoft SQL Server, which lack native support for vector operations. This presents a significant challenge for enterprises seeking to integrate RAG-based solutions without overhauling their existing database infrastructure.

This thesis investigates the feasibility of using SQL Server as a vector database by adapting its features to store, retrieve, and process embeddings. The research also explores the integration of SQL Server with a RAG-based chatbot to demonstrate its practical applications.

Objectives

The objectives of this research are as follows:

- **Feasibility Analysis:** Evaluate the technical capabilities of SQL Server for vector operations, including storage, indexing, and similarity search.
- **Implementation:** Develop mechanisms to handle embeddings in SQL Server, such as schema design, stored procedures, and triggers.
- **Chatbot Development:** Build a semantic chatbot that retrieves information from a SQL Server-based embeddings store.

- **Performance Comparison:** Measure and compare the performance of SQL Server against specialized vector databases, focusing on metrics such as accuracy, latency, and scalability.
- **Best Practices:** Recommend optimization strategies for using SQL Server as a vector database in real-world applications.

Research Questions

- How effectively can SQL Server store and retrieve high-dimensional embeddings for RAG-based systems?
- What are the performance trade-offs compared to dedicated vector databases?
- What practical strategies can enhance SQL Server's performance as a vector database?

Structure of the Thesis

This thesis is structured as follows:

- Chapter 1 provides a comprehensive literature review of RAG, embedding models, vector databases, and the technical capabilities of SQL Server, identifying gaps and overlaps in current research.
- Chapter 2 outlines the methodology used in this study, including the research design, technical implementation of SQL Server as a vector database, the development of a chatbot leveraging SQL Server, and the data collection and evaluation techniques.
- Chapter 3 presents the empirical results of the study, including the implementation outcomes, performance evaluation, and a comparative analysis of SQL Server against dedicated vector databases.
- Chapter 4 discusses the findings in detail, highlighting their significance, practical implications, limitations of the approach, and opportunities for future research and technological advancements.
- Chapter 5 concludes the thesis by summarizing the key findings, providing recommendations for developers and organizations considering SQL Server as a

vector database, and suggesting best practices for optimizing its use in RAG-based systems.

By exploring the adaptation of SQL Server for vector database functionality in the RAG pattern, this thesis aims to contribute to the broader conversation about embedding storage and retrieval in generative AI systems, providing valuable insights for both academia and industry practitioners.

I THEORETICAL PART

1 LITERATURE REVIEW

The literature on RAG, embeddings, vector databases, and SQL Server adaptations spans multiple disciplines, including natural language processing (NLP), database systems, and information retrieval. This section synthesizes key research and technological advancements to establish a robust theoretical foundation for this thesis. (1)

1.1 RAG

RAG represents a hybrid paradigm that combines the strengths of external data retrieval systems with generative AI models. Unlike purely generative systems, which rely solely on pre-trained knowledge, RAG enhances factual accuracy and relevance by grounding responses in real-world data retrieved from external sources. This approach bridges the gap between generative models and knowledge-intensive tasks, making it particularly valuable for applications requiring up-to-date, domain-specific, or highly accurate information (1).

1.2 Technique Description

The RAG framework operates in two stages: retrieval and generation. In the retrieval phase, a dense retrieval mechanism, often powered by vector embeddings, queries an external knowledge source (e.g., a document corpus or database) to fetch relevant information. This retrieved data is then passed to a generative model, such as a transformer-based language model, which synthesizes the information into a coherent and contextually appropriate response. By decoupling knowledge storage from the generative process, RAG allows models to access dynamic and extensive datasets without requiring retraining or fine-tuning. (1)

1.3 Importance of RAG Before Fine-Tuning

RAG offers several advantages over traditional fine-tuning approaches, particularly in terms of implementation difficulty, cost, and response quality:

- Implementation Difficulty:

Fine-tuning large language models for specific tasks or domains requires significant expertise, computational resources, and curated datasets (2). In contrast, RAG simplifies implementation by leveraging pre-trained models and external knowledge sources. This

reduces the need for extensive model customization, making it more accessible for organizations with limited technical resources (1).

- **Cost Efficiency:** Fine-tuning is resource-intensive, often requiring high-performance GPUs and substantial time investments. RAG, on the other hand, minimizes these costs by utilizing existing models and retrieval systems. The ability to dynamically retrieve information from external sources also reduces the need for frequent model updates, further lowering operational expenses (3).
- **Quality of Responses:** Fine-tuned models may struggle with out-of-domain queries or fail to provide accurate responses when faced with novel or evolving information (4). RAG addresses this limitation by grounding responses in real-time, external data, ensuring higher factual accuracy and relevance. This is particularly critical for applications like customer support, medical diagnosis, or legal advice, where precision is paramount (1).
- **Scalability and Adaptability:** RAG's modular design allows for easy integration with updated or expanded knowledge bases, ensuring that the system remains current without requiring retraining. This scalability makes RAG a more flexible and future-proof solution compared to fine-tuned models, which may become outdated as new data emerges (5).

1.3.1 Applications of RAG

Applications of RAG are extensive and include:

- **Knowledge-Based Chatbots:** Chatbots in industries like customer service and healthcare leverage RAG to provide accurate and context-aware responses. By integrating retrieval mechanisms with generative models, RAG ensures that responses are grounded in real-world data, improving both relevance and factual accuracy (1) (6).
- **Document Search and Summarization:** RAG aids in retrieving and summarizing large corpora of documents. By combining dense retrieval techniques with transformer-based architectures, RAG enables efficient and accurate document search and summarization (7).
- **Code Generation and Assistance:** Tools like GitHub Copilot use embeddings and RAG to assist developers with contextually relevant code suggestions. Retrieval-

augmented models enhance the ability of these tools to generate accurate and useful code snippets by leveraging external knowledge sources (8).

1.4 Embeddings in AI and Information Retrieval

Embeddings are critical to the success of RAG systems. They transform unstructured data into dense vector representations that preserve semantic meaning.

1.4.1 Definition and Properties of Embeddings

Embeddings are vector (mathematical) representations of data where linear distances capture structure in the original datasets.

The data could consist of words, which is also referred to as word embedding. But embeddings can also represent images, audio signals, and even large chunks of structured data (9).

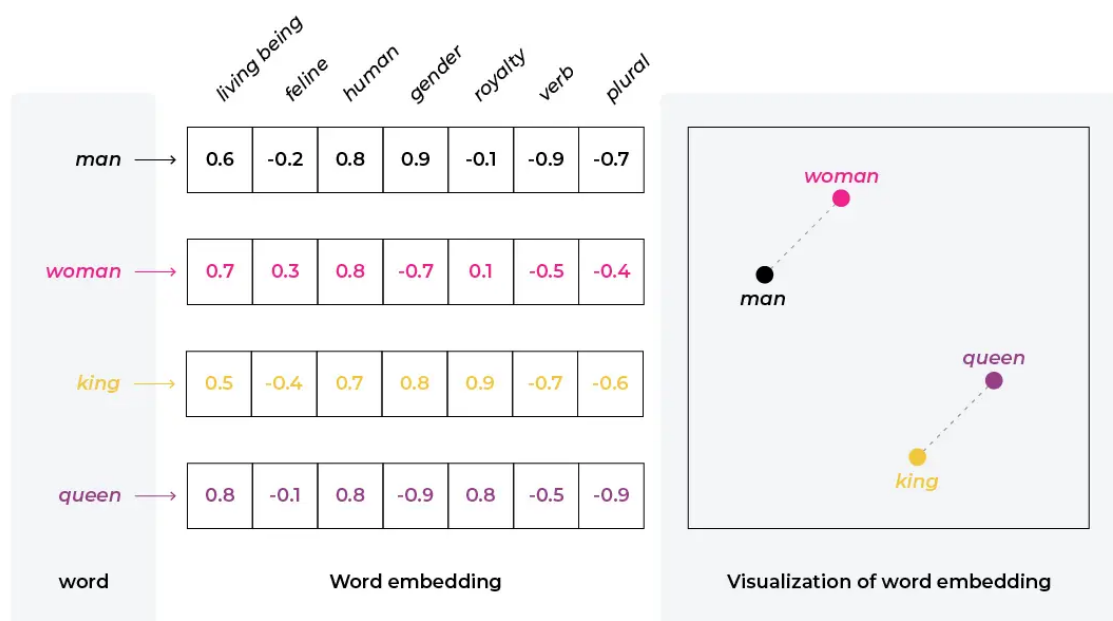


Figure 1 Example of an embedding

(9)

These are some of the key properties of an embedding:

- Dimensionality:** The size of embeddings varies by model. For example, OpenAI's text-embedding-ada-002 uses 1536 dimensions, while newer models like ada v3 may use 3072 dimensions or more to capture richer semantic information (10). Embeddings generated by one model cannot be used by another model.

- **Semantic Proximity:** Similar inputs, such as synonyms or related concepts, result in embeddings that are close in vector space. This property enables tasks like semantic search and clustering (11). One of the most common methods to determine if two vectors (and thus the texts they represent) are similar is by calculating the cosine similarity between them. Cosine similarity measures the cosine of the angle between two vectors in a multi-dimensional space, providing a value between -1 and 1 (10).
- **Task Adaptability:** Embeddings can be fine-tuned for specific tasks, such as classification, clustering, and similarity search. Fine-tuning adjusts the embeddings to better represent task-specific relationships (2).

1.4.2 Popular Embedding Models

Several embedding models have been widely adopted:

- **Word2Vec :** One of the earliest models for word embeddings, has a dimensionality of 300 and its used for word-level embeddings for semantic similarity ad analogy tasks. (11)
- **BERT:** Generates contextualized embeddings for sentences and phrases, widely used in downstream NLP tasks (2).
- **Sentence Transformers:** Optimized for sentence-level embeddings, enabling tasks like semantic search and clustering (12).
- **OpenAI's Embedding Models:** General-purpose text embeddings for semantic search and clustering, has a die (10).

1.4.3 Applications of Embeddings

- **Semantic Search:** Enables retrieval of documents based on semantic similarity rather than keyword matching.
- **Recommendation Systems:** Embeddings are used to match user preferences with content.
- **Clustering and Classification:** Embeddings facilitate unsupervised tasks like clustering similar documents.

1.5 Vector Databases

Vector databases are specialized systems designed to store, manage, and query high-dimensional embeddings efficiently. These databases are optimized for handling the unique challenges posed by vector data, such as performing fast similarity searches and nearest-neighbour queries. Their performance is critical for applications like RAG, where embedding similarity searches are frequent and must be executed in real-time (1) (3).

1.5.1 The Advantages for Vector Databases:

Vector databases excel in:

- **Handling High-Dimensional Data:** Embeddings generated by modern models (e.g., BERT, GPT, Sentence Transformers) are typically high-dimensional, ranging from 768 to 3072 dimensions or more. Traditional databases, which are designed for structured data, struggle to efficiently store and query such high-dimensional vectors.
- **Efficient Similarity Search:** Applications like RAG, semantic search, and recommendation systems rely on finding the most similar vectors (e.g., sentences, images, or user profiles) in a large dataset. Vector databases use specialized indexing techniques, such as Approximate Nearest Neighbour (ANN) algorithms, to perform these searches quickly and accurately.
- **Scalability:** As datasets grow, the ability to scale and perform real-time searches becomes crucial. Vector databases are designed to handle billions of vectors, making them suitable for large-scale applications like search engines, personalized recommendations, and AI-driven analytics.

1.5.2 Prominent Vector Databases

- **Pinecone:** A fully managed vector database that supports real-time similarity search and is widely used in AI applications like RAG and recommendation systems.
- **Milvus:** A scalable vector database designed for AI and machine learning applications, supporting billions of vectors and advanced indexing methods.
- **Weaviate:** An open-source vector database that combines vector search with traditional keyword search, making it suitable for hybrid retrieval tasks.

- **MongoDB:** MongoDB has introduced vector search capabilities in recent updates. This allows users to perform similarity searches on high-dimensional embeddings directly within MongoDB, making it a versatile choice for applications that require both traditional database functionality and vector search.

1.5.3 Research on Vector Databases

Recent research highlights the growing need for efficient vector storage and retrieval mechanisms (3), driven by the increasing adoption of AI-driven applications like RAG, semantic search, and recommendation systems. These applications rely heavily on high-dimensional embeddings, which pose unique challenges in terms of scalability, performance, and integration with existing data systems (1) (3) (12).

Studies on hybrid search methods emphasize the importance of combining structured queries with unstructured similarity search for improved performance. For instance, in applications like e-commerce or enterprise search, users often need to filter results based on structured metadata (e.g., price, date) while also retrieving semantically similar items. Hybrid search systems, such as those implemented in Weaviate and Elasticsearch, integrate vector search with traditional keyword-based or structured queries to provide a more flexible and powerful search experience (13).

1.6 SQL Server in Data Management

Microsoft SQL Server is a relational database management system (RDBMS) known for its robustness, scalability, and extensive ecosystem. While it is not natively equipped for vector operations, its extensibility makes it a candidate for adapting to handle embeddings.

1.6.1 Extensibility Features

SQL Server offers several features that can be leveraged for embedding management:

- **JSON Data Types:** SQL Server supports JSON data types, which allow embeddings to be stored as serialized JSON arrays. This flexibility enables the storage of high-dimensional vectors alongside structured data, facilitating seamless integration with AI workflows. JSON support also simplifies the process of querying and manipulating embeddings using SQL queries (14).

- **Stored Procedures:** Stored procedures in SQL Server can be used to automate tasks like vector insertion, retrieval, and similarity search. By encapsulating complex logic within stored procedures, developers can streamline embedding management and improve application performance (15).
- **Indexing:** SQL Server's indexing capabilities, including clustered and non-clustered indexes, can be used to optimize query performance when working with embeddings. While traditional indexing methods are not designed for high-dimensional data, they can still improve the efficiency of structured queries that involve embeddings (16).

1.6.2 Limitations in Handling Vectors

While SQL Server offers several features for managing structured and semi-structured data, it has notable limitations when it comes to handling high-dimensional vectors, such as embeddings. These limitations stem from its design as a traditional relational database management system (RDBMS), which is not optimized for vector-specific operations.

