VIETNAM NATIONAL UNIVERSITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

REPORT
GRADUATION PROJECT (CO4337)

INTELLIGENT CODE ASSISTANCE FOR WEB DEVELOPMENT: RAG-ENHANCED LLM APPROACH

Major: Computer Science

Supervisors: Nguyen An Khuong, PhD
Pham Nhut Huy, Engineer - Zalo AI

Secretary:

Students: Nguyen Trang Sy Lam – Student ID: 2152715
Bui Ho Hai Dang – Student ID: 2153289
Le Hoang Phuc – Student ID: 2152239

Ho Chi Minh City, May 2025

Date:

PhD Nguyen An Khuong (Supervisor)

Declaration

We hereby declare that this research is entirely our own work, conducted under the supervision and guidance of Dr. Nguyen An Khuong and Mr. Pham Nhut Huy. The results presented in this research are legitimate and have not been published in any form prior to this. All materials used in this research have been collected by us from various sources and are properly cited in the bibliography section. Additionally, all figures included in this report have been created by us.

This research also references results from other authors, which have already been published by the respective authors.

In the event of any plagiarism, we take full responsibility for our actions. Ho Chi Minh City University of Technology - Vietnam National University is not liable for any copyright infringements arising from this research.

Ho Chi Minh City, May 2025

Authors

Acknowledgements

Firstly, we would like to sincerely express our gratitude to our supervisors, Dr. Nguyen An Khuong, and Mr. Pham Nhut Huy. To Dr. Nguyen An Khuong, thank you for dedicating your precious time to us. The weekly meetings helped our team stay on track, and your timely feedback, advice, and guidance greatly contributed to the success of our research. To Mr. Pham Nhut Huy, your expertise in the field, along with your thesis writing and presentation skills, was invaluable to our progress.

Secondly, we wish to extend our heartfelt thanks to our families for their unwavering support, both emotionally and financially, throughout this research process. Our accomplishments would not have been possible without their encouragement and assistance.

Lastly, this is from Nguyen Trang Sy Lam. I would like to personally thank Dr. Nguyen An Khuong for guiding me through the most challenging days of my student life. When I was in a dark place, your support and mentorship pulled me through and helped me complete this research. Words cannot fully convey my gratitude to you.

Abstract

In this project, we present the design of a Retrieval-Augmented Generation (RAG) workflow integrated into a Visual Studio Code (VS Code) extension, serving as a local intelligent assistant for web developers working with the Next.js framework. The system uses Milvus [53] as the vector database to efficiently store and retrieve semantically indexed information from the official Next.js documentation, prepared during the data processing phase.

For response generation, the architecture is model-agnostic and designed to work with locally hosted Large Language Models (LLMs) via Ollama[1], a lightweight framework that allows users to run various LLMs depending on their hardware capabilities. This flexibility empowers developers to choose the model that best suits their system—whether it's a smaller model for lightweight environments or a more powerful one for advanced tasks.

By combining retrieval from up-to-date documentation with locally generated, context-aware responses, the proposed RAG-enhanced assistant boosts productivity and simplifies development workflows directly within the VS Code environment.

[1]Ollama - `https://ollama.com/`

Contents

1 Introduction

1.1 Motivations

The process of web development has been significantly transformed by the introduction and adoption of frameworks. Frameworks have become the industry standard, providing developers with structured, reusable tools that simplify the creation of web applications. This evolution has also been greatly influenced by the rise of LLMs like OpenAI's Codex [9], Google's Gemini [50], and GitHub Copilot[1], which have made coding more accessible by offering automated code generation, real-time suggestions, and debugging assistance. These tools have made entry into web development easier than ever, lowering barriers for beginners and speeding up development processes.

However, despite these advancements, LLMs still face certain limitations, such as hallucinations, difficulties in referencing external sources, and performance issues when handling specific tasks. A promising solution to mitigate these limitations is using RAG. RAG enhances the generative capabilities of LLMs by retrieving relevant external information before generating content, improving both the accuracy and relevance of the output. This technique is particularly beneficial in domains like web development, where staying up-to-date with framework versions and ensuring compatibility across services is critical. Incorporating RAG helps bridge the gap between rapidly evolving frameworks and the knowledge developers need to make informed decisions about the tools and technologies they use.

[1] Github Copilot - `https://copilot.github.com/`

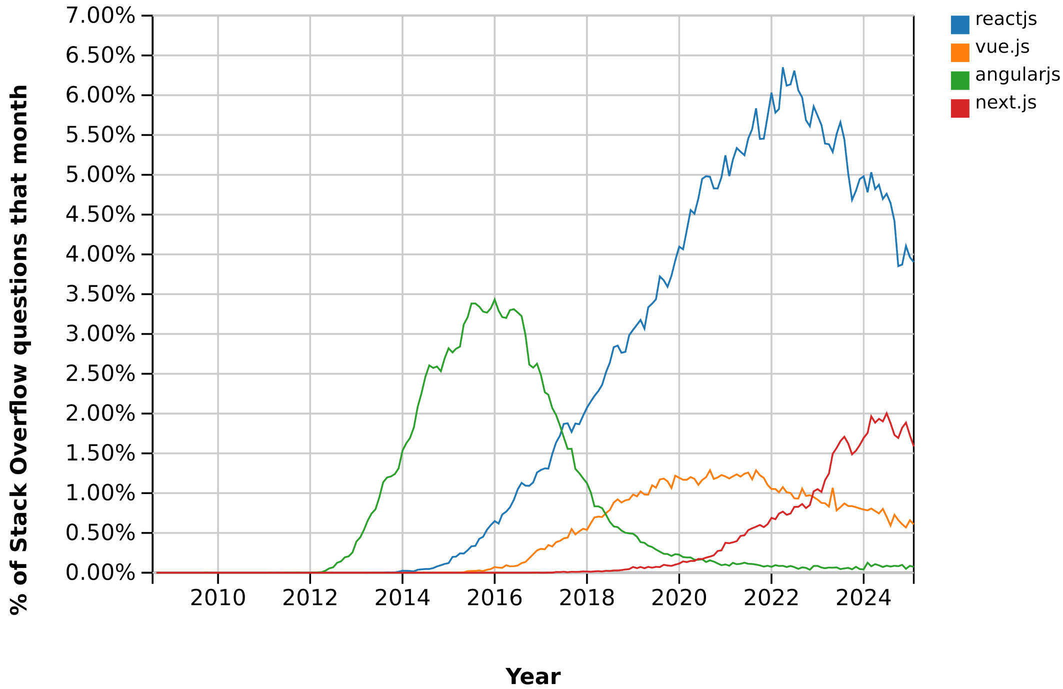

Figure 1.1: The popularity of frameworks based on StackOverflow questions 2008-2025

To illustrate, consider the Next.js framework, which exemplifies the rapid evolution of modern web technologies. Originally built on top of React.js, Next.js has transformed from a simple server-rendering tool into a comprehensive full-stack framework. What makes Next.js particularly notable is the speed at which it introduces significant updates. In just under three years, it moved from version 12 to version 15, each introducing major shifts in architecture and development paradigms.

For instance, Next.js 13 (October 2022) introduced the App Router, React Server Components, and a new file-based routing system—radically altering how pages and layouts are structured. Less than a year later, Next.js 14 (October 2023) refined these features while improving performance with deeper Turbopack integration. And most recently, Next.js 15 (March 2025) pushed the boundaries even further by stabilizing Partial Prerendering, extending Server Actions, and fully embracing React Server Components[2].

These frequent and impactful changes challenge developers to constantly relearn patterns, adapt to breaking changes, and stay updated with both Next.js and its dependency. React.js, which itself released React 19 with major improvements to transitions, refs, and concurrency[3].

1.2 Problem statement

Modern web development frameworks, such as Next.js, offer powerful features and flexibility, making them a popular choice for building high-performance applications. However, as these frameworks continue to evolve, their complexity has grown significantly. Developers are often required to refer to extensive documentation, which can be fragmented and difficult to navigate, to resolve coding challenges, learn new functionalities, or debug issues. This reliance on documentation creates several inefficiencies in the development workflow, including:

- Time-Consuming Information Retrieval: Developers frequently need to search through lengthy and intricate documentation to find specific information. Traditional search methods, such as keyword-based tools, often fail to capture the intent behind the query or retrieve relevant context, leading to prolonged and unproductive searches.

- Fragmented Context: Even when relevant information is found, it often lacks the necessary context to answer the developer's query fully. This forces developers to cross-reference multiple sections of

[2]Next.js 15 Release Notes - https://nextjs.org/blog/next-15
[3]React 19 Upgrade Guide - https://react.dev/blog/2024/04/25/react-19-upgrade-guide

the documentation, further delaying progress.

- Disruption of Workflow: Switching between the code editor and external documentation interrupts the development process, causing a loss of focus and reducing productivity.
- Inconsistent Results: Existing tools, such as search engines or static code assistants, often provide inconsistent results that fail to address the specific needs of a query or task, leaving developers to piece together partial solutions on their own.

1.3 Significant of the problem

Solving this problem is of critical importance for several reasons:

- Enhanced Developer Productivity: By significantly reducing the time spent searching for relevant information, the proposed system allows developers to focus on coding tasks, accelerating development timelines and increasing overall productivity.
- Improved Workflow Efficiency: Delivering context-aware assistance directly within the coding environment (e.g., Visual Studio Code) eliminates the need to switch between multiple tools and resources, streamlining the development process and reducing cognitive load.
- Support for Complex Frameworks: As frameworks like Next.js continue to evolve, the volume and complexity of their documentation will only increase. An intelligent system capable of efficiently managing and retrieving this knowledge ensures that developers can stay up-to-date and adapt to new features quickly.

By solving this problem, we aim to not only improve the efficiency of individual developers but also set a foundation for the future of Artificial Intelligent (AI), AI-assisted development environments. This research will demonstrate how advanced AI techniques like RAG can transform the way developers interact with documentation, enhancing both their productivity and their overall experience.

1.4 Objectives

The objective of this research is to develop a tool, specifically a VS Code extension, that facilitates the web development process by addressing common challenges associated with modern frameworks. Utilize RAG to assist developers with question answering, code generation, and code review. RAG will retrieve relevant information from documentation or community forums to enhance the accuracy and relevance of the tool's suggestions.

1.5 Scope

The scope is intentionally confined to Next.js as the primary framework for web application development, enabling a focused exploration of its interaction with RAG methodologies. Additionally, the implementation targets applications written in JavaScript, TypeScript, HTML, and CSS, ensuring compatibility with commonly used web development programming languages.

1.6 Challenges

Developing this tool involves several challenges:

- Efficient context management: Since the goal is to make the tool lightweight, storing and managing the context of a user's codebase in a way that allows for fast retrieval without overwhelming system resources is a major challenge.
- Data curation complexity: The data curation process requires extensive knowledge about the framework.
- Handling framework updates: Keeping the tool updated with the frequent changes in frameworks like Next.js and their associated packages is challenging. Continuous updates are needed to ensure compatibility and provide accurate suggestions.

- User experience: Balancing the amount of information presented to the user without overwhelming them is essential. The tool should provide meaningful insights and recommendations without causing information overload.

1.7 Structure of the report

Chapter 1. Introduction. This chapter introduces the motivations, objectives, and scope of the research. It highlights the challenges of web development and the role of RAG in enhancing efficiency and accuracy. The chapter outlines the structure of the report, providing a roadmap for the research presented.

Chapter 2. Preliminaries. This chapter provides foundational knowledge on RAG, emphasizing its retriever and generator components. It discusses indexing, vector embeddings, and augmentation methods, offering a clear understanding of the technologies used.

Chapter 3. Related works and tools. This chapter explores prior research on RAG and its application in web development. It reviews developer tools such as VS Code extensions and discusses strategies for retrieval and augmentation. Comparative analyses of existing frameworks and methodologies are also presented.

Chapter 4. Proposed solution. This chapter presents the reasoning behind adopting RAG over fine-tuning, and explains the selection of key components such as the BGE-M3 embedding model and Ollama's local LLMs. It also discusses design choices including containerization with Docker and the use of a VS Code extension. Finally, it outlines the system's core architecture, focusing on local vector storage and editor integration.

Chapter 5. Implementation. This chapter details the development process of the RAGGIN system, covering data preparation, preprocessing, database design and setup, and the implementation of a RESTful API. It also introduces the core retrieval and generation services, and concludes with the integration of these components into a VS Code extension.

Chapter 6. Evaluation. This chapter presents the evaluation of the system through quantitative metrics for both retrieval and generation components. It describes the evaluation methodology, summarizes results, and includes both unit and integration testing to validate system behavior. The analysis highlights key strengths as well as performance limitations under different usage scenarios.

Chapter 7. Conclusion. The final chapter summarizes the key results achieved, acknowledges the current system limitations, and outlines clear directions for future development.

2 Preliminaries

The RAG process combines the capabilities of retrieval systems and generative AI to deliver precise and context-aware responses. It begins with query vectorization, where the user's input is transformed into a mathematical vector using models like Bidirectional Encoder Representations from Transformers (BERT) [12] or sentence transformers. This vector is then used to search a pre-indexed vector database for semantically similar entries, retrieving the most relevant data. Once retrieved, the data undergoes preprocessing to ensure compatibility with the language model, such as summarizing or structuring it. The refined data is then integrated into the input context of a LLM, enabling it to generate responses that blend the retrieved information with its broader pre-trained knowledge. This structured pipeline ensures that the generated outputs are accurate, specific, and contextually grounded.

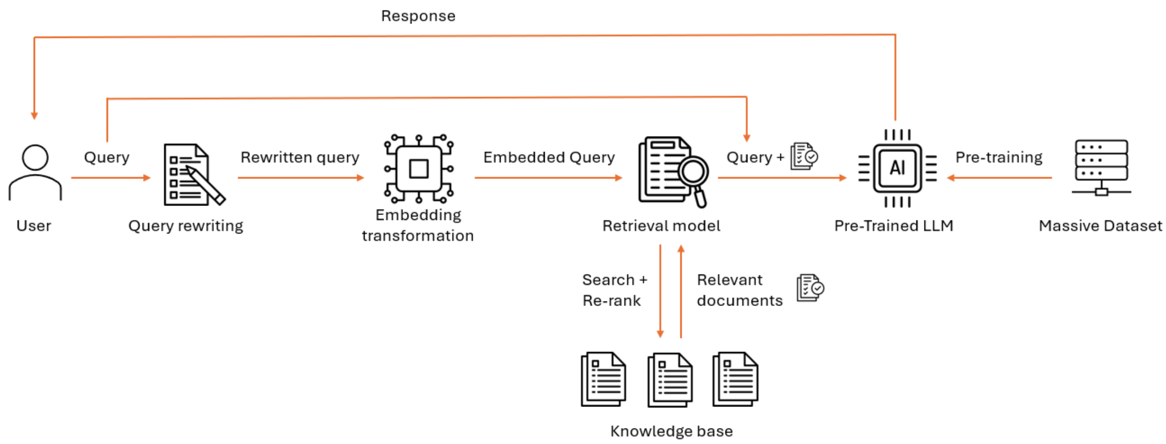

Figure 2.1: RAG Process

The RAG process is composed of two main components: the retriever and the generator. The retriever is responsible for locating and extracting relevant information from an external knowledge source, employing techniques such as keyword-based search, semantic similarity search, or neural network-based retrieval to ensure precise results. The generator then utilizes the retrieved context to produce coherent and accurate responses, ensuring the output aligns with the user's query while maintaining relevance and context. Together, these components form a robust framework for leveraging external knowledge in generative tasks.

2.1 Retriever

The retriever stage involves finding and gathering relevant text passages from an external knowledge source. This can be done through various methods such as keyword-based search, semantic similarity search, or neural network-based retrieval techniques. One major application of embedding models is neural retrieval. By measuring the semantic relationship with the text embeddings, allowing for the comparison of text based on content rather than just keyword matching, then the relevant answers to the input query can be retrieved based on the embedding similarity.