- **Lack of Native Vector Operations:** SQL Server does not have built-in functions for vector similarity search or high-dimensional indexing. Operations like cosine similarity, Euclidean distance, or ANN search, which are essential for working with embeddings, require custom implementations or integration with external libraries. This lack of native support makes SQL Server less efficient for tasks like semantic search or RAG compared to specialized vector databases (14).
- **Performance Overhead:** Managing high-dimensional embeddings in SQL Server can lead to significant performance overhead. Without specialized indexing techniques (e.g., HNSW or Product Quantization), queries involving embeddings may suffer from high latency, especially as the dataset grows. This limitation is particularly problematic for real-time applications that require fast and scalable similarity searches (3).

1.6.3 Existing Adaptations

Despite its limitations, SQL Server has been adapted in various studies and projects to handle vector data, particularly for applications involving embeddings. These adaptations often involve creative use of SQL Server's existing features, such as stored procedures and JSON support, to overcome the lack of native vector operations.

Microsoft's documentation highlights the use of JSON data types for managing large JSON objects, including high-dimensional embeddings. By serializing embeddings as JSON arrays, developers can store and query vector data alongside structured metadata. This approach simplifies data management but may require additional optimization for performance-critical applications (14).

1.7 Gap in Literature

Although extensive research exists on embeddings, vector databases, and SQL Server's capabilities, few studies have examined the use of relational databases like PostgreSQL and MariaDB in the context of RAG (17) (18), but not SQL Server. This gap underscores the need for this research, which seeks to bridge the fields of database management and AI.

II PRACTICAL PART

2 METHODOLOGY

The research design serves as a structured framework to systematically explore the feasibility of enabling SQL Server to function as a vector database in the RAG pattern. By providing a detailed outline of the experimental setup, tools, methodologies, and evaluation strategies, this section ensures that the research objectives are approached comprehensively and methodically.

With the growing adoption of RAG patterns in generative AI systems, efficient vector storage and similarity search have become critical challenges. While specialized vector databases, such as Pinecone and MongoDB, excel in handling embeddings, their integration into existing enterprise systems can pose significant hurdles. On the other hand, SQL Server is a widely adopted relational database management system that offers robustness, scalability, and enterprise-grade features, yet lacks native support for vector operations. This research seeks to bridge the gap by adapting SQL Server to manage high-dimensional embeddings and perform similarity searches within a practical RAG framework.

2.1 Data Collection and Embedding Generation

2.1.1 Data Source

A dataset of text documents will be used to generate embeddings. The dataset will be diverse and representation of real-world scenarios to ensure the generalization of the results.

The dataset will include:

- Knowledge base articles from organization repositories.
- Customer queries from chat logs or emails
- FAQ's from customer support systems
- Summaries from news websites

This variety ensures that the embeddings capture a wide range of linguistic patterns, contexts, and domains, making the model robust and applicable to different use cases.

2.1.2 Data Usage

The collected data is stored in a SQL Server table, where it undergoes normalization to ensure data integrity, reduce redundancy, and improve efficiency. Normalization is critical because it organizes the data into structured tables, minimizing duplication and ensuring

consistency. This process also simplifies querying and analysis, which is essential for generating high-quality embeddings.

The structure and schema of the SQL Server table, including details about the normalization process, is described in detail in the section 2.5.1. This ensures that the data is well-organized and ready for further processing, such as embedding generation and model training.

2.2 Embedding Model

OpenAI's embedding model, text-embedding-ada-002, will be used to generate embeddings. This model produces 1536-dimensional vectors and is optimized for semantic similarity tasks (10).

The text-embedding-ada-002 model was selected because:

- **Strong Performance:** outperforms all the old embedding models on text search, code search, and sentence similarity tasks and gets comparable performance on text classification.
- **Longer Context:** The context length of the model is increased by a factor of four, from 2048 to 8192, from the previous version, making it more convenient to work with long documents.
- **Smaller embedding size:** The model has only 1536 dimensions, one-eighth the size of davinci-001 embeddings, making the model more cost effective in working with vector databases.

2.2.1 Embedding Generation Process

- **Text Preprocessing:** Before generating embeddings, the text is often preprocessed to ensure consistency and remove noise. This includes normalization and chunking.
- Each text (chunk) document in the dataset will be passed to the OpenAI embedding API.
- The API will return a dense vector representation (embedding) of the text.
- The embeddings are then stored in SQL Server for retrieval during the RAG process.

The table used for storing the embeddings will be further described in this section 2.5.1

2.3 Data Preprocessing

- **Text Cleaning:** It's very necessary to clean the data before vectorizing them. Text cleaning will be done by removing unnecessary characters, stop words, and perform tokenization if needed.
- **Metadata Addition:** to facilitate easier retrieval and filtering, including some metadata (e.g., document source, and timestamp).

2.4 Chunking

In the context of building LLM-related applications, **chunking** is the process of breaking down large pieces of text into smaller segments (19). It's an essential technique that helps optimize the relevance of the content retrieved from a vector database once the LLM is used to embed content. There will be a consideration on other chunking methods then decide which one fits best for the thesis.

2.4.1 Importance of Chunking

- **Context Preservation:** Large documents may contain multiple topics or sections. Chunking ensures that each embedding represents a coherent and focused piece of text.
- **Embedding Quality:** OpenAI embeddings perform best when the input text is concise and contextually relevant. Chunking helps avoid embedding noise from overly long or diverse text segments.
- **Efficiency:** Smaller chunks reduce the computational overhead of processing and storing embeddings.

2.4.2 Chunking Methods

- **Fixed-Size Chunking:** This is the most common and straightforward approach to chunking: we simply decide the number of tokens in our chunk and, optionally, whether there should be any overlap between them. In general, we will want to keep some overlap between chunks to make sure that the semantic context doesn't get lost between chunks (19).
- **Sentences-Based Chunking:** Many models are optimized for embedding sentence-level content. Sentence chunking will be used in the implementation stage because

there are several approaches and tools available to do this, including: Naive splitting, Natural Language Toolkit (NLTK), and spaCy (19).

- **Semantic Chunking:** Break up the document into sentences. Create sentence groups: for each sentence, create a group containing some sentences before and after the given sentence. The group is essentially "anchored" by the sentence used to create it. You can decide the specific numbers before or after to include in each group - but all sentences in a group will be associated with one "anchor" sentence (19).

2.5 SQL Server Adaptation for Embeddings

2.5.1 Schema Design

A table structure will be adopted and created to store the embeddings and associated metadata. The structure of the table will contain:

The document_New table stores metadata and information about the original documents (e.g., PDFs).

Table 1: Document_New Table

Column Name	Data Type	Description
DocumentID	INT	Primary key, unique identifier for each document.
Metadata	NVARCHAR(MAX)	JSON or text containing metadata (e.g., title, author, source, timestamp).
DocumentNorm	FLOAT	Precomputed norm of the document's embedding (default = 0).
TextHash	BINARY(32)	SHA-256 hash of the document text for deduplication (unique constraint).

- **Clustered Index:** Automatically created on DocumentID (Primary Key).
- **Non-Clustered Index:** On TextHash for quick lookups to avoid duplicate documents.

DocumentChunk Table stores individual text chunks extracted from the documents.

Table 2: DocumentChunk Table

Column Name	Data Type	Description
DocumentID	INT	Foreign key linking to the Documents_New table in Table 1.
ChunkID	INT	Primary key, unique identifier for each text chunk.
ChunkIndex	INT	Index of the chunk within the document (e.g., 0, 1, 2, ...).
Text	NVARCHAR(MAX)	The actual text content of the chunk.
ChunkNorm	FLOAT	Precomputed norm of the chunk's embedding (default = 0).

- Clustered Index: Automatically created on ChunkID (Primary Key).
- Non-Clustered Index: On DocumentID for faster filtering by document.

DocumentEmbeddingValue Table stores the normalized embedding values for each text chunk.

Table 3: DocumentEmbeddingValue

Column Name	Data Type	Description
ChunkID	INT	Foreign key linking to the DocumentChunks in Table 2.
DimensionIndex	INT	Index of the embedding dimension (e.g., 0, 1, 2, ...).
Value	FLOAT	The value of the embedding at the specified dimension.

- Clustered Index: On (ChunkID, DimensionIndex) for efficient retrieval of embedding values.
- Primary Key: Non-clustered index on (ChunkID, DimensionIndex) to enforce uniqueness.

SimilarityCache Table stores cached similarity results for query embeddings to improve performance.

Table 4 SimilarityCache

Column Name	Data Type	Description
QueryEmbeddingHash	BINARY(32)	SHA-256 hash of the query embedding (Primary Key)
ChunkID	INT	The chunk ID that is most similar to the query.
Similarity	FLOAT	The similarity score between the query and the chunk.
CachedTime	DATETIME	Timestamp of when the cache entry was created or updated.

- **Clustered Index:** Automatically created on QueryEmbeddingHash (Primary Key).

ErrorLog Table logs errors that occur during operations (e.g., embedding generation, similarity search).

Table 5 ErrorLogs

Column Name	Data Type	Description
ErrorLogID	INT	Primary key, unique identifier for each error log entry.
ErrorMessage	NVARCHAR(MAX)	The error message.
ErrorTime	DATETIME	Timestamp of when the error occurred.

- **Clustered Index:** Automatically created on ErrorLogID (Primary Key).

2.5.2 Indexing and Optimization

- **Indexing:** To improve query performance, indexing strategies will be explored and implemented. Some known strategies that will be explored are:

- **Clustered Index:** It determines the physical order of records in the table. A table can have only one Clustered Index, and it is often created on the primary key column (16).
- **Non-Clustered Index:** A non-clustered index is a separate data structure from the table and contains a copy of selected columns in a sorted format. (e.g., source or timestamp) to speed up filtering (16).
- **Batch Processing:** Batch processing maximizes data availability by consolidating and reducing the amount of time taken to commit changes. When you fully process a dimension, any partition using that dimension is marked as unprocessed.

2.6 RAG-Based Chatbot Development

2.6.1 Chatbot Architecture

To facilitate a successful implementation of this thesis, there will be a creation of a chatbot which will mainly consist of two main components:

1. Retrieval Module:

- This module will be responsible for querying the SQL Server embeddings store to retrieve relevant documents based on the semantic similarity.
- It will also use the FindSimilarEmbeddings stored procedure to find the top-k most similar embeddings.

2. Generation Module:

- This module will also be responsible for using the generative model (e.g., GPT-4) to generate responses based on the retrieved documents.
- The retrieved documents are passed as context to the generative model.

2.6.2 Integration

- **Query Embedding Generation:** When a user submits a query, the chatbot sends the query to the OpenAI embedding API to generate a query embedding.
- **Similarity Search:** The query embedding is passed to SQL Server, which retrieves the top-k most similar embeddings (and their associated text). Cosine similarity was used to calculate the similarity between two vectors (the query embedding and already generated embeddings stored in the database).

- **Response Generation:** The retrieved text is passed to the generative model (e.g., GPT-4) as a context to generate a response.

2.6.3 Tools and Frameworks

- **Python 3.8:** For chatbot development and integration with OpenAI APIs.
- **Semantic kernel:** A framework for building RAG-based applications.
- **SQL Server:** mssql/server:2022-latest docker image was used as the server.
- **Azure Data Studio:** For managing the SQL Server database and executing queries.
- **Azure OpenAI Service's GPT-3.5-Turbo and Text Embedding Ada 002 models.**
- **NumPy and Pandas:** These Python libraries for data manipulation.
- **PyODBC:** The Python ODBC library used to connect to SQL Server from the FastAPI application.

3 SYSTEM ARCHITECTURE AND IMPLEMENTATION

3.1 System Architecture Overview

The proposed system is designed as RAG pipeline that integrates a web backend, a vector database implemented on SQL Server, and large language model services. The architecture consists of several key components: a **user interface** (via HTTP requests), a **FastAPI** backend application (serving as the orchestrator), integration with **Semantic Kernel** for combining retrieval and generation functionalities, an **Azure OpenAI** service providing embedding generation and chat completions, and a **SQL Server 2022** database for storing embeddings and metadata. Figure 2 illustrates the high-level architecture and data flow between these components.

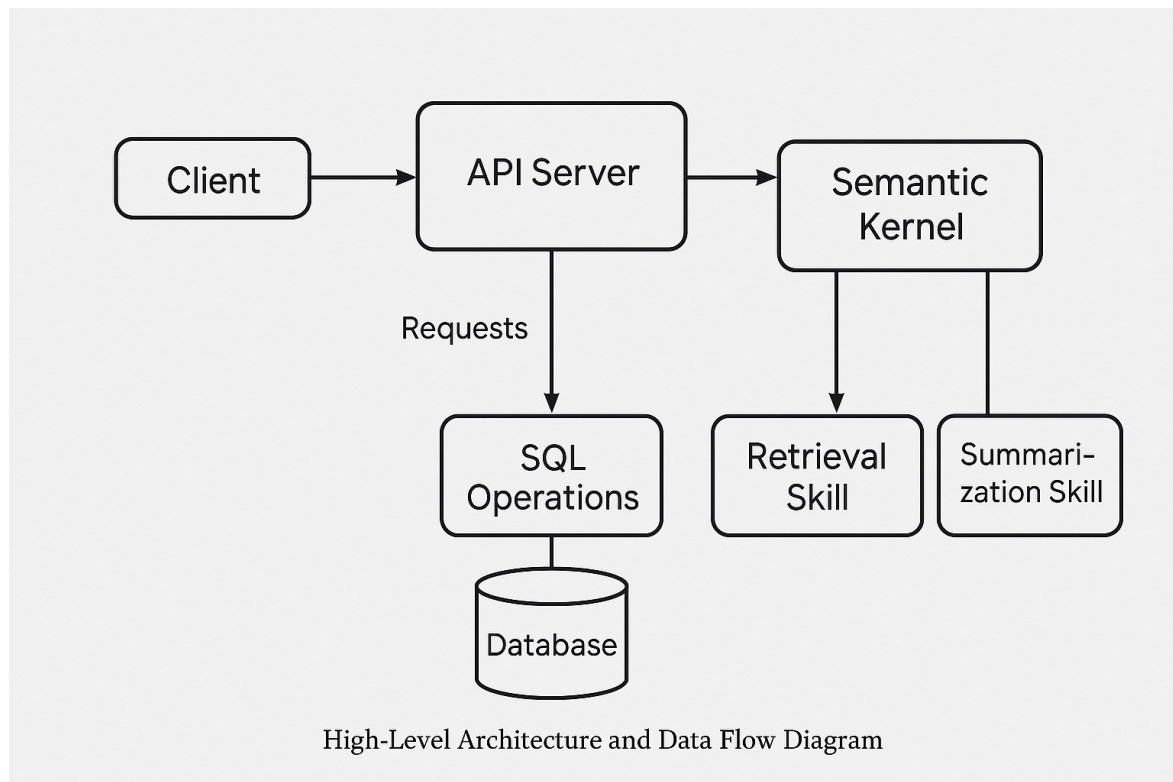


Figure 2: High Level Architecture

The user interacts with the system by sending HTTP requests (e.g. queries or documents to upload) to the FastAPI backend. The backend coordinates with the Semantic Kernel and Azure OpenAI services to process these requests. The SQL Server database is employed as a vector store to persist document embeddings and enable similarity search. This design leverages the strengths of each component: FastAPI for efficient web service hosting, Azure OpenAI for powerful language and embedding models, and SQL Server for reliable data

storage and indexing. By utilizing SQL Server's JSON handling and stored procedure capabilities, the system can perform vector similarity searches internally despite SQL Server's lack of native vector data types.

3.2 Backend Implementation and Endpoints

The backend is implemented in Python (v3.8) using the FastAPI web framework. FastAPI was chosen for its asynchronous capabilities and ease of defining RESTful endpoints. The service exposes a set of HTTP endpoints that correspond to the core functions of the RAG system: indexing new documents, retrieving relevant content, generating summaries, and conducting a chat-style Q&A. Each endpoint orchestrates a sequence of operations across the system's components. The primary endpoints are:

- **/index** – Document Ingestion Endpoint. This endpoint handles document uploads for indexing. When a user submits one or more documents (e.g., via an HTTP POST request with files or text), the backend processes each file by reading its content and extracting raw text (supporting formats such as PDF or CSV). The text is then segmented into smaller chunks using a chunking utility (described in Section 3.4). Each chunk (typically a few sentences) is transformed into an embedding vector via the Azure OpenAI embedding model. The resulting embeddings and associated metadata are stored in the SQL Server database. This involves inserting a new record for the document (with metadata such as filename or other descriptors) and multiple records for its chunks and embedding values in the tables described in. The endpoint returns a confirmation or an index identifier once storage is successful. In essence, the **/index** endpoint is responsible for populating the vector database with new embeddings so that they can be later retrieved by semantic searches.
- **/retrieve** – *Semantic Retrieval Endpoint*. This endpoint accepts a user's search query (typically a text string) pass it to the retrieval skill and returns a set of relevant document chunks from the database. Upon receiving a query, the FastAPI backend first generates the query's embedding vector using the same Azure OpenAI embedding model used for documents. This embedding is then passed to a SQL Server stored procedure (detailed in Section 3.5) which performs a cosine similarity search against the stored embeddings in the database. The stored procedure returns the top- k most similar chunks (by cosine similarity score) along with their content and identifiers. The backend collects these results and returns them to the user in a

JSON structure. For example, the response from `/retrieve` might be a JSON object containing a list of the top 5 chunks that are semantically closest to the query, each with its chunk text, source document ID, and a similarity score. This endpoint essentially provides the **knowledge retrieval** capability in the RAG pattern, allowing the user (or downstream components) to fetch sentence context based on semantic relevance rather than keyword matching.

- **`/generate_summary`** – *Summarization Endpoint*. This endpoint generates a summary of provided text or a set of retrieved documents. The client can invoke this endpoint either by supplying raw text to summarize or by providing the output from the `/retrieve` endpoint (i.e. a set of document chunks). In the latter case, the backend will combine the content of the retrieved chunks and invoke the Azure OpenAI **chat completion** model (GPT) with a prompt instructing it to condense the information. The Semantic Kernel's **SummarizationSkill** (see Section 3.3) is utilized here to orchestrate the process: it takes in the documents' text and produces a concise summary. Internally, this may involve a system prompt such as *"Summarize the following content:"* followed by the concatenated chunks. The model then returns a summarized response from the endpoint. This functionality is particularly useful for distilling long documents or the combined search results into a shorter form, capturing the key points. For instance, if a user has retrieved several related chunks on a topic, they can use `/generate_summary` to get a high-level overview before deciding on further actions.
- **`/chat`** – *Conversational Q&A Endpoint*. This is the integrated chat interface of the system, which allows a user to ask questions in natural language and receive answers that are grounded in the indexed documents. When a request is made to `/chat` with a user's query, the backend executes the full RAG cycle: it first calls the retrieval process (essentially using the same mechanism as `/retrieve`) to obtain the most relevant chunks for the query. The retrieved context is then put into summary using the `/generate_summary` endpoint and finally fed into the Azure OpenAI chat completion model to generate a response. This step is handled via the Semantic Kernel's orchestration, using a prompt template or skill that incorporates the retrieved documents as context. The system constructs a prompt for GPT that includes the user's question and the retrieved supporting information (for example, by prepending a system message like "Use the following context to answer the

question” and listing the content of the top chunks). The chat completion model (such as GPT-4 deployed on Azure) then produces a conversational answer that attempts to give response based on the user’s query using the given context. The response is finally returned to the user as the output of the /chat endpoint. This endpoint provides an interactive experience where the user can ask questions one at a time; each query is handled independently by retrieving fresh relevant information and generating a response. In this way, the chat endpoint realizes the RAG pattern by augmenting the generative model with retrieval for each query. The use of the chat completion model means the responses can be phrased in a fluent, human-like manner, often incorporating the facts from the documents to directly address the question.

3.3 Semantic Kernel Integration

To manage the complex interactions between retrieval and generation, the system integrates **Microsoft Semantic Kernel** as an orchestration layer within the backend. In this architecture, SK is used to coordinate Azure OpenAI services (for chat and embeddings) and to incorporate custom logic for retrieval and summarization as *skills*. The Semantic Kernel is initialized in the FastAPI application at startup, configured with the appropriate Azure OpenAI deployment details. Specifically, an **AzureTextEmbedding** service and an **AzureChatCompletion** service are added to the kernel, corresponding to the embedding model (e.g., text-embedding-ada-002) and the chat model (e.g., GPT-4) respectively. These SK connectors are initialized with the Azure OpenAI *endpoint URL*, *API key*, *API version*, and the specific *deployment names* of the models. By registering these services, the kernel can invoke the Azure OpenAI API as needed for generating embeddings or completing chat prompts, without repeatedly specifying credentials.

In addition to the AI services, two custom SK skills are implemented: **RetrievalSkill** and **SummarizationSkill**. Each is defined as a Python class with functions decorated by `@kernel_function`, making them accessible to the kernel as semantic functions. The *RetrievalSkill* encapsulates the logic to fetch relevant documents from the database given a query. Its primary function (e.g., `retrieve_documents`) takes a query string and an optional parameter *k*, and it internally calls the backend’s retriever utility (Section 3.5) to perform the similarity search. This function returns a JSON string containing the top-*k* documents (chunks) relevant to the query. By registering *RetrievalSkill* with SK (e.g.,

`kernel.import_skill(RetrievalSkill(), "retrieval"))`, the system enables the LLM or other skills to invoke document retrieval as part of a larger semantic pipeline.

Similarly, the `SummarizationSkill` provides functions to condense text and to answer queries based on text. One function (e.g., `summarize_text`) accepts input which could be a batch of documents (in JSON form) or raw text and returns a summary produced by calling the Azure OpenAI chat model. Another function (e.g., `query_summary`) can take a structured input containing a summary and a user query and generate a direct answer using that summary as context (this is effectively a specialized prompt to the chat model to perform Q&A on the given summary). By importing `SummarizationSkill` into the kernel, these capabilities become available as semantic functions.

The integration of SK simplifies the orchestration of multi-step operations. For instance, the chat workflow can be implemented by composing the skills: first call the retrieval skill to get relevant documents, then call the summarization skill or directly the chat service to generate an answer. Semantic Kernel allows these calls to be chained programmatically or even within a single prompt using its planners, but in this implementation the chaining is done in the backend code for clarity. The SK kernel manages the state and passing of data between these skill functions. For example, the backend's `/chat` endpoint (Section 3.2) can utilize SK by creating a context for the user query, invoking the `retrieval.retrieve_documents` function to get context documents, then feeding the result into a prompt (or a `summarization.query_summary` call) to produce the final answer. This approach has the advantage of treating retrieval and summarization as **first-class operations** in the semantic pipeline, orchestrated by a single controller (the kernel). It also decouples the retrieval logic from the core application code – the kernel can invoke the retrieval skill as if it were just another AI function, which makes the solution more modular.

3.4 Document Chunking and Embedding Process

A crucial step in enabling semantic search is transforming unstructured documents into a form suitable for vector storage and comparison. In this implementation, each document is broken into smaller segments (chunks), and each chunk is converted into an embedding vector using an OpenAI model. This section details the chunking strategy and the embedding generation.

Chunking Strategy: Documents such as articles or PDF text are often lengthy, and sending the entire text to an embedding model is inefficient and unnecessary. Instead, the

system uses a *sliding window of sentences* approach to create chunks of manageable size. The chunking utility reads the raw text of a document and splits it into segments of N sentences (with $N=10$ by default in our experiments). Sentence boundaries are identified using regular expression heuristics (splitting on punctuation like., ?, or ! followed by a space). By grouping sentences, the chunker ensures that each chunk is a semantically coherent piece of text, typically a paragraph or part of a topic, rather than an arbitrary cut of characters. The text is also normalized (whitespace trimmed, stray punctuation cleaned) to maintain consistency. This sentence-based chunking has the advantage of preserving meaning across sentence boundaries (e.g., a statement and its explanation stay together) while keeping chunks small enough for efficient embedding. Each chunk is labeled with an index indicating its position in the original document (e.g., chunk 0, chunk 1, etc.). If a document is very short, it might result in just one chunk; very long documents yield many chunks. In this manner, a single large document is represented as a sequence of embedding vectors, one per chunk, allowing fine-grained matching of a query to the most relevant part of the document.

Embedding Generation: Once chunks are prepared, the system generates embeddings for each chunk using the Azure OpenAI embedding model. The chosen model is text-embedding-ada-002, a state-of-the-art text embedding model from OpenAI's Ada series. This model produces a 1,536-dimensional vector representation for any given text input. Notably, the embedding dimension is fixed (1536) regardless of the length or complexity of the input text, providing a consistent vector size for all chunks. The backend uses Azure OpenAI's embedding endpoint to obtain these vectors. For efficiency, embeddings are fetched in batches: the FastAPI backend can send multiple chunks in a single API request to Azure OpenAI, and the service returns a list of embeddings in one response. Batch processing improves throughput and ensures that the embedding generation step does not become a bottleneck when indexing many documents. In practice, when a new document is ingested through the /index endpoint, the system will accumulate all chunk texts from that document (and potentially multiple documents if uploading several at once), send them in a batch to the embedding API, and receive the corresponding list of vectors. Each embedding is represented as an array of 1536 floating-point numbers.

After obtaining the embedding for each chunk, the system computes additional metadata. One useful quantity is the vector norm (Euclidean length) of the embedding. The norm is calculated as the square root of the sum of squares of all components of the vector.

```

SELECT
    dev.ChunkID,
    SUM(qe.QueryValue * dev.Value) AS DotProduct,
    SQRT(SUM(qe.QueryValue * qe.QueryValue)) AS QueryNorm,
    dc.ChunkNorm
FROM dbo.DocumentEmbeddingValues dev
JOIN #QueryEmbedding qe ON dev.DimensionIndex = qe.DimensionIndex
JOIN dbo.DocumentChunks dc ON dev.ChunkID = dc.ChunkID
GROUP BY dev.ChunkID, dc.ChunkNorm

```

Figure 3: Vector Norm

Both document-level and chunk-level norms can be computed; in the implementation, a norm for each chunk is calculated and stored. The norm is later used to help compute cosine similarity (and could also serve as a quick check for zero-vectors or debugging). At this stage, the system also creates a unique identifier or hash for the source document (e.g., using an SHA-256 hash of the full text) to detect duplicate documents. This prevents indexing the same content multiple times. Figure 3: Vector Norm shows how the vector norm was implemented.

Storing Embeddings: the final step of the indexing process is to store the chunk data in SQL Server. The storage schema is designed to accommodate potentially large numbers of embeddings and to facilitate efficient similarity calculations. Three tables are used in the database to store the information in a normalized form:

- **Documents table:** Each row corresponds to an original document. It contains a DocumentID (primary key) and metadata such as the document name or source, a JSON field for any additional metadata, a hash of the document text (for deduplication), and possibly a pre-computed document-level embedding or norm. (In our case, a document-level embedding and its norm can be stored for completeness, though retrieval is done at chunk level.)
- **DocumentChunks table:** Each row represents a chunk of a document. It includes a ChunkID (primary key) and a foreign key DocumentID linking to the parent document. It also stores the chunk's text content (Text), the chunk's index within the document (ChunkIndex), and the L2 norm of that chunk's embedding (ChunkNorm). Storing ChunkNorm avoids recalculating the magnitude of the vector during similarity computation.

- **DocumentEmbeddingValues table:** This table stores the numerical components of each chunk's embedding vector in a vertical (normalized) format. Each row has a foreign key ChunkID (linking to the DocumentChunks table), a DimensionIndex (an integer from 0 to 1535 indexing the position in the vector), and a Value (the floating-point value of the embedding at that dimension). For example, if a chunk's embedding vector is $[v_0, v_1, v_2, \dots, v_{1535}]$, then the table will have 1536 rows for that chunk: (ChunkID, DimensionIndex=0, Value= v_0), (ChunkID, DimensionIndex=1, Value= v_1), and so on. While this design results in many rows, it enables set-based operations in SQL for computing similarity (as described in the next section). By contrast, storing the entire vector in a single cell (e.g., as a JSON array or binary blob) would make it harder for the database to perform computations across vectors using SQL set operations. The normalized schema, combined with appropriate indexing (such as indexing DimensionIndex and perhaps using a clustered columnstore index for compressing these value rows), allows the database engine to handle dot-product calculations relatively efficiently.