2.1.1 Preprocessing

Preprocessing starts with the cleaning and extraction of raw data in diverse formats like PDF, HTML, Word, and Markdown, which is then converted into a uniform plain text format. To accommodate the context limitations of language models, text is segmented into smaller, digestible chunks. Chunks are then encoded into vector representations using an embedding model and stored in a vector database. This step is crucial for enabling efficient similarity searches in the subsequent retrieval phase [15].

There are ways to optimize the indexing process to enhance the performance of the upcoming retrieval steps:

1. Chunking strategy: Documents can be splitted into chunks using various methods like fixed-size chunking (for structured documents and surveys), recursive chunking (for documents with large amounts of data), sentence-aware chunking (for analysis of textbooks) and sematic chunking (for information retrieval systems and legal document analysis) [26].

2. Metadata attachments: Each chunk can include extra information (page number, file name, author, category, or timestamp) as metadata. Afterward, the retrieval process can be refined by filtering results based on metadata, which helps narrow down the search. Assigning different weights to document timestamps during retrieval can achieve time-aware RAG, ensuring the freshness of knowledge and avoiding outdated information [15].

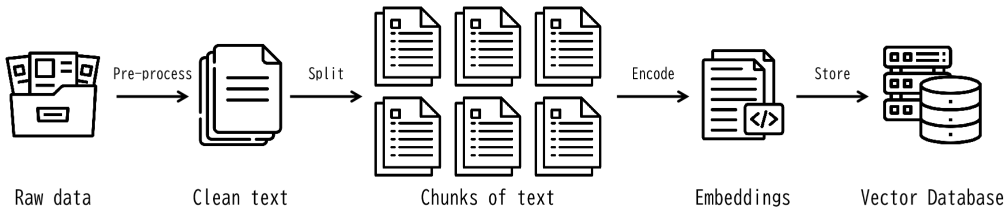

Figure 2.2: Preprocessing process for retriever

2.1.2 Embedding

Embeddings are a key component of RAG systems, which are used to effectively retrieve pertinent information from big datasets. By converting textual data into numerical representations, these embeddings are vector-based representations, typically generated by neural networks, that transform input text into a structured vector space. In this space, proximity reflects the semantic relationships of the input.

In essence, an embedding is a vector, a high-dimensional representation of a text in which every dimension represents a distinct facet of meaning. Consider a two-dimensional Cartesian coordinate system from the algebra class, but with a lot more dimensions (usually 768, 1024, or 1536) to better visualize this. Models such as BERT [12] and Sentence-BERT [41] employ this high-dimensional vector space to represent the semantic features of text. By calculating the distance between these vectors, embeddings enable the system to extract pertinent information from large datasets and choose the most contextually relevant text passages for response generation. This feature is essential for enhancing large language models' (LLMs') outputs' contextual relevance and accuracy.

Many embedding techniques have been used over the years:

1. **Word2Vec** [33]: A well-liked technique that uses the Continuous Bag of Words (CBOW) or Skip-gram models to learn word embeddings depending on local context. Word2Vec is effective at many jobs, but it has a major drawback: it has trouble handling polysemy, which is the situation where words have several meanings depending on the context. This is because Word2Vec creates a single representation for every word, independent of how it is used.

2. **BERT** [12]: A transformer-based model that generates contextual embeddings, considering both the preceding and following words in a sentence. This allows BERT to produce more accurate representations of words based on their surrounding context. It has been widely adopted for many NLP tasks such as question answering, sentiment analysis, and named entity recognition.

3. **Sentence-BERT** (SBERT) [41]: An extension of BERT, fine-tuned specifically for generating sentence embeddings. Unlike traditional BERT, which produces word-level embeddings, SBERT produces fixed-size embeddings for entire sentences. This allows for efficient comparison of sentence pairs, making SBERT highly effective for tasks like semantic textual similarity, paraphrase identification, and information retrieval.

4. **FAISS** [13] and **dense retrieval techniques**: In RAG systems, embeddings are often used with specialized retrieval tools like Facebook AI Similarity Search (FAISS) to perform fast, approximate nearest neighbor search. This helps locate the most relevant pieces of information from a large corpus, making the retrieval process both scalable and efficient.

Moreover, our embedding can be trained in different ways:

1. **Supervised learning:** In supervised learning, embeddings are trained with labeled data to predict specific outcomes, such as class labels or relationships. Examples include training embeddings for tasks like sentiment analysis or named entity recognition (NER).

2. **Unsupervised learning:** Unsupervised learning allows embeddings to be trained without labels, often by predicting words or context from surrounding text. Models like Word2Vec and global vectors for word representation (GloVe) [36] use this method to learn semantic relationships between words from large corpora.

Fine-tune embedding models: Reinforcement learning [27]: In reinforcement learning, embeddings are dynamically adjusted based on feedback from the environment to improve the system's performance over time. For instance, in a RAG system designed for domain-specific chatbots, reinforcement learning is employed to optimize the retrieval of relevant context. Using reinforcement learning with human feedback (RLHF) [27], models like GPT-4 are fine-tuned to respond accurately to domain-specific queries. A key example is optimizing FAQ retrieval for customer service chatbots. In such systems, reinforcement learning is used to decide whether to fetch additional context for each user query or rely on previously retrieved information. The system evaluates the quality of the generated responses (e.g., by GPT-4) and uses this evaluation as a reward signal. For example, if the system correctly answers a follow-up query without fetching redundant FAQ context, it receives a positive reward, thereby learning to minimize unnecessary retrievals. This approach not only improves efficiency by saving computational resources (such as API token usage) but also enhances accuracy by reducing noise in the generated outputs.

2.1.3 Hierarchical navigable small world (HNSW) indexing

1. **Vector database and HNSW:**

In RAG systems, vector database plays a pivotal role in efficiently retrieving relevant information to augment the capabilities of generative language models. These databases are designed to store and query high-dimensional embeddings, which are numerical representations of text, code, images, or other data. Embeddings capture semantic relationships between data points, enabling similarity searches based on meaning rather than exact matches. This makes vector databases a cornerstone of modern RAG pipelines, ensuring that retrieved context aligns with the query's intent. To achieve efficient and scalable retrieval from large embedding datasets, RAG systems often employ approximate nearest neighbor (ANN) algorithms, with HNSW graphs [32] being one of the most widely used techniques.

2. **HNSW algorithm:**

The HNSW algorithm is built on the concept of organizing links based on their length scales across multiple layers, enabling efficient search within a hierarchical, multilayer graph structure.

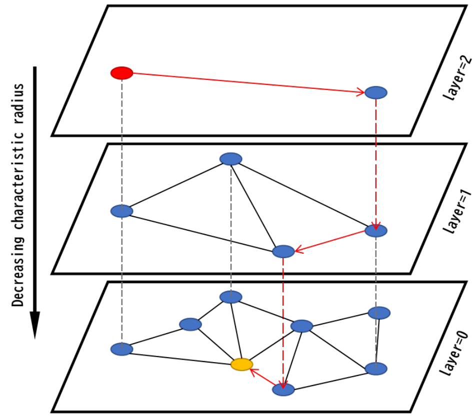

Figure 2.3: HNSW search algorithm visualization. The search starts from entry point from top layer (shown red), red arrows show direction of greedy algorithm to query point (shown yellow) [32]

The HNSW algorithm begins with a network construction phase (Algorithm 1), where the graph structure is incrementally built through the consecutive insertion of stored elements. For each new element, a maximum layer l is randomly determined using an exponentially decaying probability distribution. This hierarchical organization ensures that higher layers contain progressively fewer nodes, optimizing the search process by focusing on coarse-to-fine navigation.

The insertion process begins at the top layer of the hierarchical graph, where the algorithm performs a greedy traversal to locate the ef closest neighbors to the newly inserted element q within that layer. Once these nearest neighbors are identified, they serve as entry points for the next layer. This process is repeated layer by layer, moving downward through the graph, ensuring that the inserted element is effectively connected to its most relevant neighbors at each level.

Algorithm 1 INSERT(`hsnw`, q, M, $M_{\max}$, `efConstruction`, mL)

Input: multilayer graph `hsnw`, new element q, number of established connections M, maximum number of connections per element per layer $M_{\max}$, size of the dynamic candidate list `efConstruction`, normalization factor for level generation mL

Output: update `hsnw` inserting element q

$W \leftarrow \emptyset$ ▷ list for the currently found nearest elements
$ep \leftarrow$ get enter point for `hsnw`
$L \leftarrow$ level of ep ▷ top layer for `hsnw`
$l \leftarrow -\lfloor \ln(\text{unif}(0,1)) \cdot mL \rfloor$ ▷ new element's level
$l \leftarrow \min(l, L+1)$
for $lc \leftarrow L$ **to** $l+1$ **do**
  $W \leftarrow$ `SEARCH-LAYER`($q, ep,$ `ef` $= 1, lc$)
  $ep \leftarrow$ get the nearest element from W to q
end for
for $lc \leftarrow \min(L, l)$ **to** 0 **do**
  $W \leftarrow$ `SEARCH-LAYER`($q, ep,$ `efConstruction`$, lc$)
  `neighbors` $\leftarrow$ `SELECT-NEIGHBORS`(q, W, M, lc) ▷ Algorithm 3 or 4
  add bidirectional connections from `neighbors` to q at layer lc
  for each $e \in$ `neighbors` **do**
    $eComm \leftarrow$ `neighborhood`(e) at layer lc
    if $|eComm| > M_{\max}$ **then** ▷ shrink connections if needed
      if $lc = 0$ **then** ▷ if $lc = 0$ then $M_{\max} = M_{\max 0}$
        $M_{\max} \leftarrow M_{\max 0}$
      end if
      $eComm \leftarrow$ `SELECT-NEIGHBORS`($e, eComm, M_{\max}, lc$) ▷ Algorithm 3 or 4
    end if
    set `neighborhood`(e) at layer lc to $eComm$
  end for
end for
$ep \leftarrow W$
set enter point for `hsnw` to q

Closest neighbors at each layer are identified using a variant of the greedy search algorithm (as outlined in Algorithm 2). To approximate the ef nearest neighbors at a given layer l_c, the algorithm maintains a dynamic list W, which stores the ef closest found elements. This list is initially populated with the entry points and is iteratively updated during the search process.

Algorithm 2 SEARCH-LAYER(q, ep, ef, lc)

Input: Query element q, enter points ep, number of nearest to q elements to return ef, layer number lc
Output: ef closest neighbors to q

```
v ← ep                                              ▷ Set of visited elements
C ← ep                                              ▷ Set of candidates
W ← ep                                              ▷ Dynamic list of found nearest neighbors
while |C| > 0 do
    c ← Extract nearest element from C to q
    f ← Get furthest element from W to q
    if distance(c, q) > distance(f, q) then
        break                                       ▷ All elements in W are evaluated
    end if
    for each e ∈ neighbourhood(c) at layer lc do    ▷ Update C and W
        if e ∉ v then
            v ← v ∪ e
            f ← Get furthest element from W to q
            if distance(e, q) < distance(f, q) or |W| < ef then
                C ← C ∪ e
                W ← W ∪ e
                if |W| > ef then
                    Remove furthest element from W to q
                end if
            end if
        end if
    end for
end while
return W
```

At each step, the algorithm evaluates the neighborhood of the closest, yet-to-be-evaluated element in W. The search continues until the neighborhoods of all elements in W have been evaluated. Unlike traditional approaches that limit the number of distance calculations, the HNSW stop condition offers a key advantage: it discards candidates for evaluation that are farther from the query than the furthest element already in W. This approach prevents unnecessary evaluations and avoids bloating the search structures, ensuring a more efficient and streamlined search process.

In the HNSW algorithm, two methods are employed to select M neighbors from a pool of candidates during the graph construction phase: Simple (Algorithm 3) and Heuristic (Algorithm 4).

Algorithm 3 SELECT-NEIGHBORS-SIMPLE(q, C, M)

Input: Base element q, candidate elements C, number of neighbors to return M
Output: M nearest elements to q

```
return M nearest elements from C to q
```

Algorithm 4 SELECT-NEIGHBORS-HEURISTIC(q, C, M, lc, `extendCandidates`, `keepPrunedConnections`)

Input: Base element q, candidate elements C, number of neighbors to return M, layer number lc, flag indicating whether or not to extend candidate list `extendCandidates`, flag indicating whether or not to add discarded elements `keepPrunedConnections`

Output: M elements selected by the heuristic

$R \leftarrow \emptyset$
$W \leftarrow C$ ▷ Working queue for the candidates
if `extendCandidates` **then** ▷ Extend candidates by their neighbors
    for each $e \in C$ **do**
        for each $e_{\text{adj}} \in$ `neighbourhood`(e) at layer lc **do**
            if $e_{\text{adj}} \notin W$ **then**
                $W \leftarrow W \cup e_{\text{adj}}$
            end if
        end for
    end for
end if
$W_d \leftarrow \emptyset$ ▷ Queue for the discarded candidates
while $|W| > 0$ **and** $|R| < M$ **do**
    $e \leftarrow$ Extract nearest element from W to q
    if e is closer to q compared to any element from R **then**
        $R \leftarrow R \cup e$
    else
        $W_d \leftarrow W_d \cup e$
    end if
end while
if `keepPrunedConnections` **then** ▷ Add some of the discarded connections from W_d
    while $|W_d| > 0$ **and** $|R| < M$ **do**
        $R \leftarrow R \cup$ Extract nearest element from W_d to q
    end while
end if
return R

The K-approximate nearest neighbor search (K-ANNS) algorithm, as implemented in the HNSW framework, is detailed in Algorithm 5. It is conceptually similar to the insertion process of an item with a designated top layer $l = 0$. However, instead of establishing new connections, the goal of the K-ANNS algorithm is to identify and return the closest neighbors in the graph's ground layer as the final search results.

Algorithm 5 K-ANN-SEARCH(`hsnw`, q, K, ef)

Input: Multilayer graph `hsnw`, query element q, number of nearest neighbors to return K, size of the dynamic candidate list ef

Output: K nearest elements to q

$W \leftarrow \emptyset$ ▷ Set for the current nearest elements
$ep \leftarrow$ Get enter point for `hsnw`
$L \leftarrow$ Level of ep ▷ Top layer for `hsnw`
for $lc \leftarrow L \ldots 1$ **do**
    $W \leftarrow$ `SEARCH-LAYER`$(q, ep, \texttt{ef} = 1, lc)$
    $ep \leftarrow$ Get nearest element from W to q
end for
$W \leftarrow$ `SEARCH-LAYER`$(q, ep, ef, lc = 0)$
return K nearest elements from W to q

2.1.4 Pre-retrieval

Query expansion: In the context of RAG, a single query can be transformed and expanded into multiple variations that are semantically enriched, capturing different nuances and interpretations of the original input. By diversifying the queries, the system increases the likelihood of retrieving contextually specific and relevant information from the knowledge base. This enriched retrieval process ensures that the generated responses are not only more precise but also better aligned with the user's intent, ultimately enhancing the quality and relevance of the output provided by the model [15].

Query rewriting: The original queries are not always optimal for LLM retrieval, especially in real-world scenarios. Therefore, we can prompt LLM to rewrite the queries [15]. Query rewriting in RAG refers to the process of refining or reformulating user queries to enhance the retrieval module's effectiveness. This step is critical in ensuring that the retriever fetches highly relevant and accurate documents or context from the knowledge base. Effective query rewriting includes disambiguating vague queries, expanding them with synonyms or related terms, correcting errors, and incorporating historical or contextual information. By improving the quality of the input to the retriever, query rewriting significantly boosts the overall performance of the RAG system, leading to more accurate and contextually appropriate outputs from the generation module. A user query can be transformed and decomposed in many ways before being executed as part of a RAG query engine, agent, or any other pipeline.

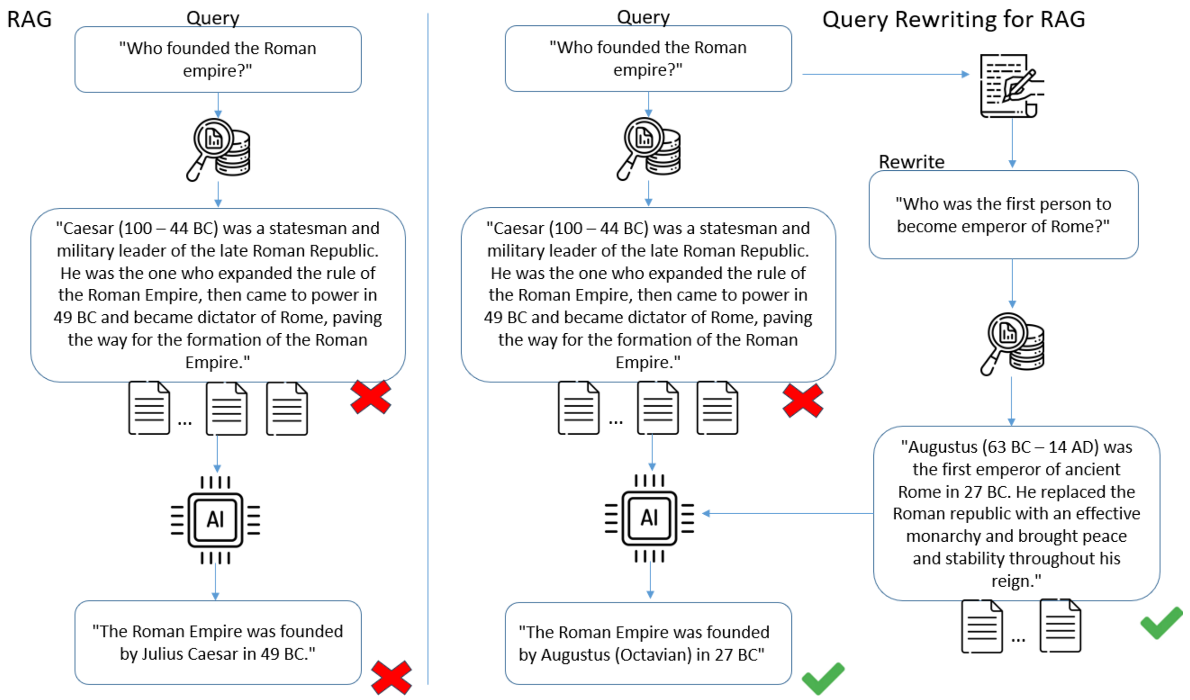

Figure 2.4: Query rewriting process in RAG

Embedding transformation: involves either pretraining language models or leveraging pretrained models to gain a deep understanding of the structural and semantic characteristics of text data. This process enables language models to map both queries and structured texts into one unified embedding space, facilitating efficient and accurate retrieval [31]. By doing so, the model enables efficient and accurate retrieval by ensuring that semantically related information from different formats is aligned within the same representation space. This unified embedding not only improves retrieval precision but also facilitates the integration of heterogeneous knowledge sources into the retrieval process.

2.1.5 Retrieval

Once the embedding process is complete, a document retrieval system identifies and returns the most relevant documents based on the query. In a RAG pipeline, this document retrieval step plays a crucial role, supplying the LLM with the appropriate documents for generating contextually accurate responses. Formally, given a query q in an arbitrary language x, it can retrieve document d in language y (which can be another language or the same language as x) from the corpus D^y. Moreover, the retrieval function $fn^*()$ belongs to any of the functions: dense, lexical, or multi-vector retrieval [8].

$$d^y \leftarrow \text{fn}^*(q^x, D^y)$$

- **Dense retrieval**

  Query q is transformed into the hidden states H_q based on a text encoder.
  The normalized hidden state of the special token "[CLS]" is used for the representation of the query:

  $$e_q = \text{norm}(H_q[0])$$

  Similarly, the embedding of passage p is:

  $$e_p = \text{norm}(H_p[0])$$

  The relevance score between query and passage is measured by the inner product between the two embeddings e_q and e_p

  $$s_{dense} \leftarrow \langle e_p, e_q \rangle$$

- **Lexical retrieval**

  For each term **t** in the query:

  $$w_{q_t} \leftarrow \text{ReLU}(W_{lex}^T H_q[i])$$

  Similarly, for the passage:

  $$w_{p_t} \leftarrow \text{ReLU}(W_{lex}^T H_p[i])$$

  Where:
  - $W_{lex} \in R^{d \times 1}$: A weight matrix mapping the hidden state to a scalar.
  - $H_q[i]$, $H_p[i]$: Hidden states (embedding vectors) for the i-th token in the query and passage.
  - ReLU: Rectified Linear Unit activation function, defined as $\text{ReLU}(x) = \max(0, x)$.

- **Multi-vector retrieval**

  Representing query and passage with multiple vectors
  Formulas:
  - Query representation:

  $$E_q = \text{norm}(W_{mul}^\top H_q)$$

  - Passage representation:

  $$E_p = \text{norm}(W_{mul}^\top H_p)$$

  - $W_{mul} \in \mathbb{R}^{d \times d}$: A learnable projection matrix.
  - H_q and H_p: Hidden states (embeddings) for the query and passage, respectively.
  - norm: Column-wise normalization function applied to each vector in the matrices.

2.1.6 Post-retrieval

Reranking: The initial retrieval stage often relies on methods like dense or lexical retrieval, which are efficient but may not consistently yield highly relevant documents. To address this, reranking plays a crucial role by reorganizing the retrieved documents to better align with the given query, thereby improving their relevance.

Contextual compression: Contextual compression addresses the challenge of retrieving relevant information from large documents by compressing and filtering content based on the query. Instead of passing entire documents, which can lead to expensive and less effective responses, this approach refines the retrieved results. Using a base retriever and a document compressor tools like Compact [60], it will shortens documents by extracting only the relevant content or removing irrelevant ones entirely, ensuring more efficient and focused query responses.

2.2 Generator

The posed query and selected documents are synthesized into a coherent prompt to which a large language model is tasked with formulating a response. The model's approach to answering may vary depending on task-specific criteria, allowing it to either draw upon its inherent parametric knowledge or restrict its responses to the information contained within the provided documents. In cases of ongoing dialogues, any existing conversational history can be integrated into the prompt, enabling the model to engage in multi-turn dialogue interactions effectively [15].

In this section, we provide an overview of the architecture of Transformer models [52] and delve into their core mechanisms that have revolutionized natural language processing. Following this, we explore various prompt engineering techniques, highlighting their role in optimizing the performance of pre-trained language models across a wide range of tasks, including RAG. These techniques not only improve the model's understanding of task-specific contexts but also enhance its ability to generate accurate and contextually relevant responses.

2.2.1 Transformer architecture

Transformer [52], a model architecture eschewing recurrence and instead relying entirely on an attention mechanism to draw global dependencies between input and output [52]. It has revolutionized natural language processing (NLP) by enabling parallel processing of sequences. Unlike recurrent architectures such as RNNs [44] or LSTMs [19], which process inputs sequentially and struggle with long-range dependencies, Transformers use self-attention mechanisms to focus on relevant parts of the input across the entire sequence simultaneously. This approach not only improves the ability to capture intricate relationships within the data but also drastically increases computational efficiency by enabling systems to handle tasks in parallel. The result is a model architecture that excels at tasks requiring a deep understanding of context, from machine translation to text generation, setting the foundation for state-of-the-art language models like GPT [37], BERT [12], and T5 [39].

Encoder and decoder stacks: A transformer consists of two components: encoder and decoder. Each of them is a stack of N identical layers (N=6 in the original transformer model).

- **Encoder**: The encoder is composed of a stack of N identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. Around each of the two sub-layers, there is a residual connection [17] followed by layer normalization [2]. That is, the output of each sub-layer is $\text{LayerNorm}(x + \text{Sublayer}(x))$, where $\text{Sublayer}(x)$ is the function implemented by the sub-layer itself [52].

- **Decoder**: The decoder, like the encoder, is also composed of a stack of N identical layers. In addition to the two sub-layers in each encoder layer, the encoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder Stack. The the self-attention sub-layer in the decoder stack is modified to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i [52].

Positional encoding: plays a critical role in capturing the order of sequences, which is essential for preserving the structural integrity of text data. Positional encodings are added to the input embeddings at the initial layers of both the Encoder and Decoder stacks, enabling the model to incorporate sequential information.

In the original model, sine and cosine functions of different frequencies is used for positional encoding:

$$\text{PE}(pos, 2i) = \sin(pos/10000^{2i/d_{model}}) \tag{2.1}$$

$$\text{PE}(pos, 2i+1) = \cos(pos/10000^{2i/d_{model}}) \tag{2.2}$$

where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid [52].

Self-attention mechanism:

- **Queries, keys and values**: In self-attention mechanism, transformer compute output by mapping a query with a set of key-value pairs. The output is computed as a weighted sum of the values, where the weight computed by scaled dot-product of query and keys.

- **Scaled-dot products**: The concept of self-attention lies in computing a weighted sum of values, where the weights represent the similarity between queries and keys. In Transformers, this similarity is determined using a dot product between the query and key vectors, followed by a softmax function. The softmax operation normalizes these weights, ensuring they fall within the range of 0 and 1, thus effectively controlling their influence.
- The vectors for queries, keys, and values are all of size d_K. When computing the dot product between queries and keys, the resulting values can become disproportionately large, especially as d_K increases. Such large values can cause gradients to vanish during backpropagation, hindering the training process. To mitigate this, transformers introduce a scaling factor $\sqrt{d_K}$, dividing the dot product by this value. This adjustment helps stabilize the gradients and ensures the training remains efficient and effective, even with large vector dimensions.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_K}}V) \tag{2.3}$$

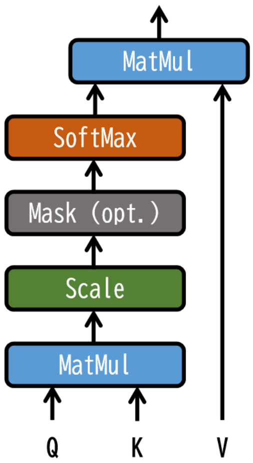

Figure 2.5: Scaled dot-product attention [52]

- **Multi-head attention**: Instead of performing a single attention function with keys, values and queries, it is good to linearly project the queries, keys and values h times with different, learned linear projections to d_k, d_k and d_v dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel. Here, $W_i^Q W_i^K \in R^{d_{model} \times d_V}, W_i^Q \in R^{d_{model} \times d_V}, W^O \in R^{hd_v \times d_{model}}$ and h is the number of attention heads.

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(head_1, ..., head_h)W^O \\ \text{where}\, head_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{2.4}$$

2.2.2 Prompt engineering

Prompt engineering is an essential methodology for maximizing the potential of pre-trained language models. It focuses on carefully crafting task-specific instructions, known as prompts, to direct the model's output effectively without requiring any modifications to its underlying parameters [46]. Various techniques have been developed to refine the practice of prompt engineering, each aimed at leveraging the full potential of pre-trained language models for diverse and complex applications. Among the most prominent methods are:

- **Zero-Shot (0S) prompting**: This technique involves providing the model with a clear and specific instruction for a task without any prior examples. This method provides maximum convenience, potential for robustness, and avoidance of spurious correlations [6]. Zero-shot prompting relies entirely on the model's pre-trained knowledge and ability to generalize across contexts, making it effective for a wide range of tasks without additional data.

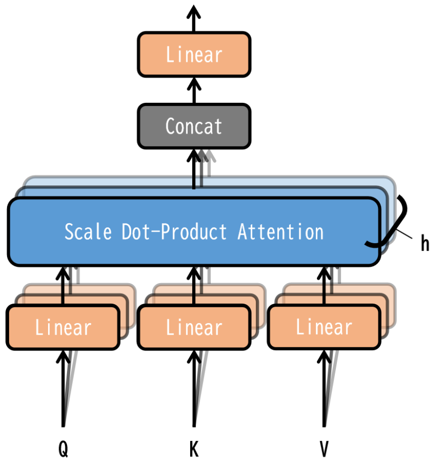

Figure 2.6: Multi-Head Attention (consists of several attention layers running in parallel) [52]

- **Few-Shot (FS) prompting**: This method works by giving K examples (with K often ranging from 10 to 100) of context and completion, and then one final example of context, then the model is expected to provide the final completion [6]. These K examples act as a guide for the model, helping it understand the structure, context, or desired behavior for the task.

- **One-Shot (1S) prompting**: This approach is the same as FS prompting but only one example is used for the prompt. This way of prompting closely matches the way in which some tasks are communicated to humans [6], normally we are given one single example or demonstration of the task then we are asked to complete other similar tasks.

- **Chain-of-Thought (CoT) prompting**: This technique involves prompting a coherent series of intermediate reasoning steps that lead to the final answer for a problem rather than providing a direct answer [58]. By simulating a logical progression of thought, the model is better equipped to handle complex tasks that require multi-step reasoning, such as mathematical problem-solving or logical deductions. This structured guidance fosters greater accuracy and depth in responses.

2.3 Augmentation methods

2.3.1 Augmentation stage: Inference

At the inference stage, augmentation involves retrieving and integrating external knowledge dynamically during the generation process. Recent work, such as in-context RALM [40], demonstrates the efficacy of prepending retrieved documents to the input text without modifying the language model (LM) architecture. This approach leverages off-the-shelf retrievers for document selection, providing significant performance gains while maintaining simplicity.

Language models define probability distributions over sequences of tokens. Given such a sequence $x_1, ..., x_n$, the standard way to model its probability is via next-token prediction:

$$\mathrm{p}(x_1, ..., x_n) = \prod_{i=1}^{n} \mathrm{p}(x_i | x_{<i}),$$

where $x_{<i} := x_1, ..., x_{i-1}$ is the sequence of tokens preceding x_i, also referred to as its prefix. This autoregressive model is usually implemented via a learned transformer network [52] parameterized by the set of parameters θ:

$$\mathrm{p}(x_1, ..., x_n) = \prod_{i=1}^{n} \mathrm{p}_\theta(x_i | x_{<i}).$$

Retrieval-augmented language models (RALMs) add an operation that retrieves one or more documents from an external corpus C, and conditions the above LM predictions on these documents. Specifically, for predicting x_i, the retrieval operation from C depends on its prefix $RC(x_{<i})$, giving rise to the general RALM decomposition:

$$p(x_1, ..., x_n) = \prod_{i=1}^{n} p(x_i | x_{<i}, RC(x_{<i})).$$

In-context RALM refers to a simple method of concatenating the retrieved documents within the transformer's input before the prefix, without altering the LM weights θ:

$$p(x_1, ..., x_n) = \prod_{i=1}^{n} p_\theta(x_i | [RC(x_{<i}); x_{<i}]),$$

where $[a; b]$ denotes the concatenation of strings a and b. This approach allows the model to incorporate external knowledge efficiently.