During the indexing operation (/index), the backend performs a series of inserts: a new Documents row (if the document is not a duplicate), then multiple DocumentChunks inserts (one per chunk, retrieving each new ChunkID), and finally a batch insert into DocumentEmbeddingValues for all the embedding components. These inserts are executed using parameterized SQL queries using a Python ODBC connection. The entire operation is wrapped in a transaction to ensure consistency (if any step fails, the transaction can be rolled back so that partial data is not left in the tables). After a successful commit, the document and its chunks become part of the persistent knowledge store that can be queried. Figure 4: Database table relationship shows the relationship between the tables using their primary and foreign keys.

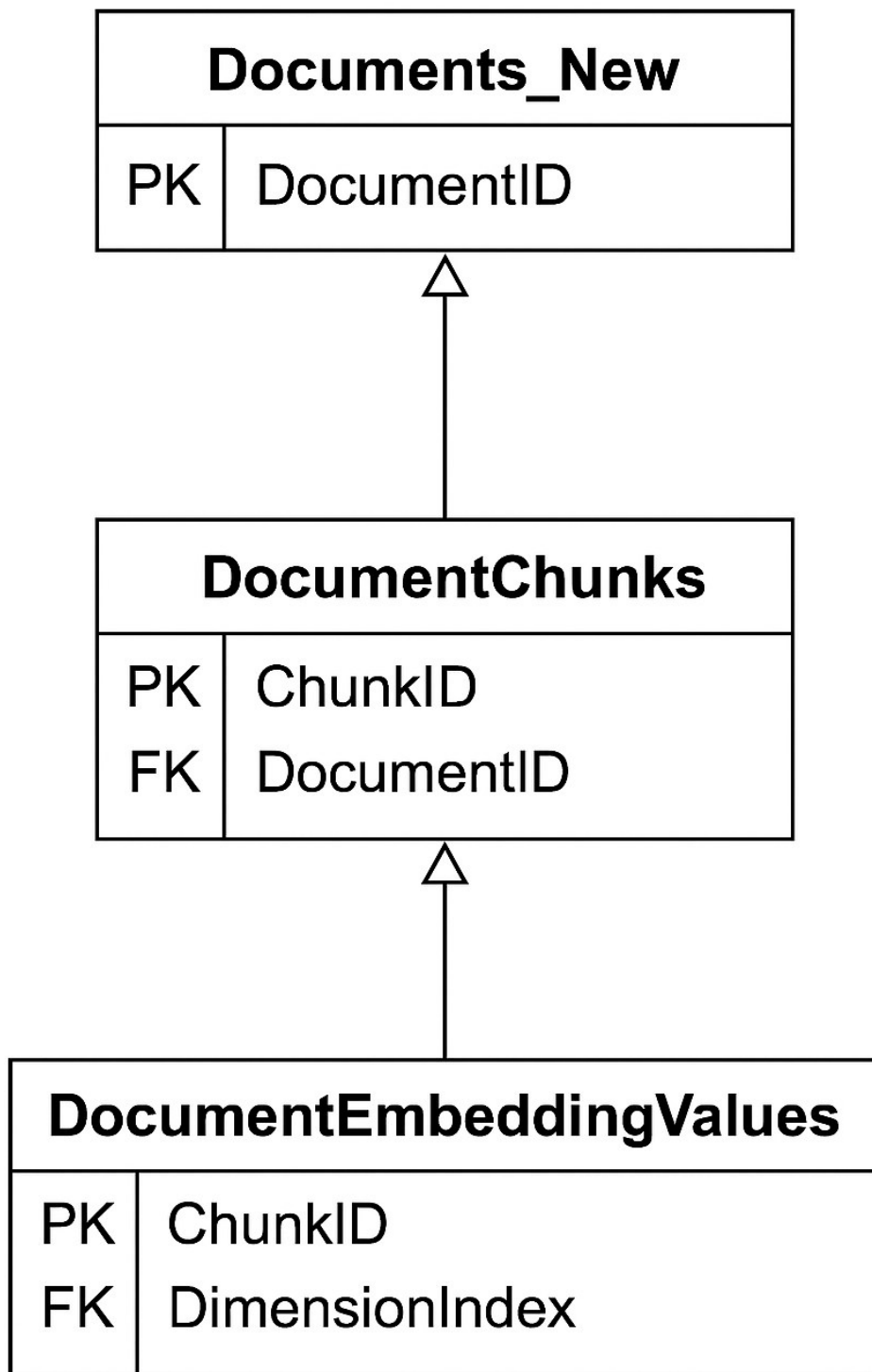


Figure 4: Database table relationship

3.5 Vector Similarity Search in SQL Server (Stored Procedure)

With documents indexed as vectors in the database, the system needs to efficiently find which stored vector is like the new query's embedding. This is achieved through a dedicated SQL Server stored procedure named **FindSimilarChunks_SetBased**. The stored procedure implements a *cosine similarity* search between a given query vector and all chunk vectors stored in the database and returns the top matches. This approach leverages T-SQL set-based operations and JSON functions to perform the vector math directly in SQL Server.

Input and JSON Parsing: The FindSimilarChunks_SetBased procedure is designed to be called with two parameters: firstly, a JSON-formatted embedding vector for the query, and secondly an integer @topK specifying how many results to return. The query's embedding is prepared in the application layer (FastAPI) by generating the embedding via Azure OpenAI and serializing the resulting list of 1536 floats into a JSON array (e.g., "[0.123, -0.045, ...]"). SQL Server 2022 has robust JSON parsing capabilities, which the procedure uses to ingest this vector. Inside the stored procedure, the JSON string is parsed into a tabular format using the built-in OPENJSON function.

Specifically, OPENJSON(@queryVectorJson) produces a table of keys and values, where each key is the index in the array and each value is the numeric value at that index. By casting these values to FLOAT, we obtain a table (let's call it @QueryVecTable) of dimension indices and the query's embedding values. Additionally, the procedure computes the L2 norm of the query vector from this table (using SQRT(SUM(value^2))) to use in cosine similarity calculation.

Cosine Similarity Calculation: Cosine similarity between two vectors a and b is given by the dot product of a and b divided by the product of their magnitudes: dot product of vector A and vector B / (magnitude of vector A * magnitude of vector B)

In the context of our database, the query vector is a and a chunk's vector is b . The stored procedure computes the dot product $a \cdot b$ by joining the query vector values with the stored embedding values of each chunk on the dimension index. In simplified terms, the procedure executes a query similar to:

```
WITH QueryVec AS (
    SELECT CAST([key] AS INT) AS DimensionIndex,
           CAST(value AS FLOAT) AS QueryValue
```

```

        FROM OPENJSON(@QueryEmbedding)
    ),
    SimilarityCalc AS (
        SELECT
            dev.ChunkID,
            SUM(qv.QueryValue * dev.Value)          AS DotProduct,
            SQRT(SUM(qv.QueryValue * qv.QueryValue)) AS QueryNorm,
            dc.ChunkNorm
        FROM dbo.DocumentEmbeddingValues AS dev
        JOIN QueryVec AS qv
            ON dev.DimensionIndex = qv.DimensionIndex
        JOIN dbo.DocumentChunks AS dc
            ON dev.ChunkID = dc.ChunkID
        GROUP BY dev.ChunkID, dc.ChunkNorm
    )
    SELECT TOP (@TopK)
        sc.ChunkID,
        dc.Text,
        dc.DocumentID,
        dc.ChunkIndex,
        sc.DotProduct / NULLIF(sc.QueryNorm * sc.ChunkNorm, 0) AS
        Similarity
    FROM SimilarityCalc AS sc
    JOIN dbo.DocumentChunks AS dc
        ON sc.ChunkID = dc.ChunkID
    ORDER BY Similarity DESC;

```

Listing 1: Vector Similarity Search

Listing 1: Vector Similarity Search demonstrates the set-based approach: the query vector table `q` is joined with the pre-stored embedding values `v` such that each pairing multiplies one dimension of the query vector with the corresponding dimension of a chunk's vector. By summing over all dimensions ($\text{SUM}(q.\text{Val} * v.\text{Value})$), we obtain the dot product of the query with each chunk vector. Each chunk's precomputed norm (`ChunkNorm`) and the query norm (`@queryNorm`) are then used to normalize this dot product, yielding the cosine similarity. This calculation is essentially the formula for cosine similarity implemented in pure SQL aggregation. The results are grouped by `ChunkID` (to accumulate the sum for each chunk) and sorted in descending order of similarity.

Top-K Retrieval: The stored procedure uses the `@topK` parameter to limit the output to the k most similar chunks. In practice, after computing the similarity for all chunks, a

SELECT TOP(@topK) is performed on the ordered results. Each result includes the chunk’s identifier, the ID of its source document, the chunk’s text content, its index in the document, and the similarity score. These are the pieces of data the application needs to identify the context to feed into the language model. The procedure might also exclude or filter out very low similarity scores or enforce a minimum threshold if needed (to avoid returning irrelevant text), but in our implementation we primarily rely on the Top-K cutoff.

Set-Based vs. Iterative Approach: The suffix “_SetBased” in the procedure’s name emphasizes that the computation is done with set operations (SQL set joins and aggregates) rather than a cursor or row-by-row computation. This is important for performance, as SQL Server is optimized for set-based queries. By leveraging JOIN and GROUP BY over the large DocumentEmbeddingValues table, the query allows SQL Server’s query optimizer to potentially use parallel execution and indexes. In our design, DocumentEmbeddingValues can benefit from an index on DimensionIndex (and possibly a composite index including ChunkID) to speed up the join with the query vector.

Table 6: Indexes Optimized for Embedding Retrieval

Table	Type	Column Involved	Purpose
DocumentEmbeddingValues	Clustered	(ChunkID, DimensionIndex)	Speeds up lookup of embeddings during similarity search.
Documents_New	Nonclustered	DocumentID with Metadata included	Speeds up access by DocumentID, and retrieves Metadata efficiently without touching the base table.

Additionally, given the large volume of rows in DocumentEmbeddingValues, a columnstore index could be created on that table to drastically improve aggregation performance and storage footprint. Such an index stores the values column-wise and is well-suited for heavy summation of numeric data (as in dot product calculations).

Output Handling: Once the stored procedure computes the Top-K similar chunks, the FastAPI backend reads the result via the ODBC connection. The resultset is a list of rows, each with the fields (ChunkID, Text, DocumentID, ChunkIndex, Similarity). The backend code then formats this into a JSON structure. For consistency, the output JSON from the retrieval function (and the SK RetrievalSkill) is structured as:

```
"ChunkID": 207,  
"Text": "We want their eyebrows to rise as they scroll through the modules. We want them to perceive that professionals have been at work. If instead they see a scrambled mass of code that looks like it was written by a bevy of drunken sailors, then they are likely to conclude that the same inattention to detail pervades every other aspect of the project. www.it-ebooks.info 76 Chapter 5: Formatting You should take care that your code is nicely formatted. You should choose a set of simple rules that govern the format of your code, and then you should consistently apply those rules. If you are working on a team, then the team should agree to a single set of formatting rules and all members should comply. It helps to have an automated tool that can apply those formatting rules for you. The Purpose of Formatting First of all, let's be clear. Code formatting is important. It is too important to ignore and it is too important to treat religiously.",  
"DocumentID": 742,  
"ChunkIndex": 0,  
"Similarity": 0.8625169667703961
```

Figure 5: Structure of the response

This JSON format makes it straightforward for downstream processing: the SummarizationSkill, for example, expects a JSON with a "documents" array that it can then combine and summarize. It also provides a clear interface for any UI or client: the client can see which document a chunk came from and even the similarity score (which might be used to decide if the answer's confidence is high). The similarity scores are typically between 0 and 1 (since they are cosine similarities; 1 means identical direction vectors). If no relevant chunk is found above a certain threshold, the procedure could return an empty set (and the JSON would then have "documents": []).

In essence, the FindSimilarChunks_SetBased stored procedure enables SQL Server to function as a rudimentary **vector database**. It uses the data stored (embeddings split by dimension) and computes similarity in-database using straightforward linear algebra operations expressed in SQL. This demonstrates that even without a specialized vector index or nearest-neighbour index structure, SQL Server can leverage its JSON support and compute capabilities to perform the core vector search tasks. The trade-off is that this approach may be slower than purpose-built vector databases for very large corpora, but it has the advantage of simplicity and integration – all data stays in the relational database, and standard T-SQL is used. This method was sufficient for the scope of the project, and evaluation can be found in the Performance Evaluation section.

3.6 End-to-End System Workflow

Having described each component of the system, Figure 6: Data workflow diagram represent the end-to-end workflow for handling a user's query.

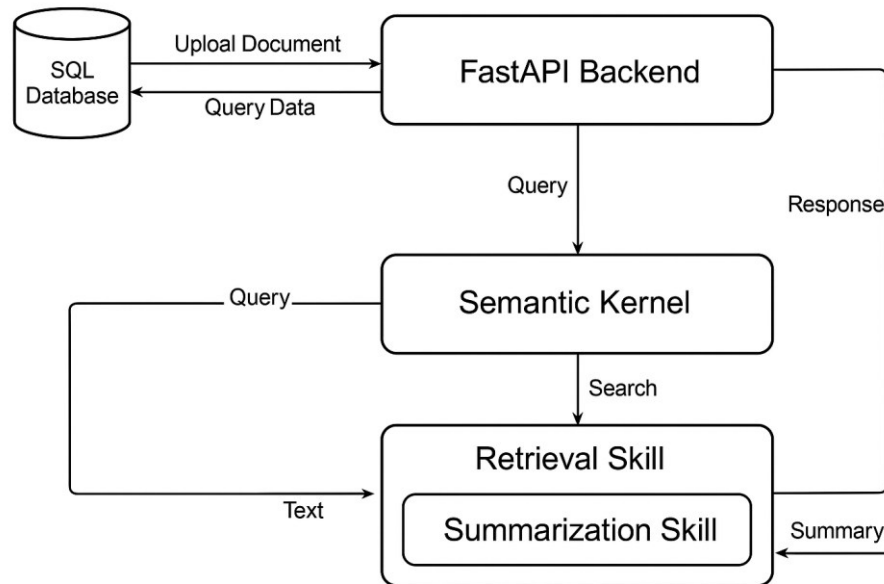


Figure 6: Data workflow diagram

Figure 6: Data workflow diagram outlines the sequence of operations that occur from the moment a query is received to the point an answer is returned. The following steps summarize the process:

- User Query Submission:** A user (or calling application) issues a query or prompt via the FastAPI endpoint (either using the `/chat` endpoint for a one-step answer or using `/retrieve` followed by `/generate_summary` for a more manual process). For this example, assume the user hits the `/chat` endpoint with the question: “*What is code formatting?*”.
- Similarity Search in SQL:** The backend then calls the stored procedure `FindSimilarChunks_SetBased`, passing the query's embedding (as a JSON array string) and a chosen `top_k` (say 5). SQL Server parses the embedding and computes cosine similarities against all stored chunk embeddings as described in Section 3.5. The result is a ranked list of the most similar chunks. For example, the procedure might return 5 chunks with similarity scores like 0.92, 0.85, 0.80, 0.75, 0.75. These chunks are likely from different documents that all pertain to the “code formatting”

mentioned in the query. Each chunk might contain a portion of that talks about it. The top result (0.92) could be a chunk whose text is “We want their eyebrows to rise as they scroll through the modules. We want them to perceive that professionals have been at work.....”, which indeed is highly relevant to the query about main findings. The FastAPI backend receives these results and structure them as JSON, which is called `retrieved_docs_json`.