Two practical design choices in RALM systems significantly impact performance:

1. **Retrieval stride:** Retrieval operations can be performed once every $s > 1$ tokens to reduce computation costs. The retrieval stride s determines how often new documents are fetched, trading runtime for performance. The formulation becomes:

$$p(x_1, ..., x_n) = \prod_{j=0}^{\lfloor n/s \rfloor - 1} \prod_{i=1}^{s} p_\theta(x_{s \cdot j+i} | RC(x_{\leq s \cdot j}); x_{<(s \cdot j+i)}).$$

2. **Retrieval query length:** The retrieval query at stride j is typically restricted to the last ℓ tokens of the prefix ($p_j^{s,\ell} := x_{s \cdot j - \ell + 1}, ..., x_{s \cdot j}$) to avoid diluting the relevance of the information. The resulting formulation is:

$$p(x_1, ..., x_n) = \prod_{j=0}^{\lfloor n/s \rfloor - 1} \prod_{i=1}^{s} p_\theta(x_{s \cdot j+i} | [RC(q_j^{s,\ell}); x_{<(s \cdot j+i)}]).$$

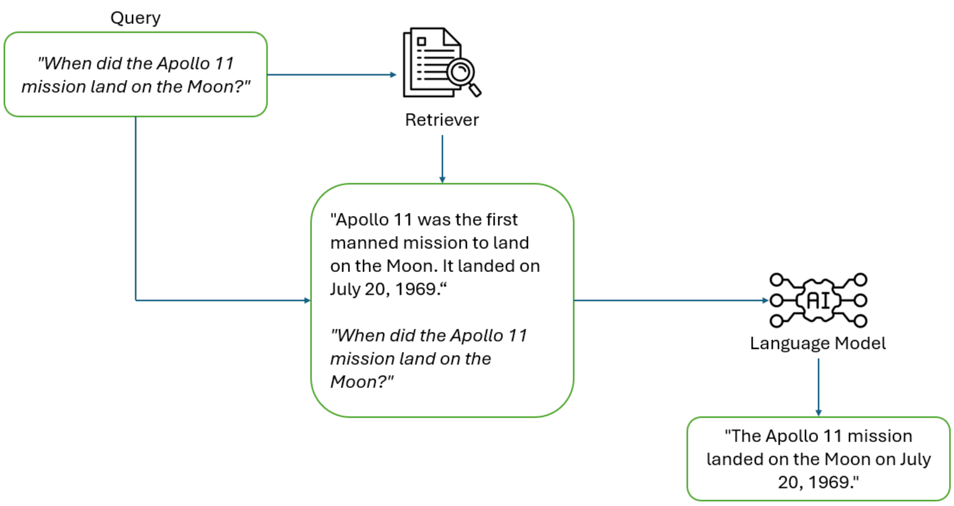

Figure 2.7: Augmentation during the inference stage. Relevant documents are retrieved and prepended to the input for improved factual grounding

2.3.2 Augmentation source: Structured data

Structured data, such as knowledge bases (KBs), offers a rich source for augmentation. The KnowledGPT framework [56] demonstrates how LLMs can interact with KBs for both retrieval and storage. By generating programmatic queries, LLMs can navigate multi-hop and entity-specific questions effectively.

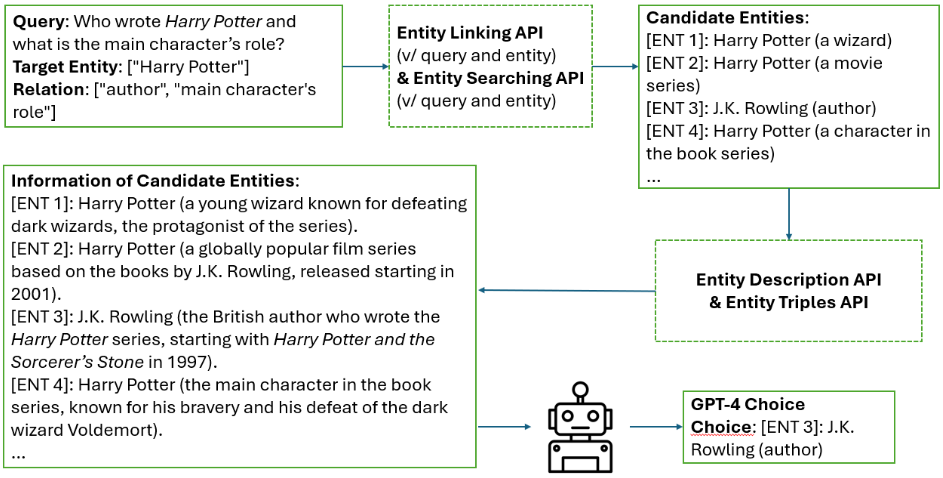

Figure 2.8: Workflow of augmentation with structured data sources [56]

To enable effective interaction with structured data, KnowledGPT employs several key functions:

- *get_ entity_ info:* Retrieves encyclopedic descriptions of an entity from the KB.
- *find_ entity_ or_ value:* Extracts entities or attribute values related to a query by identifying the most relevant relation.
- *find_ relationship:* Identifies relationships between two entities within the KB.

These functions allow KnowledGPT to leverage the structured nature of KBs to extract precise and relevant knowledge.

The *find_ entity_ or_ value* function is central to querying entities and attribute values. It matches the query's relation to KB triples using embedding-based similarity and retrieves the corresponding entities or values. The algorithm is detailed below.

2.3.3 Augmentation process

The augmentation process in RAG frameworks can be categorized into various types, such as once, iterative, recursive, and adaptive. Among these, the Self-reflective retrieval-augmented generation (SELF-RAG) [1] approach employs an adaptive process where the model dynamically determines the necessity of retrieval, evaluates the relevance and support of retrieved passages, and critiques its own output for factuality and quality.

Reflection tokens are special tokens used by SELF-RAG to evaluate and guide the retrieval and generation process.

Type	Input	Output	Definitions
Retriever	$x/x, y$	{yes, no, continue}	Decide to Retrieve with $\mathcal{R}$
IsREL	x, d	{**relevant**, irrelevant }	d provide useful information to solve x
IsSUP	x, d, y	{**fully supported**, partially supported, no support}	All of the verification-worthy statement in y is supported by d
IsUSE	x, y	{5, 4, 3, 2, 1}	y is useful response to x

Table 2.1: Reflection tokens [1]

The inference process in SELF-RAG leverages reflection tokens to dynamically retrieve relevant passages and critique the model's outputs. The steps are as follows:

1. **Input and prediction:** The model receives the input prompt x and preceding generation $y < t$. It predicts whether retrieval is necessary using the retrieve token.
2. **Retrieve:** If retrieval is needed (retrieve = yes), the retriever fetches relevant passages D based on the input and context.
3. **Generate and evaluate:** For each retrieved passage $d \in D$, the model:

Algorithm 6 find_entity_or_value

Input: query q, alias list of entity $\mathcal{E}$, alias list of relation $\mathcal{R}$.
Output: a list of target entities or attribute values $\mathcal{T}$.

```
function EMBSIM(str r, list[str] R)
    s = −1
    r = embedding(r)
    for r_i ∈ R do
        r_i = embedding(r_i)
        if cos(r, r_i) > s then
            s = cos(r, r_i)
        end if
    end for
    return s
end function
e = entity_linking(E)
if e == NULL then
    return NULL
end if
r = NULL
s_r = −1
for triple ∈ triples do
    r_i = triple.rel
    s_i = EMBSIM(r_i, R)
    if s_i > s_r then
        s_r = s_i
        r = r_i
    end if
end for
if r == NULL then
    return NULL
end if
triples_r = triples with relation r
R = target entities or attribute values in triples_r
return R
```

- Predicts IsREL to assess relevance.
- Generates the next segment y_t.
- Predicts IsSUP and IsUSE to evaluate support and utility.

4. **Rank and select:** The model ranks all generated segments based on IsREL, IsSUP, and IsUSE, and selects the top-ranked segment.

5. **No retrieval case:** If retrieval is not needed (Retrieve = no), the model directly generates the next segment and predicts its utility.

6. **Iterative refinement:** The process repeats until the response is complete.

Algorithm 7 SELF-RAG inference

Require Generator LM $\mathcal{M}$, Retriever $\mathcal{R}$, Large-scale passage collections $\{d_1, \ldots, d_N\}$

```
Input: input prompt x and preceding generation y_{<t}, Output: next output segment y_t
M predicts Retriever given (x, y_{<t})
if Retriever == Yes then
    Retrieve relevant text passages D using R given (x, y_{t-1})        ▷ Retrieve
    M predicts IsREL given x, d and y_{<t} for each d ∈ D               ▷ Generate
    M predicts IsSUP and IsUSE given x, d for each d ∈ D                ▷ Critique
    Rank y_t based on IsREL, IsSUP, IsUSE
else if Retriever == No then
    M_gen predicts y_t given x                                          ▷ Generate
    M_gen predicts IsUSE given x, y_t                                   ▷ Critique
end if
```

3
Related works and tools

3.1 RAG and LLMs in coding tasks

3.1.1 Retrieval strategies

Sparse retrieval methods: Tools like BM25 [5] rank documents based on term frequency and inverse document frequency (TF-IDF). While highly effective for small-scale knowledge bases, their efficiency declines as data scales increase [18].

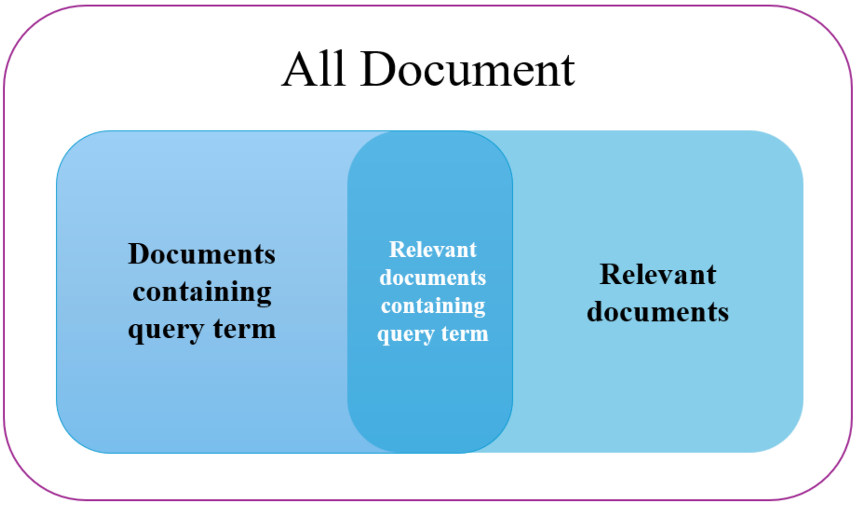

Figure 3.1: BM25 visualization

Dense retrieval methods: Leveraging semantic embeddings, dense retrieval tools like SBERT [41]'s semantic search and approximate techniques (e.g., HNSW [32], ANNOY [4], and LSH [23]) provide scalable solutions for large knowledge bases. These approaches achieve significant speed improvements with minimal accuracy trade-offs [18].

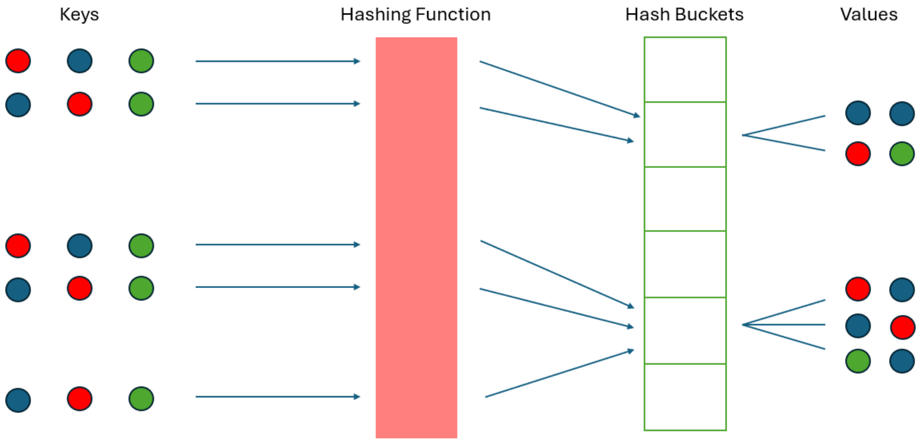

Figure 3.2: LSH visualization

Selective retrieval: Frameworks such as Repoformer [59] employ selective retrieval, determining the necessity of retrieval based on contextual evaluation. This approach reduces inefficiencies and avoid adding irrelevant context.

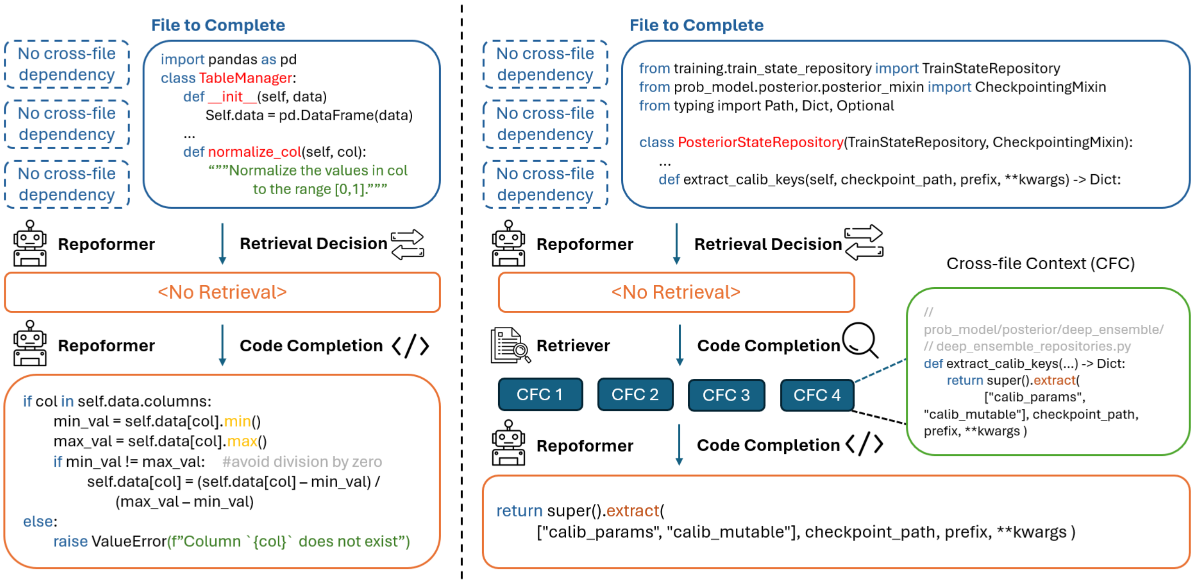

Figure 3.3: REPOFORMER framework [59]

Multi-retrieval perspectives: Systems like ProCC [48] utilize diverse perspectives, including lexical semantics, hypothetical line embeddings, and code summarization, to enrich context and improve code completion accuracy.

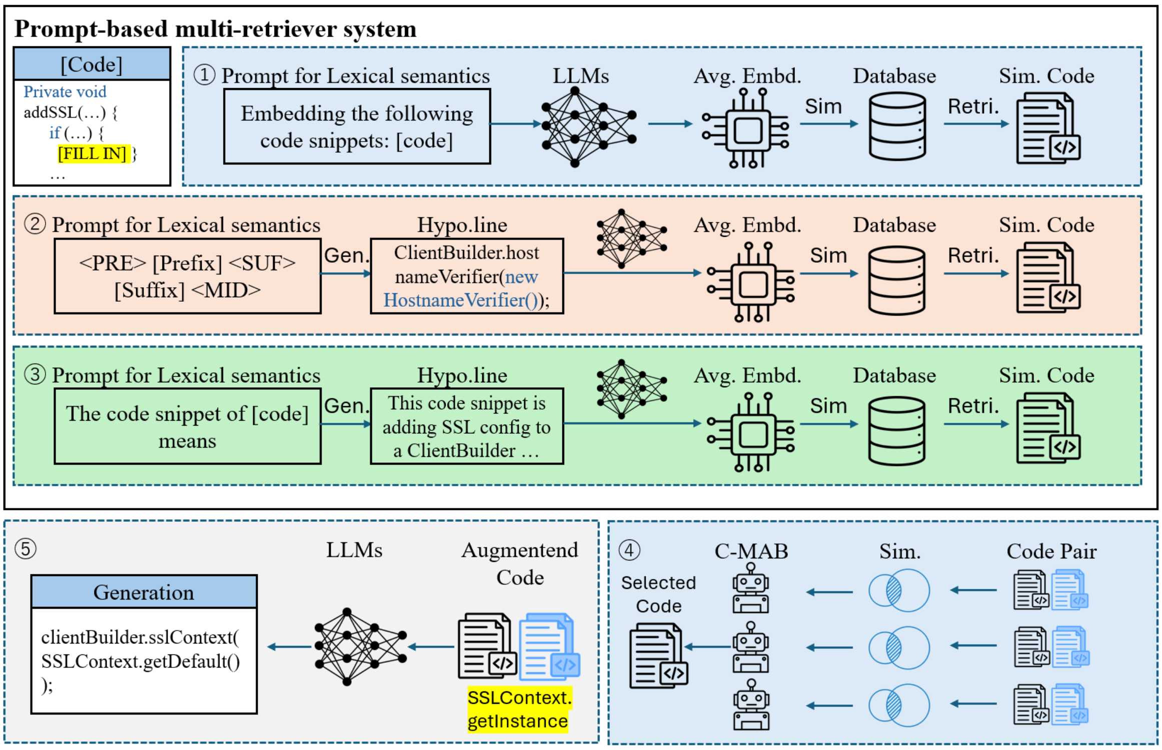

Figure 3.4: ProCC framework [48]

Multilingual and multifunctional embeddings: Models like BGE M3-Embedding [8] support over 100 languages and various retrieval functionalities, including dense, multi-vector, and sparse retrieval, enhacing versatility in coding tasks.

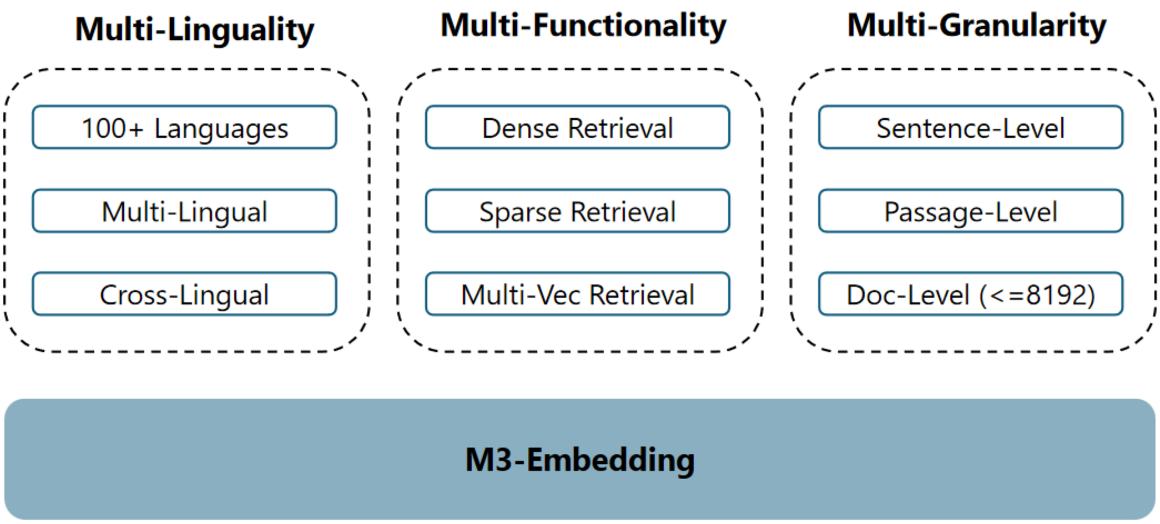

Figure 3.5: Characters of M3-Embedding [8]

3.1.2 Generative enhancements

Generative Pre-trained Transformers: LLMs like GPT-2 [38] and code Llama [43] are trained on extensive code corpora, enabling them to handle diverse tasks such as function completion and API usage.

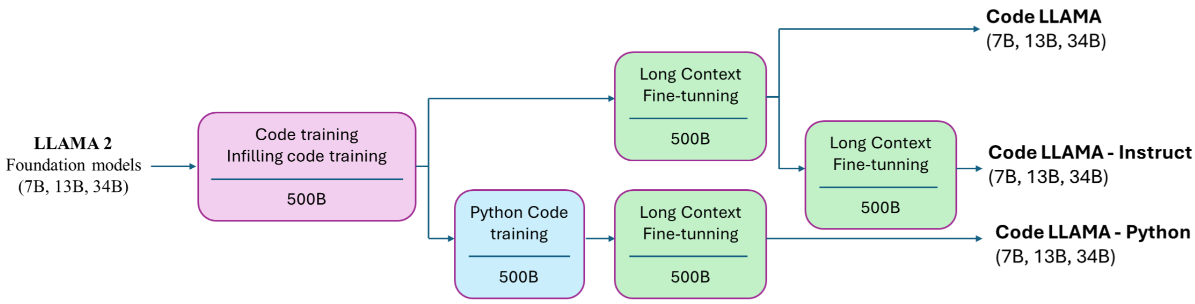

Figure 3.6: The Code Llama specialization pipeline [43]

In-context augmentation: RAG pipelines integrate retrieved code snippets directly into prompts, enhancing generation without additional fine-tuning. This approach is particularly effective for repository-level code generation [20].

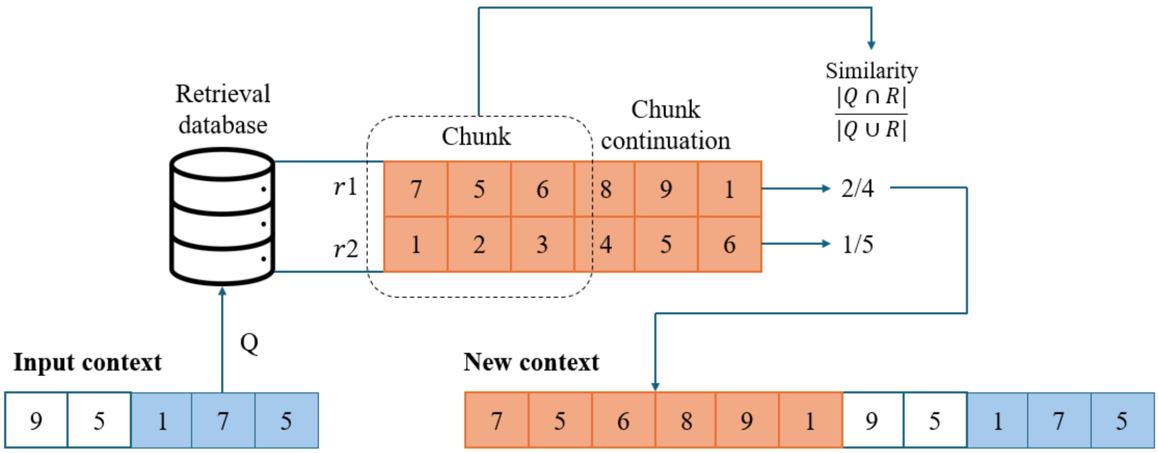

Figure 3.7: The retrieved snippet contains continuation tokens and additional metadata [20]

Specialized models for coding: Models like Qwen2.5-Coder [21] and CodeQwen1.5-7B-Chat [3] are fine-tuned for coding tasks, demonstrating improved performance in code generation and comprehension.

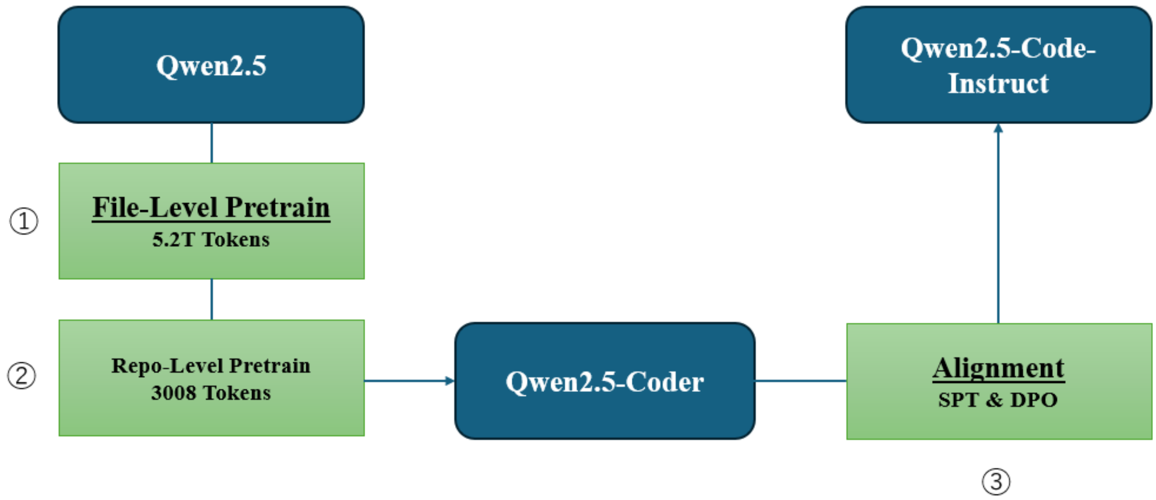

Figure 3.8: The three-stage training pipeline for Qwen2.5-Coder [20]

3.1.3 Specialized Applications

Repository-level code completion: Tools like Repoformer and RepoCoder [61] focus on cross-file dependencies, retrieving and integrating relevant code snippets to improve repository-wide coherence.

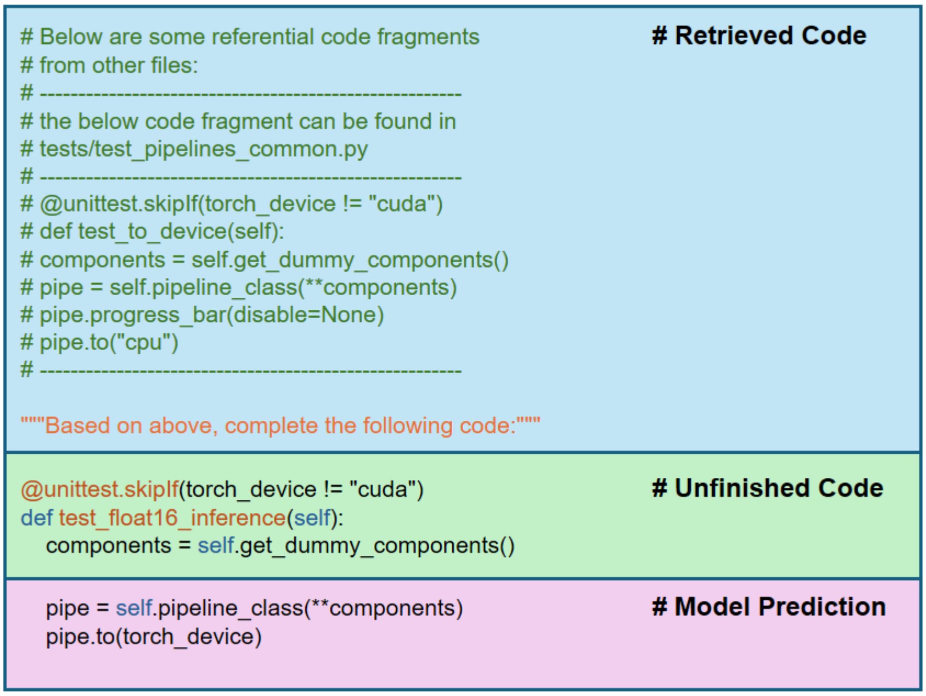

Figure 3.9: Demonstrating the format of the RepoCoder prompt [61]

Commit message and assertion generation: RAG systems generate commit messages and assertions by retrieving semantically aligned examples, balancing efficiency and effectiveness [18].

Code hallucination mitigation: Automated code generation, powered by LLMs, has significantly enhanced development efficiency by generating code from input requirements. However, LLMs often produce hallucinations, particularly when handling complex contextual dependencies in repository-level development scenarios. To address these challenges, RAG-based mitigation methods have been proposed [64].

3.2 Developer assistance tools

3.2.1 Functionality

Search and Retrieval Tools: Platforms like stack overflow[1] provide a database of Q&A posts where users search for solutions based on keywords or queries. Neural code search (NCS) [45] enhances this by enabling natrual language queries over raw codebases without curated answers, using vector embeddings and ranking strategies.

[1]Stack Overflow - https://stackoverflow.com/

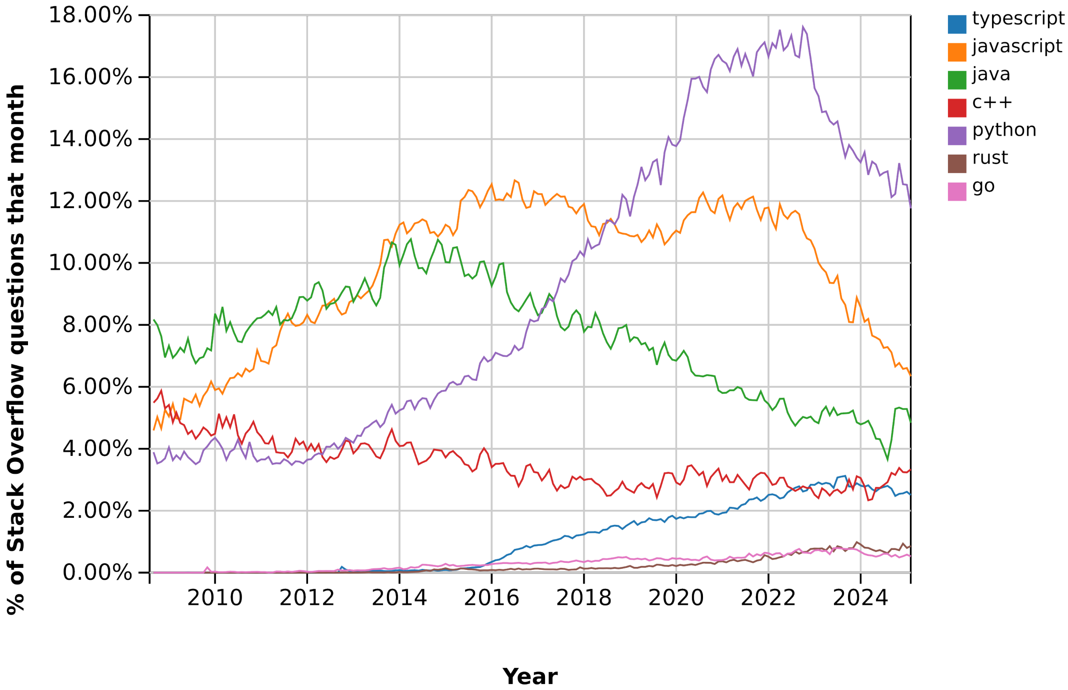

Figure 3.10: Stackoverflow trends chart 2008-2025

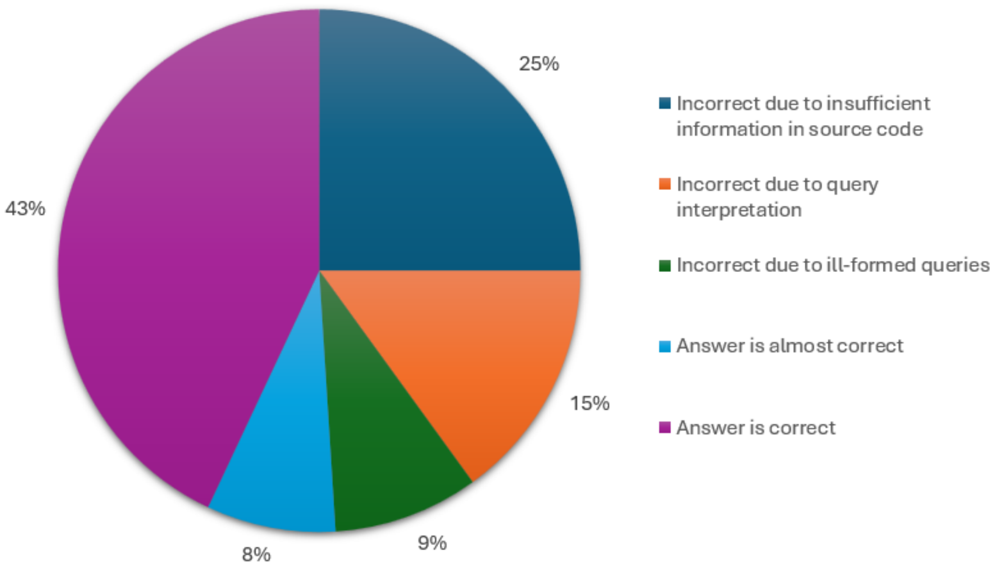

Figure 3.11: NCS evaluation results [45]

Code completion and suggestion systems: Intelligent tools like IntelliCode Compose [47] and example-based systems [7] prioritize providing relevant suggestions for method calls, arguments, and entire lines of code. These systems are designed to enhance developer productivity by learning from existing codebases.