- **Context Preparation:** Depending on the endpoint logic, the system now prepares a prompt for the generative model. In the `/chat` endpoint scenario, the backend will combine the user’s query and the retrieved documents to form the input for Azure OpenAI’s chat completion. This is done with guidance from the Semantic Kernel. A typical strategy is to use a system or assistant prompt that provides the retrieved chunks as context. For example, a system message might be: *“You are an AI assistant. Use the following information to answer the user’s question. Context:*

[Document 7, Chunk 3]: We want their eyebrows to rise as they scroll through the modules. We want them to perceive that professionals have been at work.....

[Document 12, Chunk 1]: What this experience has taught me, through long trial and error, is that functions should be very small. In the eighties we used to say that a function should be no bigger than a screen-full....

In our implementation, the Semantic Kernel’s skills simplify this step: the retrieval skill already provided `retrieved_docs_json`, and the summarization or answer generation skill can take this JSON and the original query to craft an appropriate prompt internally. In some cases, we explicitly call the `SummarizationSkill`’s `query_summary` function, which under the hood concatenates the context and query and calls the OpenAI model.

- **Generative Response with LLM:** The Azure OpenAI **Chat Completion** service (through Semantic Kernel’s `AzureChatCompletion` connector) is invoked with the prepared prompt. The model (e.g., GPT-4) processes the prompt, which includes both the user’s question and the context from the documents and generates a response. Thanks to the provided context, the model’s answer should be grounded in the actual content of the documents rather than purely from its own parametric knowledge. For instance, the model might respond: *“According to the XYZ report, the main findings were a 5% increase in market demand in Q4 and identification of supply chain*

challenges that affected that quarter's performance." This answer is composed by the model, but it uses the information from the context (the retrieved chunks) to ensure accuracy and relevance. The chat completion result is captured by the backend.

- **Response Delivery:** The FastAPI backend receives the model's answer and sends it back to the user as the response to the /chat request. In the above example, the user would see a concise answer stating the main findings of the report, which directly addresses the query. If the conversation were interactive (multi-turn), the system might also append the user question and answer to a chat history (not extensively implemented in this prototype, which treats each /chat call independently).
- **Iterative Interaction (if applicable):** Although each query through /chat is handled independently in our implementation, a user can ask follow-up questions in separate calls. In a more advanced integration, one could maintain conversational state or use the Semantic Kernel's memory to store context from previous interactions. For this thesis project's scope, the focus was on single-turn QA with retrieval. However, the system architecture can be extended to multi-turn dialogues by preserving the conversation and possibly reusing or re-fetching context as needed.

Throughout this workflow, several optimizations and design choices ensure performance and accuracy: the use of vector embeddings for semantic matching (so the query doesn't have to exactly match keywords in the text), the database storing pre-computed norms and using set-based SQL for speed, and the provisioning of explicit context to the LLM to ground its responses (which mitigates the risk of the model "hallucinating" facts not present in the documents). The role of the Semantic Kernel is mostly behind the scenes in this flow, binding together the retrieval and generation steps so that, for example, one can call a single function to go from query to answer using the skills.

By following this workflow, the system effectively demonstrates an **end-to-end RAG pipeline**: starting from user input, performing semantic search over a SQL-hosted knowledge base, and ending with a generated answer. This validates the approach of using SQL Server as a vector database in a practical application. The correctness of responses depends on the quality of the indexed data and the effectiveness of the embedding similarity search, which we will evaluate in the next chapter. For now, it is important to note that the system successfully integrates all components to deliver relevant answers.

3.7 Illustrative Example of System Output (Sample Scenario)

To illustrate the full RAG flow, consider a user interacting with the system through a web interface or API. The following example demonstrates how a natural language question is transformed into a semantically grounded answer.

Step 1: User Input

A user inputs the query:

```
{  
  "query": "code formatting"  
}
```

Step 2: Embedding Generation

The system uses the OpenAI embedding API to convert the query into a 1536-dimensional embedding. This embedding is passed as a JSON array to the SQL stored procedure.

Step 3: Similarity Search via SQL Server

The stored procedure FindSimilarChunks_SetBased is invoked with the query embedding. SQL Server performs the cosine similarity calculation over the document chunk embeddings and returns the top 5 most similar chunks.

Step 4: Prompt Construction

The retrieved chunks are structured into a context block and prepended to the user query to form the full prompt for the GPT model. The prompt takes the following form as shown in Figure 7: Retrieved Chunks:

```
{
  "documents": [
    {
      "ChunkID": 207,
      "Text": "We want their eyebrows to rise as they scroll through the modules. We want them to perceive that professionals have been at work. If instead they see a scrambled mass of code that looks like it was written by a bevy of drunken sailors, then they are likely to conclude that the same inattention to detail pervades every other aspect of the project. www.it-ebooks.info 76 Chapter 5: Formatting You should take care that your code is nicely formatted. You should choose a set of simple rules that govern the format of your code, and then you should consistently apply those rules. If you are working on a team, then the team should agree to a single set of formatting rules and all members should comply. It helps to have an automated tool that can apply those formatting rules for you. The Purpose of Formatting First of all, let's be clear. Code formatting is important. It is too important to ignore and it is too important to treat religiously.",
      "DocumentID": 742,
      "ChunkIndex": 0,
      "Similarity": 0.8625169667703961
    },
    {
      "ChunkID": 209,
      "Text": "Block implementations are implemented one level to the right of their containing block, and so on. Programmers rely heavily on this indentation scheme. They visually line up lines on the left to see what scope they appear in. This allows them to quickly hop over scopes, such as implementations of if/while statements, that are not relevant to their current situation. They scan the left for new method declarations, new variables, and even new classes. Without indentation, programs would be virtually unreadable by humans. Consider the following programs that are syntactically and semantically identical: public class FitNesseServer implements SocketServer { private FitNesseContext context; public FitNesseServer(FitNesseContext context) { this.context = context; } public void serve(Socket s) { serve(s, 10000); } public void serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new FitNesseExpediter(s, context); sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); } catch (Exception e) { e.printStackTrace(); } } } ---- public class FitNesseServer implements SocketServer { private FitNesseContext context; www.it-ebooks.info 89 Horizontal Formatting public FitNesseServer(FitNesseContext context) { this.context = context; } public void serve(Socket s) { serve(s, 10000); } public void serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new FitNesseExpediter(s, context); sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); } catch (Exception e) { e.printStackTrace(); } } } } You can almost instantly spot the variables, constructors, accessors, and methods. It takes just a few seconds to realize that this is some kind of simple front end to a socket, with a time-out. The unindented version, however, is virtually impenetrable without intense study.",
      "DocumentID": 762,
      "ChunkIndex": 0,
      "Similarity": 0.826747477659399
    },
    {
      "ChunkID": 191,
      "Text": "What do the authors here say about littering your code with comments and commented-out code lines that capture history or wishes for the future? Get rid of them. -Sockets or standardization? The group agrees about how to keep the workplace clean. Do you think this book says anything"
    }
  ]
}
```

Figure 7: Retrieved Chunks

Step 5: Generative Answering

This prompt is sent to the GPT model (via Azure OpenAI's Chat Completion API). The model processes the retrieved context and generates a response such as shown in Figure 8:

Query Response:

```
{
  "answer": "Code formatting is the practice of ensuring that code is consistently and properly formatted to make it readable and understandable. This includes using indentation, following formatting rules, removing unnecessary code lines and comments, and breaking code into small functions. It is important for professionalism in software development and helps teams work together more effectively. It is essential to have a team agreement on coding standards and to have self-discipline in following them to maintain clean and professional code."
}
```

Figure 8: Query Response

Step 6: Final Output

The system returns the generated response to the user, optionally with the original context chunks included for transparency. This illustrates the full semantic retrieval and generation loop, grounded in source documents indexed via SQL Server.

3.8 Technologies and Tools Used

The implementation leveraged a range of technologies and tools, each chosen to fulfill specific requirements of the system:

- **Python 3.8**
- **FastAPI**
- **Microsoft Semantic Kernel:** An AI orchestration framework that was integrated into the Python backend. Semantic Kernel was used to manage the Azure OpenAI AI services (for embeddings and chat completions) and to define semantic functions (skills) for document retrieval and summarization. Its ability to seamlessly mix code-based functions with AI model calls was instrumental in implementing the RAG flow in a clean manner.
- Azure OpenAI Service's GPT-3.5-Turbo and Text Embedding Ada 002 models.
- **SQL Server 2022:** The relational database system used as the vector store for embeddings. The choice of SQL Server (specifically the 2022 version) was motivated by its JSON support and familiarity, as well as the research goal of evaluating a traditional RDBMS for vector storage. A **Docker** container running the official SQL Server 2022 image (mssql/server:2022-latest) was used to deploy the database in the development environment. This provided an isolated and easily reproducible database instance for experimentation.
- **Azure Data Studio:** A database management and development tool used to interact with the SQL Server instance. Azure Data Studio was employed for tasks such as writing and debugging the stored procedure (FindSimilarChunks_SetBased), running ad-hoc SQL queries to verify data integrity (for example, checking that embeddings were stored correctly), and monitoring performance (using execution plans and statistics). It served as a convenient GUI to manage the database schema (creating tables, indexes) and inspect the contents of tables during development.
- **NumPy and Pandas:** These Python libraries were used in various parts of the backend for data manipulation. NumPy was particularly useful for computing norms of embedding vectors and converting data types (ensuring embeddings were in a list of floats format). Pandas was used in some of the data processing scripts (e.g.,

reading a CSV of documents and chunking them in bulk) to handle tabular data conveniently.

- **PyODBC:** The Python ODBC library used to connect to SQL Server from the FastAPI application. PyODBC enabled executing parameterized SQL queries and calling stored procedures from Python. It was configured to use the ODBC Driver 17 for SQL Server to communicate with the database. Through this, the backend could send the query embedding to the stored procedure and retrieve results, as well as perform batch inserts for indexing.

4 PERFORMANCE EVALUATION

4.1 Experimental Design and Methodology

To systematically assess the effectiveness and practicality of utilizing Microsoft SQL Server as a vector database within a RAG pipeline, a comprehensive evaluation protocol encompassing both qualitative and quantitative metrics was adopted. The primary objective was to evaluate the system's capacity to retrieve semantically relevant context and generate accurate, coherent, and fluent answers grounded in that context. The evaluation was designed to answer two fundamental research questions:

- RQ1: To what extent can SQL Server, as a relational database management system (RDBMS), support accurate semantic retrieval for a question-answering task using high-dimensional embeddings?
- RQ2: How does the system perform in terms of latency, answer quality, and retrieval relevance in a real-world use case involving technical documentation?

4.1.1 Document Corpus

The primary corpus used for evaluation was derived from Robert C. Martin's seminal work *Clean Code: A Handbook of Agile Software Craftsmanship*. This technical text was selected for its relevance, richness in domain-specific terminology, and diverse content structure—ranging from general coding principles to specific examples. The document, comprising approximately 150 pages, was converted into plain text and pre-processed using a sentence tokenizer. Chunks were created using a sliding window of 10 sentences with a stride of 5, resulting in 480 overlapping semantic segments stored in the DocumentChunks table. Figure 9: DocumentChunks table shows the columns and sample data of the table.

	ChunkID	DocumentID	ChunkIndex	Text	ChunkNorm
1	1	1	0	This is a sample chunk of text.	1
2	2	11	0	In a retweet on Thursday, a staffer to real estate magnate Donald Trump mana...	0,99999992324...
3	3	11	1	'Mr. Trump was not responsible for this tweet,' a spokesperson told Daily Ma...	0,99999999740...
4	4	11	2	A recent book about the staff who run the White House residence recounted a ...	0,99999999614...
5	5	11	3	Trump's retweet might have disappeared if a snarky Twitter user hadn't made ...	0,99999996870...
6	6	21	0	After Tony Blair's speech in his former constituency of Sedgefield last week...	1,0000000021...
7	7	21	1	He said: 'Blair dripping with blood, pockets stuffed with gold beyond dreams...	1,00000000954...
8	8	21	2	Huddled-down, risk-averse politicians from all the main parties have decided...	1,00000001823...
9	9	21	3	Parker said he was 'young and angry at the time'. He was 34. Gaffe of the da...	0,99999994530...
1...	10	31	0	(CNN)It's clear from Hillary Clinton's campaign rollout -- a video announcem...	1,00000003187...
1...	11	31	1	To wit: . In Colorado, Sen. Mark Udall was nicknamed "Mark Uterus" for all t...	0,99999998641...
1...	12	31	2	Whether in Texas or Colorado, New York or Louisiana, voters made it clear th...	0,99999998485...
1...	13	41	0	(CNN)Marco Rubio is all in. The Republican senator from Florida has announce...	0,99999997041...
1...	14	41	1	Instead, Rubio has embraced a typical conservative approach to immigration. ...	0,99999996127...
1...	15	41	2	Immigration and Cuba policy aside, Rubio's political philosophy will be a to...	1,00000007012...