Hybrid generation systems: RAG models combine retrieval with generative capabilities [28], producing specific and contextually appropriate outputs by accessing external document repositories and leveraging models

3.2.2 Retrieval mechanisms

Keyword-based retrieval: Traditional forums rely on keyword searches, often yielding broad results with low contextual relevance.

Embedding-based retrieval: Tools like NCS employ word embeddings, TF-IDF weighting [34], and vector similarity for precise retrieval of relevant code snippets. These systems also incorporate syntactic and semantic elements extracted from source code.

Rule-based retrieval: Example-based systems mine patterns and associations (e.g., frequent co-occurrence of API calls) from repositories to recommend contextually relevant options.

3.2.3 Code generation capabilities

Static suggestion systems: Earlier systems [7] relied solely on static type information, offering alphabetically sorted method recommendations without understanding contextual relevance.

Generative models: IntelliCode Compose represents a generative approach, using transformer-based architectures [52] like GPT-C to predict and generate entire lines or blocks of syntactically correct code. This includes multilingual support and optimized decoding methods like beam search.

3.2.4 Integration in development processes

Standalone search interfaces:

- **NCS** [45]: Functions as an external platform where developers input natural language queries to retrieve relevant code snippets directly from large codebases.
- **Sourcegraph**[2]: Provides a universal code search and navigation tool, enabling developers to search across multiple repositories and languages from a centralized interface.
- **Krugle**[3]: A specialized search engine designed for searching open-source and enterprise code repositories. It indexes and organizes code by context, allowing developers to locate relevant examples, API usage, and documentation efficiently.

Feature	NCS	Sourcegraph	Krugle
Functionality	Retrieves code snippets from large codebases using natural language queries.	Provides universal code search and navigation across multiple repositories and languages.	Searches open-source and enterprise code repositories, indexing code by context.
Search methodology	Maps natural language queries and code snippets into a shared vector space using neural networks.	Offers structural code search, matching nested expressions and code blocks beyond regular expressions.	Indexes code with advanced filtering for examples, API usage, and documentation.
Supported languages	Initially focused on Java.	Supports over 30 programming languages.	Supports multiple programming languages.
Deployment options	Research project with datasets available for evaluation.	Offers self-hosted solutions via docker compose and kubernetes, as well as a managed cloud service.	Web-based platform.

Table 3.1: Comparison of NCS, Sourcegraph, and Krugle

IDE-embedded assistance:

- **IntelliCode Compose** [47]: Integrates into VS Code, offering real-time code generation and suggestions.
- **Github Copilot**[4]: Embedded within various IDEs, including VS Code and JetBrains, it leverages AI to provide code completions and suggestions based on the current context.

[2]Sourcegraph - https://sourcegraph.com/
[3]Krugle - https://www.krugle.com/
[4]Github Copilot - https://github.com/features/copilot

- **Tabnine**[5]: An AI-powered code completion tool that integrates with various IDEs, offering whole-line and full-function code completions based on the current code context.

Feature	IntelliCode Compose	GitHub Copilot	Tabnine
Functionality	Provides real-time code generation and suggestions within VS Code.	Offers AI-driven code completions and suggestions across various IDEs.	Delivers whole-line and full-function code completions based on current code context.
Supported IDEs	VSCode.	VSCode, JetBrains suite, Neovim, and others.	VSCode, IntelliJ IDEA, Sublime Text, Atom, and more.
AI model	Utilizes machine learning models trained on open-source GitHub repositories.	Powered by OpenAI's Codex model, trained on a vast dataset of public code.	Employs proprietary models trained on permissively licensed open-source code.
Privacy and security	Processes code locally, enhancing privacy.	Processes code in the cloud, which may raise privacy concerns for sensitive codebases.	Offers both cloud-based and on-premises solutions, with options for local processing to ensure code privacy.
Pricing	Included with VS Code.	Individual model: \$10/month; Business version at \$19/month and Enterprise at \$39/user/month	Basic model: free version with basic features; Dev version at \$9/month and Enterprise at \$39/user/month

Table 3.2: Comparison of IntelliCode Compose, GitHub Copilot, and Tabnine

```python
# From: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
import torch
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5,
    0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=1, shuffle=False,
    num_workers=2)
```

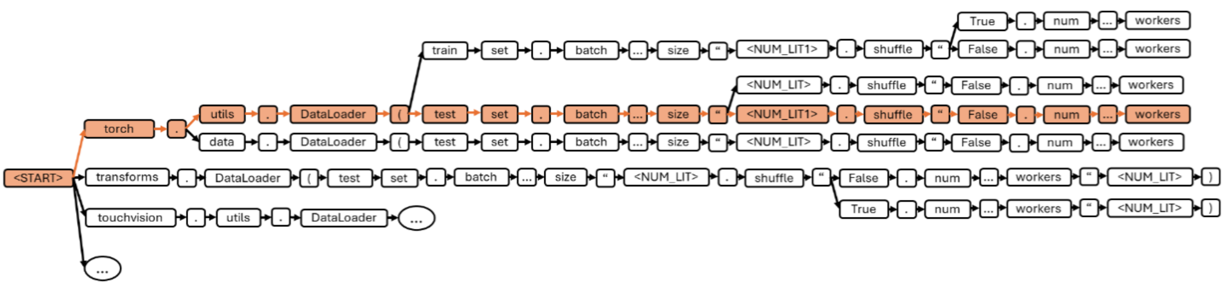

Figure 3.12: IntelliCode Compose's code snippet completion suggestion and completion-tree demonstration [47]

[5]Tabnine - https://www.tabnine.com/

4

Proposed solution

4.1 Fine-tuning vs. RAG

4.1.1 Fine-tuning

Fine-tuning involves adapting a pre-trained LLM to specific tasks or domains by further training it on a targeted dataset. This process refines the model's parameters to align with specialized requirements, enhancing its performance in particular applications.

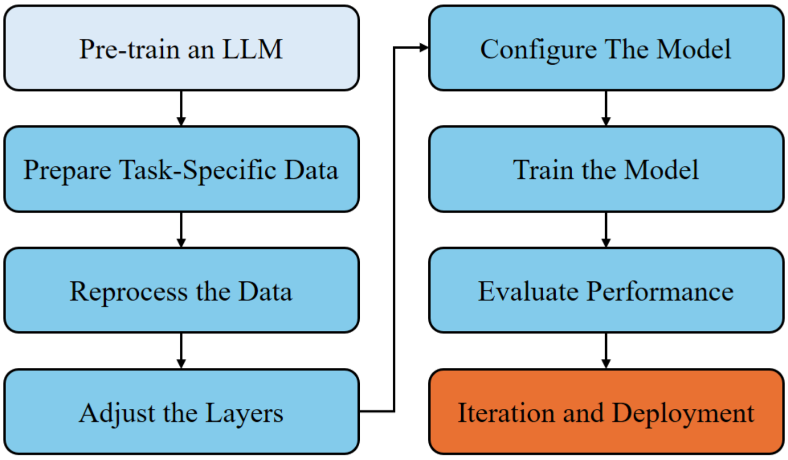

Figure 4.1: Fine-tuning process[1]

[1] RAG vs Fine-Tuning: A Comparative Analysis of LLM Learning Techniques - `https://addepto.com/blog/rag-vs-fine-tuning-a-comparative-analysis-of-llm-learning-techniques/`

4.1.2 Comparison between fine-tuning and RAG

Aspect	Fine-tuning	RAG
Dynamic vs. Static	Static: Relies on pre-trained knowledge, requiring retraining to update information.	Dynamic: Accesses up-to-date external data sources, providing current information without retraining.
Architecture	Utilizes a pre-trained LLM adjusted with task-specific datasets.	Combines LLM with external knowledge bases, enabling retrieval and generation.
Customization	High customization in domain-specific knowledge, tone, and style.	Limited customization to domain behavior but excels in integrating external content.
Hallucinations	Reduces hallucinations through domain-specific training but may hallucinate with unknown queries.	Minimizes hallucinations by grounding responses in retrieved documents.
Accuracy	High accuracy in specific, trained domains but limited outside the training scope.	Highly accurate for retrieving up-to-date, domain-relevant information.
Transparency	Functions as a "black box" with limited insight into response reasoning.	Transparent response generation with traceable data sources.
Cost	Higher cost due to significant computational resources, labeled data requirements, and periodic retraining for updates.	Lower cost as it eliminates the need for retraining and requires minimal or no labeled data by leveraging external knowledge bases.
Complexity	More complex due to the need for expertise in NLP, deep learning, and hyperparameter tuning.	Simpler to set up with minimal coding for retrieval integration.
Adaptability	Less adaptable, requiring retraining for new information.	Easily adapts to new data and dynamic environments.

Table 4.1: Comparison of fine-tuning and RAG

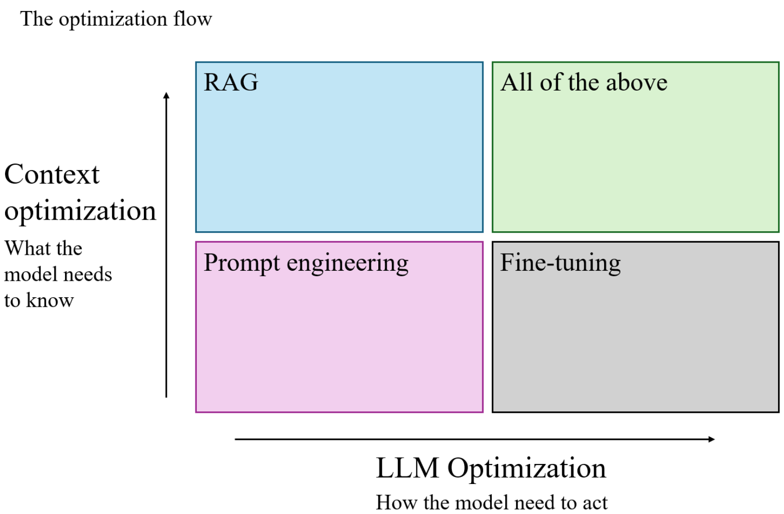

Figure 4.2: The nature of fine-tuning and RAG[2]

4.1.3 Rationale for choosing RAG

Minimize hallucinations: RAG provides real-time access to the latest documentation and programming guidelines, ensuring that code suggestions are current and precise without the need for extensive retraining.

[2]Fine-tuning vs. RAG: Understanding the Difference - `https://finetunedb.com/blog/fine-tuning-vs-rag/`

Transparency: By grounding responses in specific, retrievable documents, RAG offers clear traceability of information sources. This transparency fosters trust in the generated outputs, as developers can verify the origins of the suggestions provided.

4.2 Why BGE-M3?

In RAG systems, the quality of embeddings plays a critical role in determining the relevance and precision of retrieved content. BGE-M3 [8], a state-of-the-art embedding model, has demonstrated superior performance in multilingual retrieval tasks, as shown in Table 4.2. Compared to baseline models such as BM25 [42], mDPR [63], mContriever [22], $mE5_{large}$ [55], $E5_{mistral-7b}$ [54], BGE-M3 achieves an average retrieval score of 71.5 on the MIRACL dataset [62], significantly outperforming prior methods. This score reflects its advanced ability to generate robust and semantically meaningful representations.

The strength of BGE-M3 lies in its ability to unify dense retrieval, sparse retrieval, and multi-vector retrieval techniques. This integration allows it to handle diverse query types with high accuracy and flexibility. Notably, its performance exceeds that of other models in the M3-Embedding framework, with its "All" configuration achieving the highest score by combining the strengths of different retrieval approaches.

In our RAG pipeline, BGE-M3 serves as the foundation for the retrieval stage, ensuring that only the most relevant and semantically precise information is retrieved from the vector database. By leveraging BGE-M3's advanced capabilities, we aim to enhance the overall effectiveness of the RAG system by improving the quality and relevance of the retrieved context, thus contributing to more accurate and contextually appropriate responses.

Model	Avg Score
Baselines	
BM25 [42]	31.9
$E5_{mistral-7b}$ [54]	63.4
OpenAI-3	54.9
M3-Embedding	[8]
Dense	69.2
Sparse	53.9
Multi-vector	70.5
Dense+Sparse	70.4
All	**71.5**

Table 4.2: Comparison of BGE-M3 embeddings with baseline models for multi-lingual retrieval performance on the MIRACL dataset [8]

4.3 Why Ollama's LLMs

In designing a fully local RAG system, Ollama was selected as the LLM runtime for its ease of use, lightweight architecture, and compatibility with a wide range of open-source models. Ollama provides a simple yet powerful API interface for serving large language models directly on local machines, removing the dependency on cloud-based inference and ensuring full control over data privacy and latency. Its support for models such as Qwen, LLaMA, and Mistral makes it a versatile backend for experimentation and deployment. Moreover, the CLI-based installation and Docker integration simplify the setup process, aligning with the goal of being developer-friendly and easy to deploy. By utilizing Ollama, our system guarantees that all generation processes remain offline, making it suitable for privacy-conscious environments and offline development workflows.

4.4 Why Docker?

Docker was chosen as the containerization solution for our RAG system to ensure portability, consistency, and ease of deployment across various development environments. Our backend—comprising the retrieval service, embedding logic, and integration with the vector database—is relatively complex and includes multiple dependencies and components that can be challenging to configure manually. Docker provides a robust solution to package these services into isolated containers, enabling users to launch the full backend stack with a single command, regardless of their underlying operating system or local environment setup.

By using Docker, we abstract away the intricacies of installing and configuring backend tools such as the Milvus vector database and the application logic for hybrid retrieval. This not only simplifies onboarding for new users but also ensures that the system behaves consistently across machines, eliminating common "it works on my machine" problems. Additionally, Docker's support for volume mounting and environment variables offers flexibility in managing different versions of data, models, or configurations without altering the core codebase.

Although the backend is not lightweight due to the nature of the services it includes, containerization allows the system to remain modular and maintainable. Users can deploy updates or scale specific services independently, without affecting the rest of the system. This separation of concerns aligns with modern software engineering best practices and makes the system easier to extend and maintain in the long term.

4.5 Why VS Code Extension?

Integrating the RAG system into a Visual Studio Code extension provides seamless, in-editor support for developers within their preferred development environment. This integration enables users to query documentation, generate code, and receive intelligent suggestions in real time—without needing to switch between tools or leave the editor. By minimizing context-switching, it streamlines the development workflow and boosts productivity. Visual Studio Code's extensibility, widespread usage in the web development community, and robust support for terminals and custom interfaces make it an ideal platform for embedding such a developer-focused RAG solution.

4.6 Main architecture

Our system, called RAGGIN (Retrieval-Augmented Generation for Guided Intelligence in Next.js), is designed to enhance the performance and contextual understanding of large language models by leveraging external knowledge sources. The system architecture is composed of the following key components:

- **Vector database:** Stores the Next.js documentation in an optimized vector format to enable efficient semantic retrieval.
- **Retriever:** Accepts the user query sent from the VS Code extension and retrieves the most relevant documents from the vector database.
- **Augmentation:** Combines the user query and retrieved documents to construct a context-rich prompt, including relevant context and additional questions, and forwards it to the generator.
- **Generator:** Receives the augmented prompt and utilizes the local LLM to generate accurate and contextually relevant responses.

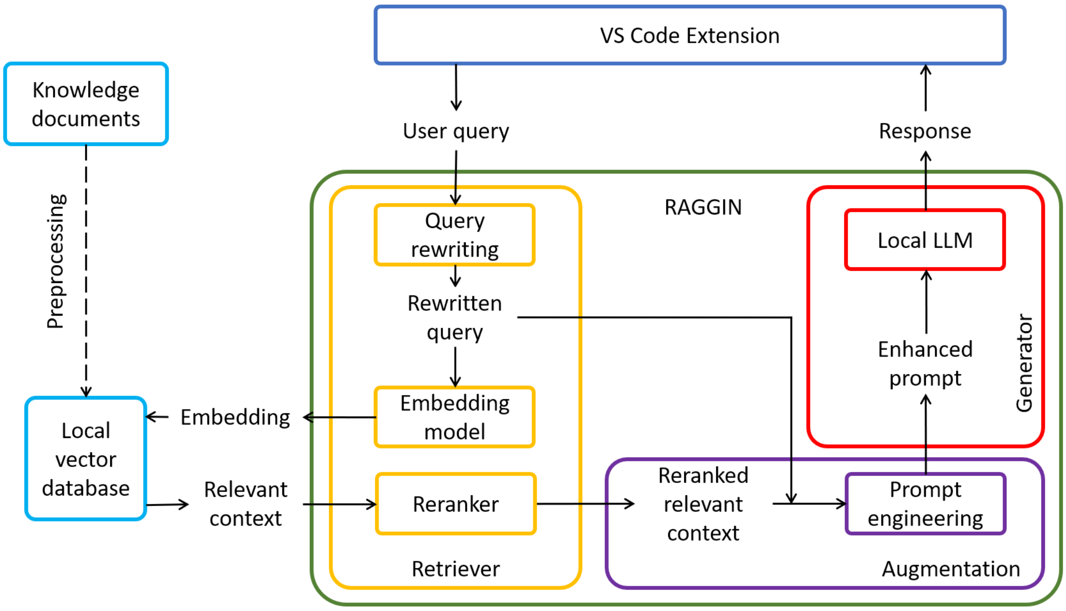

Figure 4.3: RAGGIN architecture design

In this architecture, the input is the user's query, and the output is the corresponding response generated by the system. The response is constructed by utilizing the retrieved data and contextual knowledge extracted from the Next.js documentation, ensuring that it is both accurate and relevant to the user's query.

How the RAGGIN system solves the problems

The proposed RAGGIN system is designed to address the key issues outlined in the problem statement by combining advanced retrieval and generation techniques with seamless integration into the developer workflow. Below is how the system resolves the identified problems:

1. **Time-consuming searches**

Problem: Developers spend excessive time manually searching through forums like StackOverflow or through vast and scattered Next.js documentation to find specific information.

Solutions:

- The retriever component of the system uses a vector database to store Next.js documentation in a semantically indexed format.
- When a user submits a query, the retriever efficiently retrieves the most relevant documents based on semantic similarity
- By narrowing down the search scope and returning only the most contextually relevant sections, the system reduces the time spent on manual searches and ensures higher accuracy in retrieval.

2. **Lack of context-aware assistance**

Problem: Retrieved information often lacks sufficient context, leaving developers to manually piece together fragmented answers.

Solutions:

- The augmentation component combines the retrieved documents with the user's query to construct a prompt enriched with relevant context.
- The generator uses the augmented prompt to produce coherent, actionable, and contextually accurate responses. This eliminates the need for developers to manually correlate information from multiple sources.

3. **Workflow disruption**

Problems: Switching between the code editor and external documentation disrupts the developer's workflow, reducing productivity.

Solutions:

- The system is fully integrated into the VS Code environment, allowing developers to access it without leaving their coding workspace.
- Queries can be sent directly from VS Code, and responses are displayed within the editor. This seamless integration ensures that developers remain focused on their work, minimizing interruptions and enhancing their overall efficiency.

4. **Inconsistent support from existing tools**

Problem: Traditional tools like keyword-based search engines fail to handle nuanced queries or provide tailored, actionable solutions.

Solutions:

- The system leverages advanced semantic retrieval techniques through the vector database to locate relevant content based on the meaning of the query rather than just matching keywords.
- The local LLM and vector database ensure an uninterrupted workflow, even in the event of an internet outage.

4.6.1 Local vector storage

The vector storage component is a fundamental part of our system, enabling efficient retrieval of embedded data for the user. Since our primary dataset consists of documentation from the Next.js framework, it inherently contains a combination of both structured and unstructured data:

- **Structured data:** The Next.js documentation is systematically organized. Each documentation page belongs to a specific subject or category, resembling a hierarchical folder structure. This organization provides metadata that aids in filtering and contextual retrieval.
- **Unstructured data:** The text content and code snippets within the documentation represent unstructured data. These elements lack a predefined schema but carry crucial information necessary for coding assistance.

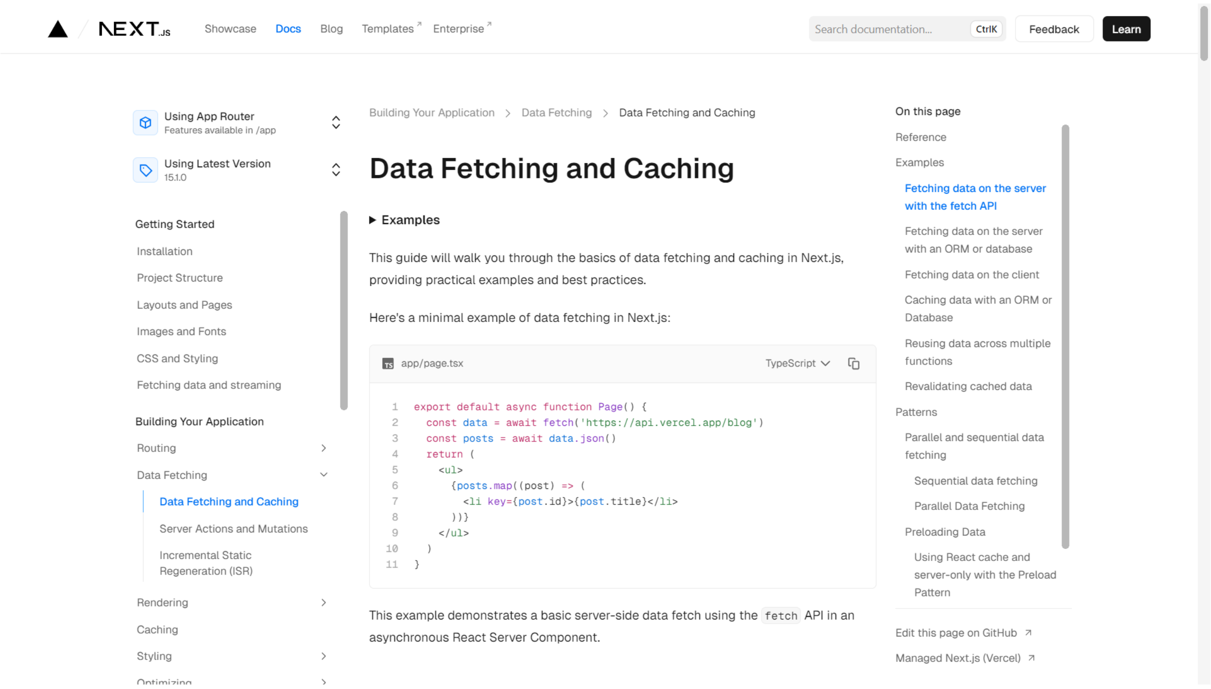

Figure 4.4: Example of a Next.js documentation page

```
---
title: Data Fetching and Caching
nav_title: Data Fetching and Caching
description: Learn best practices for fetching data on the server or
↪  client in Next.js.
---

<details>
<summary>Examples</summary>

- [Next.js
↪  Commerce](https://vercel.com/templates/next.js/nextjs-commerce)
- [On-Demand ISR](https://on-demand-isr.vercel.app)
- [Next.js
↪  Forms](https://github.com/vercel/next.js/tree/canary/examples/next-forms)

</details>

This guide will walk you through the basics of data fetching and
↪  caching in Next.js, providing practical examples and best
↪  practices.

Here's a minimal example of data fetching in Next.js:

```tsx filename="app/page.tsx" switcher
export default async function Page() {
        const data = await fetch('https://api.vercel.app/blog')
        const posts = await data.json()
        return (
        <ul>
        {posts.map((post) => (
                <li key={post.id}>{post.title}</li>
                ))}
        </ul>
        )
}
```

```jsx filename="app/page.js" switcher
export default async function Page() {
        const data = await fetch('https://api.vercel.app/blog')
        const posts = await data.json()
        return (
        <ul>
        {posts.map((post) => (
                <li key={post.id}>{post.title}</li>
                ))}
        </ul>
        )
}
```

This example demonstrates a basic server-side data fetch using the
↪ `fetch` API in an asynchronous React Server Component.
```

Figure 4.5: Markdown representation of a documentation page

Why Milvus?

Milvus is our chosen vector database for storing embedded data in the system. This decision is based on several compelling reasons:

- **Open source:** Milvus is an open-source database, perfectly aligning with our vision of delivering a free, open-source, distributed code assistance tool for VS Code. Its open nature encourages transparency and community contributions.

- **Support for structured and unstructured data:** The nature of our dataset, comprising both structured metadata (e.g., documentation sections, categories) and unstructured content (e.g., textual descriptions, code snippets), makes Milvus an ideal choice. Structured data benefits from attribute-based filtering, allowing users to narrow their queries to specific documentation sections. Meanwhile, unstructured data, such as free-form text and code snippets, is embedded into high-dimensional vectors for similarity-based retrieval.

- **Comprehensive SDK and model support:** Milvus's SDK seamlessly integrates with our chosen embedding model, BGE-M3, for generating high-quality embeddings. It also supports the HNSW algorithm for efficient indexing of unstructured data. This compatibility with our preferred methods ensures smooth implementation and reliable performance.

- **Scalability:** Milvus employs LSM-based storage to minimize disk I/O and ensure efficient handling of frequent updates. This scalability is critical for managing dynamic and evolving datasets, such as continuously updated framework documentation.

- **Performance:** Since our system is designed to run entirely on the user's local machine, Milvus's ability to leverage both CPU and GPU resources significantly enhances performance. This ensures that computationally intensive tasks, such as vector similarity searches and indexing, are executed efficiently, maintaining a responsive user experience.

System	Billion-Scale Data	Dynamic Data	GPU	Attribute Filtering	Multi-Vector Query	Distributed System
Facebook Faiss [13, 25]	✓	✗	✓	✗	✗	✗
Microsoft SPTAG [10]	✓	✗	✓	✗	✗	✓
ElasticSearch [14]	✗	✓	✗	✓	✗	✓
Jingdong Vearch [29, 30]	✓	✓	✓	✓	✗	✓
Alibaba AnalyticDB-V [57]	✓	✓	✗	✗	✗	✓
Alibaba PASE (PostgreSQL) [35]	✗	✓	✗	✓	✗	✓
Milvus [53]	✓	✓	✓	✓	✓	✓

Table 4.3: Vector database comparison

4.6.2 VS Code integration

The interactive user interface is a crucial part of the system, designed to optimize the developer's experience while working. This component leverages the power of VS Code Extension to provide seamless integration between the development tool and the AI assistant.

Seamless experience in the programming environment: Users can interact directly with the RAG system within their working environment using a VS Code Extension. This is convenient because:

- **No context switching**: By eliminating the need for users to alternate between the code editor and web browser, this feature reduces distractions and boosts efficiency.

- **Direct integration into the project**: The RAG system can quickly access the source code being changed thanks to the VS Code plugin, offering solutions or recommendations without forcing users to copy and paste code into a different interface.

Supportive in IDE environment: The most often used IDE environment for web installers is Visual Studio Code, particularly when using more recent frameworks like Next.js. The RAG system's VS Code integration enables:

- **Use code context**: From the Next.js structure folder (which includes pages, api, and app) to the file configuration (next.config.js,.env), the system may directly examine the project's coding**.**

- **Faster debugging and code optimization**: The system may access the coding link to offer more precise answers than the standard web interface information when users need to debug, test logic, or optimize coding.

Security and privacy: When working with source code, especially in sensitive projects, users are often concerned about security issues. A VS Code Extension:

- **Local storage**: Allows the RAG system to operate locally on the user's machine (when using an internal RAG model such as LLaMA), avoiding the need to upload source code to an external server.

- **Limit information leakage**: With extensions, source code and sensitive data do not need to be transferred to a third browser or server, reducing the risk of information disclosure.

5 Implementation

This chapter provides a detailed explanation of our implementation strategy based on the outlined approach. The implemented solution consists of three core components: the Database, the RAGGIN core service, and the VSCode extension.

5.1 Database

5.1.1 Data Preparation

We prepared documentation data for Next.js starting from version `v13.0.0` onwards. At the time of writing, the latest version is `v15.3.1`, encompassing a total of 107 supported versions. The data was sourced directly from the official Next.js GitHub repository.

Steps involved in data preparation:

- **Data Acquisition:**A Python script leveraging Git's `sparse-checkout` feature was developed to efficiently download only relevant directories, such as `docs` and `errors`, corresponding to specific version tags. This script automates repository cloning, sparse-checkout configuration, and version-specific checkout.
- **Data Organization:**Retrieved files were systematically organized into directories structured by version. For example, documentation for version `v15.1.0` was organized as follows:

```
./v15.1.0/
    docs/
        01-app/
        index.mdx
        ...
    errors/
        404-get-initial-props.mdx
        amp-bind-jsx-alt.mdx
        ...
```

Upon reviewing the markdown files, we identified several main elements:

- **Metadata:** Essential metadata includes:
  - `title`: Title of the documentation page.
  - `nav_title`: Title displayed in the navigation bar.
  - `description`: Concise summary of the content.
  - `related`: Optional links to related documentation pages.
- **Text Content:** Main documentation content organized into sections and subsections via markdown syntax.

- **Code Snippets:** Each snippet includes:
  - `language`: Programming language (e.g., JavaScript or TypeScript).
  - `file`: Corresponding filename or component name following Next.js conventions.
  - `switcher`: Indicates if an alternative language snippet is available (e.g., JavaScript/TypeScript).
  - `content`: Actual source code.
- **Version History:** A dedicated section describing significant documentation updates across versions, although available in only a limited subset of documents.

After completing data preparation, the quantity of markdown files per version ranged significantly, from 294 files for `v13.0.0` to 591 files for `v15.3.1`. This increase nearly doubled the amount of documentation and documented errors, highlighting substantial growth in the capabilities and complexity of the Next.js framework.

5.1.2 Data Preprocessing

After acquiring and organizing the data, the next critical step was preprocessing it to ensure alignment with our schema design for optimal storage and retrieval. This phase transformed raw markdown files into structured data via a Python-based preprocessing pipeline, leveraging markdown parsing libraries, embedding models, and structured data processing techniques.

Key preprocessing steps

- **Metadata Extraction:** Metadata fields such as `title`, `description`, and `nav_title` were extracted using the `frontmatter` library. Additional metadata, including `related` links when available, were also captured. Unique `entry_id`s were generated for hierarchical referencing within the database schema.

  Example code for generating a unique entry ID:

```python
    def generate_entry_id(file_path):
    relative_path = os.path.relpath(file_path)
    entry_id = re.sub(r'[^a-zA-Z0-9]', '_', relative_path)
    return entry_id.lower()

```

Listing 5.1: Generating unique entry IDs

- **Markdown parsing:** Markdown files were parsed into components such as headings, paragraphs, lists, and code blocks using the `mistletoe` library. The root token, `Document`, represented the entire markdown file, while specific tokens such as `Heading`, `Paragraph`, and `List` enabled detailed content extraction while maintaining the document's hierarchy.
- **Content extraction:** Each token's child elements were recursively processed to capture plain text, inline code, and hyperlinks. For example, `RawText` tokens represented basic text, `InlineCode` tokens identified inline code snippets, and `Link` tokens extracted hyperlink text and URLs.
- **Section segmentation:** Logical sections were created based on `Heading` tokens in the markdown. Each section included a title, derived from the heading, and grouped content such as paragraphs, lists, or code blocks. This segmentation maintained the hierarchical structure necessary for efficient data retrieval.
- **Code snippet identification:** Code blocks were identified using `CodeFence` tokens and extracted along with relevant metadata, including the programming language, associated filename, and optional switcher flags. This information linked code snippets to their respective sections and allowed for language-specific queries.
- **Embedding preparation:** Text and code content from each section were processed to prepare for embedding generation. Sparse embeddings were used for keyword indexing, while dense embeddings captured semantic meaning, enabling similarity-based queries.