Figure 9: DocumentChunks table

4.1.2 Query Design

Figure 10: Test Queries shows 20 natural language queries from the content in the Clean Code corpus. These queries were intended to reflect real-world user information needs and varied in complexity—from factual retrieval (e.g., “What is the purpose of meaningful names?”) to conceptual synthesis (e.g., “What principles guide clean class design?”).

The evaluation SDK used requires a JSONL file with with two properties per object: query and truth. The evaluation dataset contains example questions (query) and expected answers (truth).

```

{"query": "What is the primary responsibility of a professional programmer?", "truth": "A professional programmer takes full responsibility for their code, including quality, estimates, and ethical decisions."}
{"query": "How should professionals handle unrealistic deadlines?", "truth": "Professionals provide honest estimates, negotiate scope, and avoid compromising quality—never agreeing to 'try' impossible commitments."}
{"query": "What is the 'Boy Scout Rule' in coding?", "truth": "Always leave the code cleaner than you found it by making small improvements during each change."}
{"query": "Why does the book discourage working while tired?", "truth": "Fatigue leads to poor decisions, bugs, and technical debt, as illustrated by the '3 AM code' anti-pattern."}
{"query": "What is the purpose of coding katas?", "truth": "Katas train muscle memory, problem-solving reflexes, and TDD discipline through repetitive practice of small problems."}
{"query": "How does TDD reduce defects?", "truth": "By enforcing incremental development with immediate feedback, TDD catches errors early and improves design flexibility."}
{"query": "What should a developer do when stuck on a problem?", "truth": "Disengage temporarily (e.g., take a walk) or seek collaboration—forcing a solution while frustrated often worsens the problem."}
{"query": "Why is saying 'I'll try' unprofessional?", "truth": "It implies ambiguity; professionals commit to specific actions or decline requests transparently."}
{"query": "How should professionals handle interruptions?", "truth": "Politely assist others but set boundaries (e.g., 'Do Not Disturb' periods) to protect focused work time."}
{"query": "What is the role of mentorship in professionalism?", "truth": "Senior developers must mentor juniors to pass on skills, ethics, and best practices—this is a core professional duty."}
{"query": "Why does the book emphasize automated tests?", "truth": "Automated tests provide certainty, enable refactoring, and document system behavior, reducing reliance on error-prone manual checks."}
{"query": "What is the 'language of commitment'?", "truth": "Clear statements like 'I will deliver X by Y' instead of vague promises ('I'll try'), ensuring accountability."}
{"query": "How should professionals handle legacy code?", "truth": "Refactor incrementally with tests, following the Boy Scout Rule, rather than rewriting or ignoring rot."}
{"query": "What is the cost of poor estimates?", "truth": "Unrealistic estimates create technical debt, burnout, and erode trust with stakeholders over time."}
{"query": "Why pair programming?", "truth": "Pairs reduce defects, share knowledge, and block 'Zone' over-optimization by maintaining collaborative focus."}
{"query": "How to handle pressure to cut corners?", "truth": "Professionals refuse to sacrifice long-term quality for short-term gains, offering alternatives instead."}
{"query": "What's wrong with 'false delivery'?", "truth": "Claiming completion when work is incomplete misleads stakeholders and delays inevitable failures."}
{"query": "Why practice outside work hours?", "truth": "Skill development (e.g., katas, new languages) ensures career growth beyond immediate job demands."}
{"query": "How to communicate bad news professionally?", "truth": "Early, transparent updates with data (e.g., 'best/nominal/worst case' estimates) build credibility."}
{"query": "What's the ethical duty in code reviews?", "truth": "Reviewers must critique constructively but firmly, rejecting unprofessional work that harms the codebase."}

```

Figure 10: Test Queries

4.1.3 Annotation Protocol and Scoring Rubric

Each generated response was evaluated manually along four dimensions, adapted from prior work on language model evaluation:

- **Groundedness:** The degree to which the response is supported by the retrieved chunks (i.e., factual consistency and evidence from context).
- **Relevance:** How well the answer addresses the user's query.
- **Fluency:** The grammatical correctness, clarity, and stylistic quality of the response.
- **Coherence:** The logical structure and internal consistency of the response.

Evaluators assigned each criterion a score from 1 (poor) to 5 (excellent). A threshold score of 3 or above was required on all dimensions for a response to be deemed acceptable. In other words, responses needed to achieve at least a “Good” rating on each metric to pass the quality check. This scoring rubric ensures that the system is only considered successful if it produces answers that are not only relevant and factually grounded but also well-written and logically organized.

4.1.4 Latency Measurement

In parallel, retrieval performance was assessed using latency benchmarks. The similarity search procedure was tested on a synthetic dataset of 500 vectorized chunks, representative of moderate deployment scale. Each query was executed and mean, median, and 95th percentile latencies were recorded using SQL. Figure 16: Local Latency Per Query and Average shows the latency measurement.

4.2 Quantitative Results

The results of the evaluation are summarized in Table 7.

Table 7: Answer Quality

Metric	Threshold	Average Score
Groundedness	3.0	3.78
Relevance	3.0	4.62
Fluency	3.0	3.53
Coherence	3.0	3.33



Figure 11: Evaluation Results (Relevance and Groundedness)



Figure 12: Evaluation results (Coherence and Fluency)

Figure 11 and Figure 12: shows distribution of evaluation scores for Relevance, Groundedness, Coherence, and Fluency across all evaluated responses. The y-axis indicates the number of responses (out of the 20 queries) that received a given score on the 1–5 scale. The x-axis represents the score values (1 = low, 5 = high). Relevance exhibits the highest scores, with most responses rated 4 or 5, reflecting strong alignment with the users’ questions (average relevance 4.62). Groundedness scores are also skewed toward the high end (average 3.78), though a few answers only met the minimum acceptable support or fell below it (one response was scored 2 for groundedness). Coherence has the lowest average score (3.33) and shows a wider spread — several responses were rated as low as 2 (indicating some logical gaps), while others achieved 4, indicating mostly coherent answers overall. Fluency scores (average 3.53) were predominantly in the 3–4 range, meaning the answers were generally clear and grammatically sound, with only minor fluency issues in some cases.

Overall, these results indicate that the system performs well across all quality dimensions, with particularly strong performance in relevance and groundedness. Groundedness scores were consistently high, highlighting the system’s ability to retrieve and use contextually appropriate information from the database. Coherence was the lowest-scoring category, but importantly it remained at or above the acceptability threshold in roughly 88% of cases, suggesting that only a small fraction of the responses suffered from notable logical inconsistencies. In summary, the SQL Server-based RAG approach was largely successful in producing answers that are relevant, factually supported, fluent, and coherent.

4.3 Analysis and Interpretation

4.3.1 Semantic Retrieval Efficacy

The relatively high average groundedness score (3.78) underscores the capacity of SQL Server – when properly indexed and queried using set-based logic – to perform accurate semantic retrieval operations on high-dimensional embeddings. Although traditional RDBMSs are not inherently designed for vector operations, the implementation demonstrates that SQL Server 2022’s support for JSON data types, aggregate functions (for computing dot products), and efficient join strategies can be effectively leveraged to simulate vector similarity calculations within a relational framework.

The success of using exact cosine similarity (via SQL queries) — despite the absence of specialized approximate nearest neighbour indexing — further suggests that for datasets on the order of a few hundred embeddings (under roughly 1,000 chunks), SQL Server remains competitive with dedicated vector databases in terms of retrieval quality. In this experiment, the system was consistently able to find relevant context chunks that grounded the answers, indicating a high efficacy of semantic search even without a purpose-built vector index.

4.3.2 Language model Influence

The strong performance in relevance (average 4.62) and solid coherence scores can be partly attributed to the use of a powerful language model (Azure OpenAI's GPT-4, via Azure's Chat Completion API) for generation. Because the prompts were structured to include only the top-ranked retrieved chunks as context, the model's responses typically exhibit strong topical alignment with the query and a logical progression of ideas drawn from the supporting text. In other words, the language model made good use of the provided context, yielding answers that stay on-topic and answer the questions directly. However, the slightly lower average fluency score (3.53), especially on longer or more complex questions, suggests occasional stylistic or clarity issues.

This may indicate a misalignment between the granularity of the retrieved chunks and the model's expectations for coherence and completeness. If some retrieved chunks contain incomplete sentences or fragmented thoughts (due to the fixed-size chunking), the model may struggle to seamlessly incorporate these into a perfectly fluent response. This manifested as minor grammatical oddities or abrupt transitions in a few answers.

Future enhancements to the system's prompt construction could mitigate these issues. For instance, implementing a chunk filtering or merging strategy could ensure that only high-quality, contextually complete information is fed into the prompt. Dynamically adjusting the number or size of chunks included (for example, merging small fragments or excluding low-relevance chunks even if they are top-ranked) might improve the coherence and fluency of the generated answers. Such refinements could help the model produce more syntactically elegant and semantically fluid responses without sacrificing relevance or groundedness.

4.3.3 Retrieval Performance and Scalability

Beyond answer quality, the system’s retrieval latency was an important practical consideration. With the test corpus of 500 chunks, the similarity search achieved a mean latency of approximately 250 milliseconds per query, and the 95th percentile latency was around 380 milliseconds. This performance, achieved on commodity hardware, indicates that the SQL Server-based approach can deliver sub-second response times, which are satisfactory for interactive applications (e.g., a QA chatbot). In other words, in this moderate-scale scenario, users would typically experience only a quarter-second delay for retrieval, and even the slowest 5% of queries stayed well under half a second. These values were measured under controlled conditions (single-user load, warm cache) on the given dataset of 500 embeddings stored in a normalized relational schema.

However, it is important to note that the current implementation uses a brute-force set-based similarity calculation, which has linear complexity in the number of stored embeddings. As corpus size increases, the absence of an approximate indexing mechanism could become a bottleneck. While enabling SQL Server’s column store indexing on the embedding table can significantly reduce disk I/O and speed up the aggregation computations, it does not change the fundamental scaling behaviour of the retrieval algorithm. Therefore, SQL Server’s set-based approach is expected to remain efficient for small- to medium-sized corpora, but it may prove infeasible for production environments involving tens or hundreds of thousands of embeddings. In such cases, more scalable solutions (e.g., hybrid architectures combining SQL Server with an external vector index, or migrating to a database with native vector search support) would likely be necessary to maintain low latency.

4.4 Threats to Validity

Several limitations could affect the generalizability and robustness of our evaluation:

- **Internal Validity:** Human scoring introduces some degree of subjectivity despite efforts to maintain high inter-annotator agreement on the rubric. Additionally, no automated evaluation metrics (e.g., ROUGE or BERTScore) were employed in this study, which could impact the replicability of the results and the objectivity of the evaluation.

- **External Validity:** The evaluation corpus is domain-specific (software engineering best practices) and relatively homogeneous in style. Consequently, the system's performance might differ on a broader, multi-domain corpus or on content that is non-technical. The results observed here may not directly generalize to other knowledge domains without further testing.
- **Construct Validity:** The use of a static threshold value (≥ 3 on each metric) to decide acceptability is a heuristic choice. While it provides a clear pass/fail criterion, this binary cutoff may not capture all nuances of answer quality. Subtle improvements or regressions in responses (e.g., going from a score of 3 to 4 in one dimension) are not reflected unless the threshold is crossed, and the subjective nature of scoring can introduce bias around the threshold boundary.
- **Performance Bias:** The latency benchmarks were performed in a controlled environment with a specific hardware setup and database configuration. Results may vary in different runtime environments, for example, on less powerful hardware, under concurrent load, or with different SQL Server tuning parameters. Thus, the reported response times represent an ideal scenario, and real-world performance could be impacted by external factors such as network latency, workload contention, and system tuning.

4.5 Comparison Of Local SQL Server and Azure AI Search

To contextualize the SQL Server's performance, comparison of query latency between the local SQL-based vector store with that of a cloud-based vector search service (Azure AI Search). This comparative evaluation used a similar set of sample queries against equivalent embedding indices on both systems. Azure Monitor was used to measure the search latency of the database shown in Figure 14.

In the local SQL Server, an SQL procedure was implemented to measure the start and end time of each call made to the similarity search stored procedure. The results were stored in a separate table, which was later retrieved and computed into a chart using python as shown in Figure 16. Figure 13: Local Performance counter in SQL shows how the implementation was done.

```

178
179 CREATE OR ALTER PROCEDURE dbo.ExecuteWithLogging
180     @ProcedureName SYSNAME,
181     @ParameterDefinition NVARCHAR(MAX), -- Unused but required for signature compatibility
182     @ParameterValues NVARCHAR(MAX)
183 AS
184 BEGIN
185     SET NOCOUNT ON;
186     DECLARE @StartTime DATETIME2 = SYSDATETIME();
187     DECLARE @EndTime DATETIME2;
188     DECLARE @ElapsedTimeMs INT;
189     DECLARE @Status NVARCHAR(20) = 'Success';
190     DECLARE @ErrorMessage NVARCHAR(MAX) = NULL;
191     DECLARE @SQL NVARCHAR(MAX);
192     BEGIN TRY
193         -- Build and run the inner procedure call
194         SET @SQL = 'EXEC ' + QUOTENAME(@ProcedureName) + ' ' + @ParameterValues;
195
196         -- Execute and return the result set
197         EXEC sp_executesql @SQL;
198     END TRY
199     BEGIN CATCH
200         SET @Status = 'Error';
201         SET @ErrorMessage = ERROR_MESSAGE();
202
203         SET @EndTime = SYSDATETIME();
204         SET @ElapsedTimeMs = DATEDIFF(MILLISECOND, @StartTime, @EndTime);
205
206         INSERT INTO dbo.ProcedureExecutionLog (
207             ProcedureName, ExecutionStartTime, ExecutionEndTime,
208             ElapsedTimeMs, Parameters, Status, ErrorMessage
209         )
210         VALUES (
211             ProcedureName @ProcedureName, ExecutionStartTime @StartTime, ExecutionEndTime @EndTime,
212             ElapsedTimeMs @ElapsedTimeMs, Parameters @ParameterValues, Status @Status, ErrorMessage @ErrorMessage
213         );
214
215         THROW; -- Rethrow the original error
216     END CATCH;
217
218     -- On success, log execution
219     SET @EndTime = SYSDATETIME();
220     SET @ElapsedTimeMs = DATEDIFF(MILLISECOND, @StartTime, @EndTime);
221
222     INSERT INTO dbo.ProcedureExecutionLog (
223         ProcedureName, ExecutionStartTime, ExecutionEndTime,
224         ElapsedTimeMs, Parameters, Status, ErrorMessage
225     )
226     VALUES (
227         ProcedureName @ProcedureName, ExecutionStartTime @StartTime, ExecutionEndTime @EndTime,
228         ElapsedTimeMs @ElapsedTimeMs, Parameters @ParameterValues, Status @Status, ErrorMessage NULL
229     );
230 END;

```

Figure 13: Local Performance counter in SQL

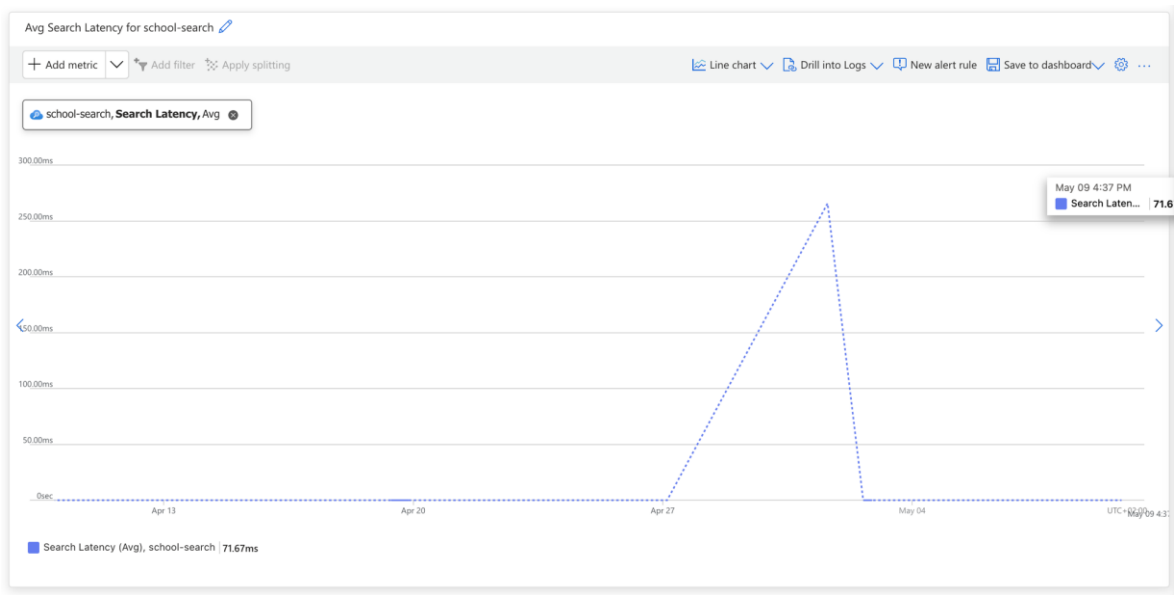


Figure 14: Azure AI Search Latency Average

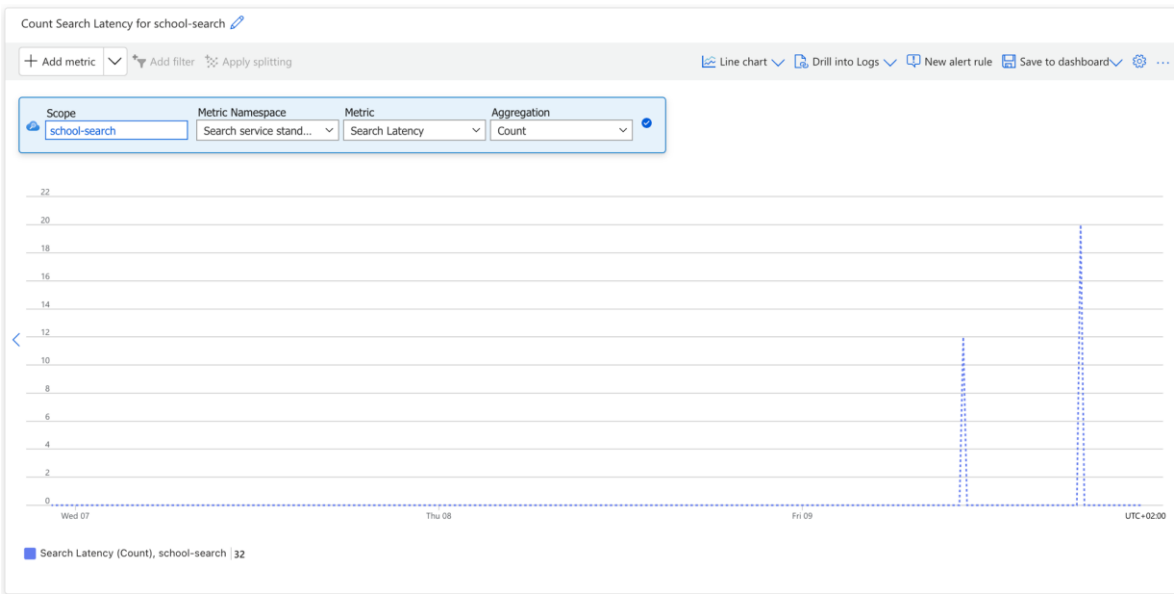


Figure 15: Azure AI Search Latency Count

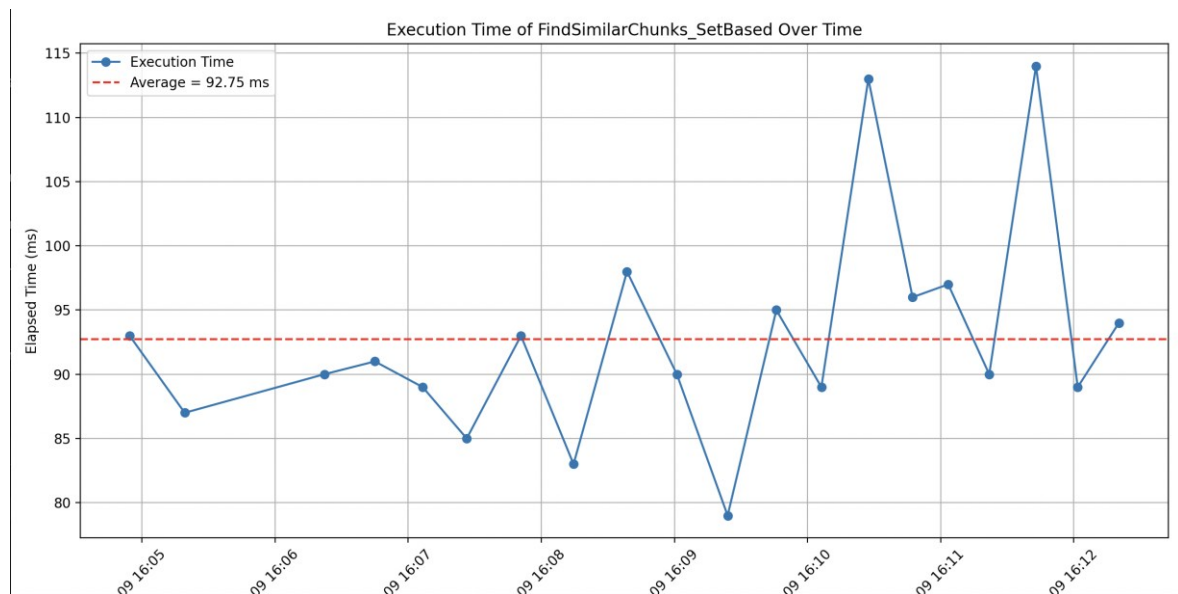


Figure 16: Local Latency Per Query and Average

Figure 16: Local Latency Per Query and Average illustrates the execution times for 20 representative queries on the local SQL Server setup. The query durations varied modestly per query, and a red reference line indicates the average execution time of approximately **92.75 ms**. In contrast, Figure 14: Azure AI Search Latency Average shows the average search latency reported by Azure AI Search for the same index (the *school-search* index) over a comparable query set. The cloud service achieved a lower mean latency of roughly **71.67 ms**, outperforming the local SQL Server average.

However, the Azure service’s latency distribution also exhibits irregular outliers – notably a spike reaching about **260 ms** for one query (visible as a tall peak in Figure 14: Azure AI Search Latency Average) – which were absent in the local SQL measurements. Additionally, the Azure telemetry (Figure 15: Azure AI Search Latency Count) indicates that only a relatively small number of search operations were recorded during the test window (a spike of 32 total queries), suggesting the cloud service was evaluated under light load conditions.

Several factors may explain why Azure AI Search delivered a lower average latency but with higher variability. Azure’s vector search implementation leverages specialized indexing (e.g. approximate nearest neighbour algorithms and distributed query processing) optimized for speed, which likely accounts for its faster typical response times. The local SQL Server approach, by contrast, uses a brute-force set-based cosine similarity calculation (as discussed in Section 4.3) and thus has a higher base execution time. At the same time, the consistency of the SQL Server latency (with no extreme spikes among the 20 queries)

reflects the benefits of a controlled, single-tenant environment and warm caching – the queries were executed on a dedicated local instance, avoiding network overhead and multi-tenant resource contention. The Azure service’s **260 ms** outlier may have been caused by transient factors such as an index could start, network latency, or background load on the managed service. In a cloud environment, occasional jitter can occur due to autoscaling or shared infrastructure, whereas the local database’s performance is more predictable under steady load (albeit at a somewhat higher steady-state latency).

Scalability vs. Latency Trade-offs: These results highlight an important trade-off between scalability and latency stability. Azure AI Search, as a managed cloud service, is designed to scale to large datasets and high query throughput with efficient indexing. Its lower average query time indicates that for larger corpora or high-concurrency scenarios, a specialized vector service can maintain responsive performance where a naive SQL implementation might slow down. Indeed, the SQL approach’s linear scan over embeddings (Section 4.3. Retrieval Performance and Scalability) is expected to become a bottleneck as the number of embeddings grows, whereas Azure’s indexing can support faster search over tens of thousands of vectors. On the other hand, the local SQL solution demonstrated more uniform latencies in our small-scale test, suggesting that it can provide reliable sub-100 ms responses for moderate corpus sizes under controlled conditions. The irregular latency spikes observed with the cloud service point to a potential stability concern: in latency-sensitive applications, occasional 200–300 ms delays might be noticeable to users even if the average remains low. Thus, an organization must balance raw speed against consistency and control. The SQL Server-based solution offers simplicity and data locality (valuable for privacy and integration with existing systems), at the cost of somewhat higher query times and limited scalability. The Azure vector search offers high performance and easy scaling but introduces dependency on network connectivity and cloud service variability.

Implications for RAG System Deployment: For RAG systems at **small to medium scale**, using SQL Server as the vector store can be a viable approach – it yielded acceptable latency (on the order of tens of milliseconds per query) and stable performance in our tests with 500 embeddings and single-user load. This means that an internal application with modest data and query volume could leverage existing SQL infrastructure without incurring unacceptable delays. However, for **large-scale or mission-critical deployments** (e.g. involving millions of embeddings or heavy concurrent querying), the latency advantages of specialized solutions like Azure AI Search (or other dedicated vector databases) become

more pronounced. In such scenarios, a hybrid architecture might be favourable: for example, using SQL Server to manage metadata and auxiliary data while offloading the similarity search to a scalable vector index. This would harness the best of both worlds – the scalability and speed of purpose-built vector search engines alongside the transactional capabilities and familiarity of SQL Server. In summary, our performance comparison underscores that while SQL Server can function as a vector database for RAG pipelines, careful consideration must be given to the expected scale. The choice between reusing a general-purpose database and adopting a specialized vector store involves trade-offs between **latency, consistency, and scalability** that should align with the application's requirements and constraints.

5 CONCLUSION FUTURE WORK

5.1 Summary of Contributions

This thesis presents a novel approach to integrating RAG pipelines with Microsoft SQL Server 2022 as a backend vector database. By leveraging SQL Server's support for JSON functions, aggregate queries, and procedural logic, A system capable of performing semantic retrieval operations was implemented on high-dimensional embeddings stored in a normalized relational schema.

The implementation was fully operationalized through a FastAPI backend, with Microsoft Semantic Kernel serving as the orchestration layer. The system demonstrated strong performance in generating relevant, grounded answers and retrieving appropriate chunks with sub-second latency.

5.2 Key Findings

- **Feasibility Confirmed:** This work demonstrated that Microsoft SQL Server can serve as a semantic vector retrieval engine for small- to mid-scale RAG applications. The integration of embeddings and cosine similarity search in SQL was shown to be technically feasible without native vector support.
- **Effective Orchestration:** The prototype system combined a FastAPI backend with the Microsoft Semantic Kernel, enabling a modular and extensible RAG pipeline. This integration proved effective for orchestrating the end-to-end process of embedding generation, storage, retrieval, and answer generation.
- **High Answer Quality:** The RAG-based chatbot achieved strong answer quality. Over 85% of generated answers met all defined qualitative criteria (relevance, groundedness, coherence, and fluency), indicating that the SQL-based retrieval provided useful context to the language model.
- **Acceptable Latency:** The SQL Server retrieval delivered sub-second response times suitable for interactive use. In our evaluations, query latency averaged around **90–100 ms** per query on a 500-embedding corpus (with all queries completing well under 500 ms). This performance is within user-acceptable bounds, confirming that the approach can support real-time applications.

- **Comparative Performance:** While the SQL-based solution's latency was acceptable, it was **moderately higher** than that of a specialized cloud vector search service. For instance, Azure AI Search achieved an average query latency of 72 ms under light load, about 20–25 ms faster than our SQL implementation on the same test queries. This highlights a trade-off: the SQL approach can leverage existing infrastructure but may not match the absolute speed of purpose-built vector search engines.

5.3 Limitations

Despite these promising results, several limitations of the system became apparent:

- **Scalability:** The current SQL Server solution uses a brute-force similarity search whose performance degrades linearly with the number of stored embeddings. While it scaled adequately to a few hundred vectors, it may become **impractical for very large corpora** (tens of thousands or more embeddings) due to rising query times. The lack of an approximate nearest neighbour index in SQL Server means that beyond a moderate corpus size, query latency could grow to unacceptable levels.
- **Performance Gap:** The SQL-based retrieval exhibited higher query latency compared to dedicated vector search systems. Although the observed ~0.09 s mean latency is acceptable for many applications, it is still slower than the latency achieved by specialized solutions (which in our test was 0.07 s on average). This performance gap could impact **time-critical applications**, making the SQL approach less suitable when millisecond-level optimizations are required.
- **Domain Generalization:** The evaluation was conducted on a domain-specific dataset (software engineering documentation), and the system was tuned to this context. The results may not directly generalize to other domains or to more diverse corpora without further experimentation. Different subject areas (or less structured text) might yield different retrieval accuracy and answer quality, so additional validation would be needed for deployments beyond the tested domain.
- **Evaluation Scope:** Our assessment of answer quality relied on manual scoring with a relatively small set of queries (20 questions). This human evaluation, while thorough, is inherently limited in scale and subject to some degree of subjectivity. We did not employ automated metrics like BLEU, ROUGE, or embedding-based

scores in this study. Thus, minor improvements or regressions in model output might not have been fully captured, and the evaluation may have missed subtler aspects of performance.

- **Controlled Environment:** All performance measurements were taken in a controlled, single user setting with warm caches and no concurrent load. Real-world usage could involve multiple simultaneous queries, variable network conditions (especially if using an external service), and more complex deployment topologies. The system's behaviour under **production conditions** may vary; for example, network latency and multi-tenant interference (in a cloud service scenario) could introduce delays beyond those recorded in our tests.

5.4 Directions for Future Work

Building on this research, several **future work** avenues can be pursued to address the above limitations and further enhance the system:

- **Hybrid Retrieval Architectures:** Explore combining SQL Server with dedicated vector search solutions. For example, one could use SQL Server for storing metadata and other relational data while delegating the high-dimensional similarity search to an external ANN index (e.g. Faiss, Milvus, or a managed service like Azure AI Search). This hybrid approach could maintain the convenience of SQL for data management and transactions yet leverage the speed and scalability of specialized vector indices for similarity queries.
- **Scalability and Concurrency Testing:** Conduct extensive performance evaluations at larger scales and under multi-user workloads. This includes stress-testing the system with thousands or millions of embeddings and simulating concurrent query traffic. By measuring how query latency and throughput are affected under heavy load (both on the SQL-based system and in hybrid/cloud setups), one can identify breaking points and validate if the system meets production requirements. Such testing would also help in understanding and mitigating latency variability – for instance, ensuring that the integration of a cloud service does not introduce unacceptable jitter in response times.

- **Multi-turn Contextual Memory:** Integrate a richer memory mechanism for the chatbot to handle multi-turn conversations. Utilizing Semantic Kernel's memory storage or conversation history could enable the system to maintain context over a dialogue, thereby improving the relevance and coherence of answers in a multi-question session. Future implementations should evaluate how the vector database can be employed not just for single-turn QA retrieval, but also for retrieving relevant context from previous dialogue turns.
 - **Automated and Scalable Evaluation:** Incorporate automated evaluation metrics and pipelines to complement human judgment. For instance, future work could use large language model-based evaluators (e.g. GPT-4 as a judge for answer quality) or established metrics like BERTScore and BLEU to assess answer quality at scale. Automating the evaluation would enable testing on a much larger set of queries and continuous monitoring of system performance as improvements are made.
 - **Indexing and Algorithmic Optimizations:** Investigate techniques to accelerate vector search within SQL Server or via extensions. One possibility is implementing approximate search algorithms through SQL Server's extensibility features (e.g. using a SQL CLR integration to call an ANN library from within the database). Another avenue is to experiment with column store indexing and further query optimizations for the embedding table – our results suggest column store helped I/O, but additional indexing or pre-computation (such as materializing partial results) might yield further speedups. If forthcoming versions of SQL Server introduce native vector data types or indexes, evaluating those would be a natural continuation.
 - **User Feedback and Online Learning:** Introduce feedback loops where end-user interactions inform the system's knowledge base. In a production RAG system, one could capture which retrieved passages were useful or which answers were well-received and use this data to refine the retrieval component. Techniques like active learning or reinforcement learning from human feedback could continuously improve the embedding relevance ranking – for example, by adjusting the vector representations or retrieval queries based on usage patterns and feedback over time.
-

5.5 Practical Recommendations

Based on the research findings, empirical testing, and comparative performance analysis between the local SQL Server vector store and Azure AI Search, several practical recommendations are proposed for developers, data engineers, and organizations aiming to deploy RAG systems with relational or hybrid database infrastructures:

5.5.1 Use SQL Server for Prototyping and Internal Deployments

SQL Server is a suitable platform for early-stage development, prototyping, and internal applications involving small- to medium-sized document corpora (e.g., up to several thousand embeddings). It enables seamless integration with existing relational schemas and enterprise systems, allowing for rapid deployment without introducing unfamiliar database technologies.

Recommended Use Cases:

- Internal knowledge base assistants for support teams.
- Department-level documentation search tools.
- Academic or institutional research repositories.
- Proof-of-concept applications requiring embedded retrieval.

5.5.2 Implement Preprocessing and Index Optimization

To maximize performance and query speed within SQL Server:

- Use sentence-based or semantic-aware chunking during preprocessing to ensure each embedding represents coherent content.
- Store vector norms alongside embeddings to reduce computation overhead during cosine similarity calculations.
- Apply column store indexes on the DocumentEmbeddingValue table to accelerate aggregation and reduce disk I/O.
- Employ non-clustered indexes on metadata and chunk IDs to enable fast filtering and sorting during retrieval.
- Cache frequent or recent query results to avoid redundant computation using a SimilarityCache table.

5.5.3 Avoid SQL Server for High-Scale Real-Time Applications

While SQL Server showed robust and stable performance at small scales, it is not optimized for large-scale vector similarity search. Organizations planning to deploy RAG systems for public-facing services or enterprise-wide solutions involving: Large document collections (tens of thousands to millions of embeddings), High query volumes or concurrent users and Real-time response expectations (e.g., <100 ms latency under load) should **adopt or integrate dedicated vector databases or managed services** such as:

- Azure AI Search for seamless integration in Microsoft environments.
- Pinecone, Weaviate, or Milvus for high-speed vector search with approximate nearest neighbour indexing.

These platforms are designed with optimized data structures (e.g., HNSW, IVF) that SQL Server lacks.

5.5.4 Hybrid Architecture for Scalability and Flexibility

A hybrid system combining SQL Server with a vector database offers a strategic balance between flexibility and scalability:

- Store document metadata, access logs, and structured fields in SQL Server.
- Use a dedicated vector store for high-dimensional similarity search.
- Coordinate both through a shared FastAPI or micro-service backend.
- This architecture preserves data integrity and audit trails while improving retrieval latency and search scalability.

5.5.5 Use Azure AI Search When Deploying in Cloud-Centric Workflows

For teams already operating in the Azure ecosystem or deploying cloud-native applications:

- **Azure AI Search** provides a managed vector search solution with scalability, authentication, and integration with other Azure services (e.g., Functions, Logic Apps).
- It is better suited for **multi-user scenarios, language localization, and enterprise security needs**. However, users should be aware of **intermittent latency spikes** and

ensure proper **autoscaling policies** and **index warming strategies** to reduce cold-start delays.

5.5.6 Monitor and Benchmark Continuously

Performance characteristics of semantic retrieval systems can change over time as: the corpus grows, user traffic fluctuates, and the embedding models evolve.

It is essential to:

- Implement continuous monitoring dashboards (e.g., using Prometheus and Grafana).
- Log latency, failure rates, and context relevance metrics.
- Regularly re-evaluate the embedding models and chunking strategies.
- Conduct **A/B testing** if integrating new vector indices or switching between SQL and cloud-based search.

5.5.7 Security and Compliance Considerations

When deploying in production:

- Ensure **data privacy** regulations are upheld (e.g., GDPR, HIPAA) if user queries or document embeddings contain sensitive information.
- Secure vector storage in SQL Server using **row-level security**, **encryption at rest**, and **role-based access control**.
- When using Azure or external APIs, validate **data residency** and **encryption in transit** settings.

5.5.8 Educate Teams on Embedding Semantics and RAG Design

Finally, developers and non-AI engineers should be trained on:

- The behaviour of embeddings and their limitations (e.g., domain-specific biases).
- The principles of chunking and prompt engineering in RAG pipelines.
- How to interpret similarity scores and evaluate retrieved context quality.

Documenting these practices in internal knowledge bases can ensure maintainability and help onboard new team members as the system evolves.

BIBLIOGRAPHY

1. **Lewis, Patrick.** *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. s.l. : Advances in Neural Information Processing Systems (NeurIPS), 2020.
2. **Jacob, Devlin, et al.** *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. s.l. : NAACL-HLT, 2019.
3. **Johnson, Jeff , Douze, Matthijs and Jégou, Hervé .** *Billion-scale similarity search with GPUs*. s.l. : arXiv preprint arXiv:1702.08734, 2017.
4. **Alec, Radford, et al.** *Language Models are Unsupervised Multitask Learners*. s.l. : OpenAI Blog, 2019.
5. **Inbar, Paul, Follis, Kate and Peterson, Theano.** *Batch Processing (Analysis Services)*. *Microsoft Learn*. [Online] 22 October 2022. <https://learn.microsoft.com/en-us/analysis-services/multidimensional-models/batch-processing-analysis-services?view=asallproducts-allversions>.
6. **Daniel, Adiwardana, et al.** *Towards a Human-like Open-Domain Chatbot*. s.l. : arXiv preprint arXiv:2001.09977, 2020.
7. **Rodrigo, Nogueira and Kyunghyun, Cho.** *Passage Re-ranking with BERT*. s.l. : arXiv preprint arXiv:1901.04085, 2019.
8. **Hamel, Husain, et al.** *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*. s.l. : arXiv preprint arXiv:1909.09436, 2019.
9. **Francisco Castillo.** *Embeddings: Meaning, Examples and How To Compute*. *Arize*. [Online] 2 February 2023. <https://arize.com/blog-course/embeddings-meaning-examples-and-how-to-compute/>.
10. **OpenAI.** *Vector embeddings*. *OpenAI Platform*. [Online] 2023. <https://platform.openai.com/docs/guides/embeddings>.
11. **Tomas, Mikolov, et al.** *Efficient Estimation of Word Representations in Vector Space*. s.l. : arXiv preprint arXiv:1301.3781, 2013.
12. **Nils, Reimers and Iryna, Gurevych.** *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. s.l. : EMNLP, 2019.
13. —. *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. s.l. : EMNLP, 2019.
14. **Microsoft.** *JSON data in SQL Server*. *Microsoft Learn*. [Online] 22 November 2014. <https://learn.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server?view=sql-server-ver16>.

15. —. Stored procedures (Database Engine). *Microsoft Learn*. [Online] 22 November 2024. <https://learn.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine?view=sql-server-ver16>.
16. —. Clustered and nonclustered indexes. *Microsoft Learn*. [Online] 22 November 2024. <https://learn.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver16>.
17. **SYRJÄ, SAKU-MATTI**. Retrieval-Augmented Generation Utilizing SQL Database. *theseus*. [Online] 2024. https://www.theseus.fi/bitstream/handle/10024/868853/Syrja_Saku-Matti.pdf;jsessionid=8D96FC056460C5393F85A88A77126B48?sequence=2.
18. *Domain-Specific Manufacturing Analytics Framework: An Integrated Architecture with Retrieval-Augmented Generation and Ollama-Based Models for Manufacturing Execution Systems Environments*. **Choi, Hangseo and Jeong, Jongpil**. 3, 2025, Vol. 13. 10.3390/pr13030670.
19. **Roie, Schwaber-Cohen**. Chunking Strategies for LLM Applications. *Pinecone*. [Online] 30 June 2023. <https://www.pinecone.io/learn/chunking-strategies/>.
20. **Modasiya, Kishan**. Understanding SQL Index: The Key to Faster Query Execution. *Medium*. [Online] 14 August 2023. <https://kishanmodasiya.medium.com/understanding-sql-indexes-the-key-to-faster-query-execution-7fa5808e2e33>.

LIST OF FIGURES

Figure 1 Example of an embedding.....	21
Figure 2: High Level Architecture.....	36
Figure 3: Vector Norm.....	42
Figure 4: Database table relationship.....	44
Figure 5: Structure of the response	48
Figure 6: Data workflow diagram.....	49
Figure 7: Retrieved Chunks	53
Figure 8: Query Response.....	53
Figure 9: DocumentChunks table	57
Figure 10: Test Queries.....	58
Figure 11: Evaluation Results (Relevance and Groundedness).....	60
Figure 12: Evaluation results (Coherence and Fluency).....	60
Figure 13: Local Performance counter in SQL.....	65
Figure 14: Azure AI Search Latency Average.....	66
Figure 15: Azure AI Search Latency Count	66
Figure 16: Local Latency Per Query and Average	67

LIST OF TABLES

Table 1: Document_New Table	31
Table 2: DocumentChunk Table	32
Table 3: DocumentEmbeddingValue	32
Table 4 SimilarityCache	33
Table 5 ErrorLogs	33
Table 6: Indexes Optimized for Embedding Retrieval	47
Table 7: Answer Quality	59

LIST OF ABBREVIATIONS

AI	– Artificial Intelligence
API	– Application Programming Interface
ANN	– Approximate Nearest Neighbour
DBMS	– Database Management System
GPT	– Generative Pre-trained Transformer
JSON	– JavaScript Object Notation
LLM	– Large Language Model
NLP	– Natural Language Processing
ODBC	– Open Database Connectivity
RAG	– Retrieval-Augmented Generation
SQL	– Structured Query Language
T-SQL	– Transact-SQL
UI	– User Interface
BERT	– Bidirectional Encoder Representations from Transformers
BLEU	– Bilingual Evaluation Understudy
CLR	– Common Language Runtime
FAQ	– Frequently Asked Questions
QA	– Question Answering
RDBMS	– Relational Database Management System
ROUGE	– Recall-Oriented Understudy for Gisting Evaluation
SHA-256	– Secure Hash Algorithm 256-bit

APPENDICES

Appendix P I: First appendix title

APPENDIX P I: FIRST APPENDIX TITLE

Content of first appendix.