The parse_markdown_sections function, implemented using mistletoe, played a central role in these steps by iterating over the document tokens to extract structured information. For instance, a document with headings, paragraphs, and code blocks was transformed into hierarchical sections with associated metadata and content.

Example of parsing and organizing code snippets:

```
if isinstance(token, CodeFence):
code_info = {
  "code": token.children[0].content if token.children else "",
  "language": token.language or "text",
  "filename": filename,
  "switcher": switcher
}
```

Listing 5.2: Code snippet parsing

- **Embedding generation:** Dense and sparse embeddings were generated for both text and code content using a pre-trained embedding model (BGEM3EmbeddingFunction). Dense vectors captured semantic meaning, while sparse vectors indexed key terms for efficient retrieval.
- **Storage:** Processed entries were saved in batches as CSV files, with fields such as entry_id, text_content, and embeddings.

The preprocessing resulted in 107 CSV files, publicly accessible via our dataset NextJs Documentation for RAGGIN, achieving a usability score of 9.41/10 on Kaggle. Dataset insights include:

- **Dataset Size:** From 15.53 MB for version v13.0.0 up to 49.97 MB for version v15.3.1.
- **Entry Count:** Ranges from 396 entries for v13.0.0 to 1,381 entries for v15.3.1.

5.1.3 Database design

Before presenting our schema design, we detail the properties utilized within our fields.

Properties	Description	Note
name	Name of the field in the collection to create	Data type: String. Mandatory
dtype	Data type of the field	Mandatory
description	Description of the field	Data type: String. Optional
is_primary	Whether to set the field as the primary key field or not	Data type: Boolean (true or false). Mandatory for the primary key field
auto_id	Switch to enable or disable automatic ID (primary key) allocation	true or false. Mandatory for primary key field
max_length	Maximum byte length for strings allowed to be inserted.	[1, 65,535]. Mandatory for VARCHAR field. Multibyte characters may occupy more than one byte
dim	Dimension of the vector	Data type: Integer [1, 32,768]. Mandatory for dense vector field. Omit for sparse vector field
is_partition_key	Whether this field is a partition-key field	Data type: Boolean (true or false)

Table 5.1: Field schema properties(Manage Schema[1])

We developed a schema optimized for organizing and storing documentation data. The schema is implemented within a single collection named nextjs_docs.

[1] https://milvus.io/docs/schema.md#Manage-Schema

Field	dtype	Len / Dim	Notes / Description
`entry_id`	VARCHAR	255	**Primary key**. auto_id = false. Built by concatenating version + doc name + section name.
`title`	VARCHAR	255	Built from documentation name, description, and section name.
`metadata`	JSON	–	JSON blob of title, description, and related links.
`version`	VARCHAR	20	**Partition key**. Indicates documentation version.
`tag`	VARCHAR	255	Labels source (documentation vs. documented errors).
`text_content`	VARCHAR	65 535	Markdown body.
`code_content`	VARCHAR	65 535	JSON-stringified dict of code snippets (language, file name, switcher).
`sparse_title`	SPARSE_FLOAT_VECTOR	–	Sparse embedding of concatenated title + description. *Index*: SPARSE_INVERTED_INDEX; *Metric*: Inner Product.
`dense_text_content`	FLOAT_VECTOR	1024	Dense embedding of full text. *Index*: HNSW; *Metric*: Cosine.
`dense_code_snippet`	FLOAT_VECTOR	1024	Avg-pooled vector over all snippet embeddings. *Index*: HNSW; *Metric*: Cosine.

Table 5.2: Schema Structure

When designing the schema, it is crucial to determine how to store code snippets effectively while facilitating the identification of which snippet belongs to which section. Our proposal is to avoid saving code snippets as separate entries in the database in order to reduce latency and computational overhead.

Given this decision, a challenge arises: how should we store embeddings of code snippets, especially since some sections contain none, one, or multiple code snippets? Additionally, for sections with multiple snippets, we must define an efficient storage method.

Figure 5.1 illustrates the distribution of code snippets across sections from version `v15.1.4` to `v15.3.1`. It shows that while some sections have multiple snippets, others have none, reinforcing the need for a flexible storage strategy.

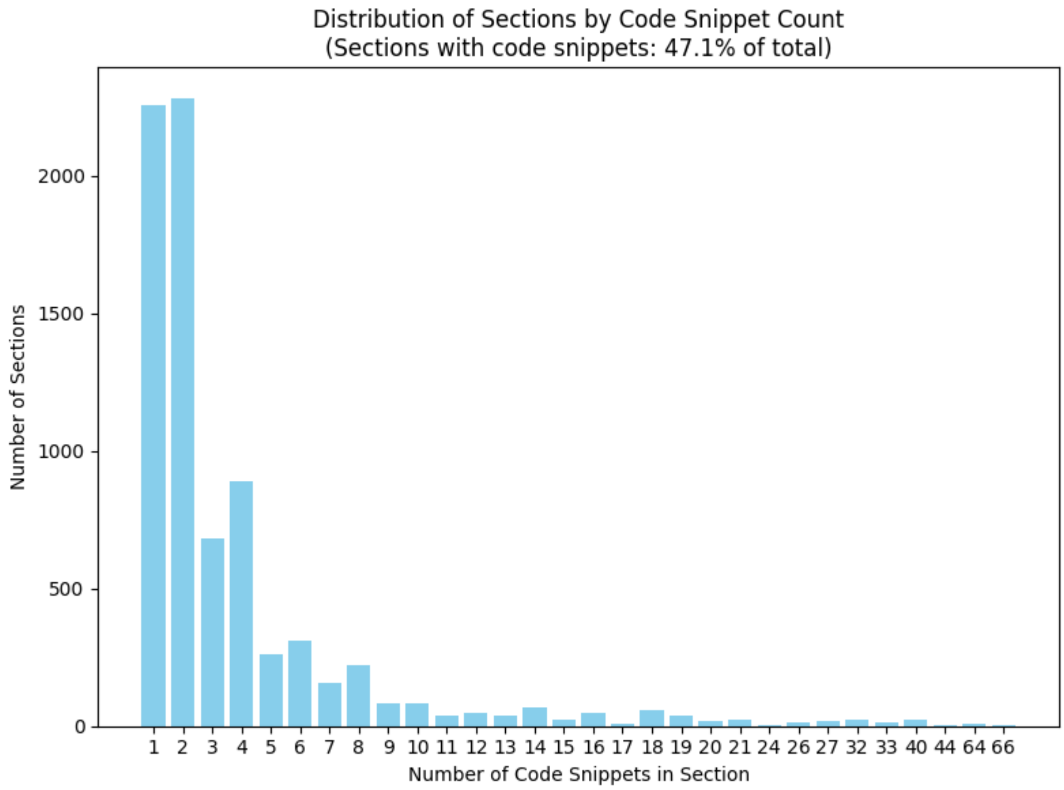

Figure 5.1: Distribution of Code Snippets per Section (`v15.1.4` to `v15.3.1`)

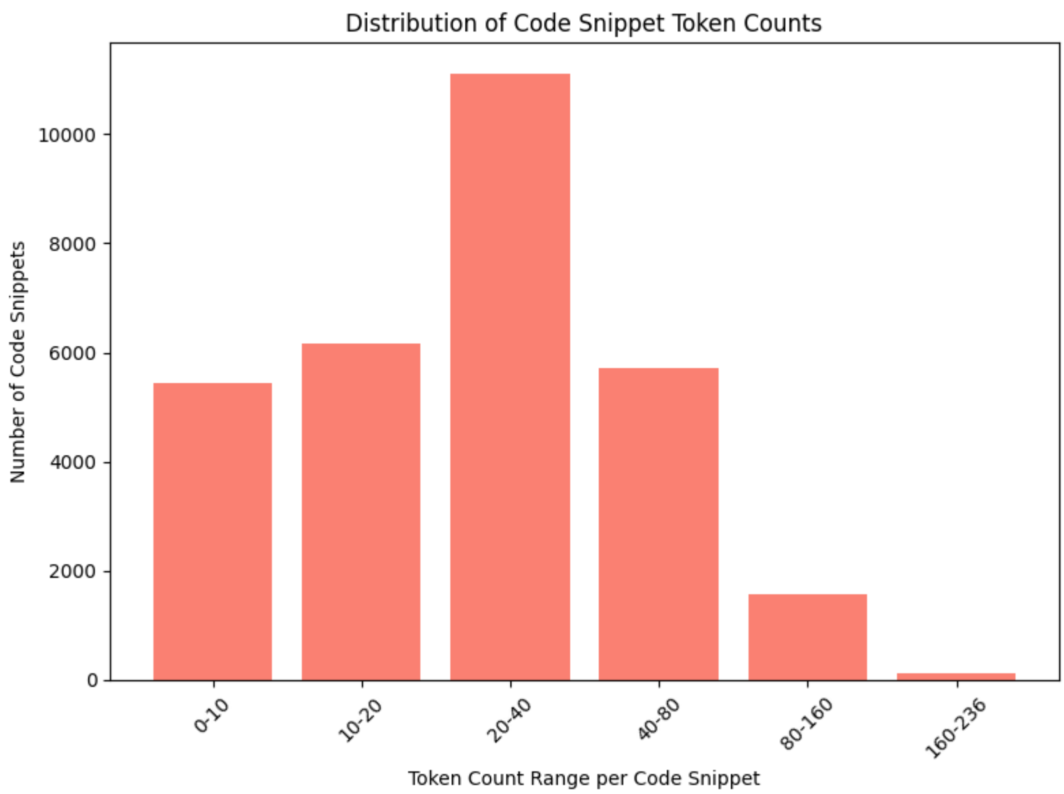

Figure 5.2: Token Count Distribution of Code Snippets (`v15.1.4` to `v15.3.1`)

There are two primary methods for embedding code snippets:

- **Separate Storage:** Each code snippet embedding is stored individually within an array field associated with its section entry.

- **Aggregated Storage:** Embeddings of all code snippets within a section are aggregated into a single vector.

The separate storage method enables precise comparison between a query and individual code snippets. However, it suffers from major drawbacks:

- Additional complexity in indexing entries, as the number of code snippets per section varies.
- Increased database size and computational cost, particularly problematic for large-scale retrieval tasks.

Thus, **aggregated storage** is the preferred option for efficiency.
To unify embeddings into a single vector per section, two aggregation methods are considered:

- **Average Pooling:** Element-wise averaging of all embeddings.
- **Max Pooling:** Element-wise maximum of all embeddings.

To evaluate which aggregation method performs better, we utilize Normalized Discounted Cumulative Gain (NDCG) as the ranking metric.
Figures 5.3 and 5.4 present the NDCG performance of average pooling and max pooling, respectively, evaluated under both dense code search and hybrid search scenarios.

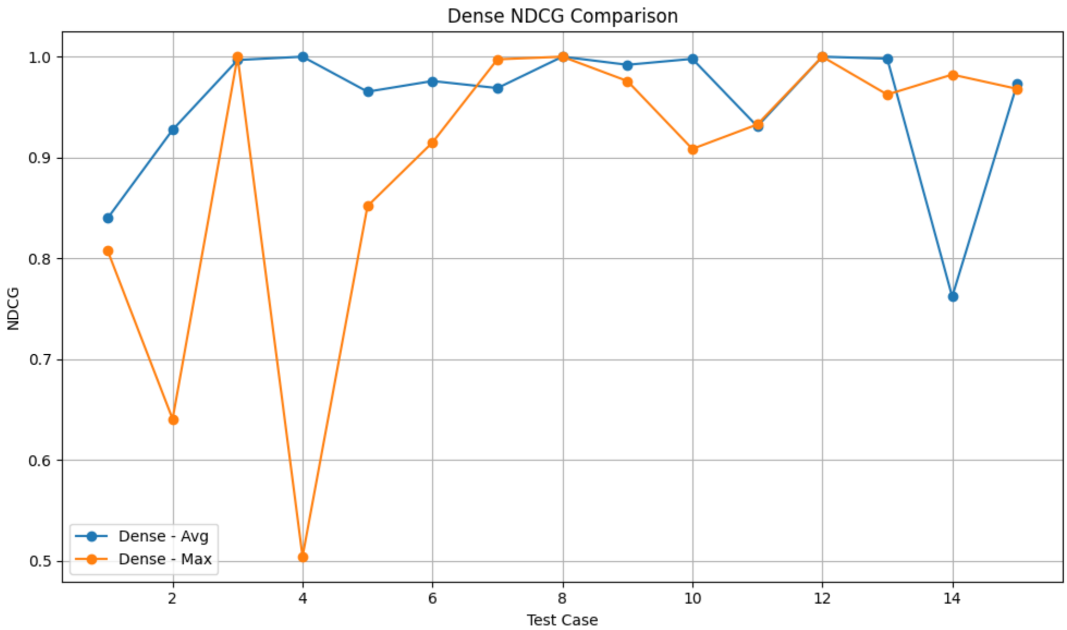

Figure 5.3: NDCG Results with Average Pooling

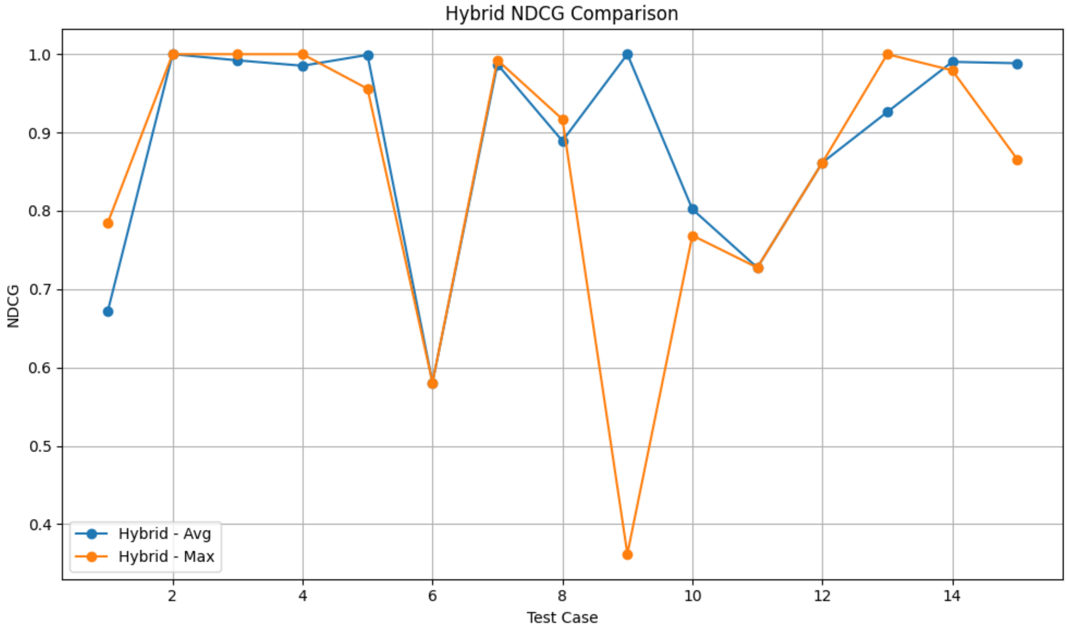

Figure 5.4: NDCG Results with Max Pooling

The evaluation results indicate that average pooling delivers more stable and reliable retrieval performance than max pooling, particularly in maintaining consistent ranking quality.

The retrieved document structure includes a text body containing special placeholder tokens in the form of `code_snippet_{idx}`. During post-processing, each placeholder is replaced by the corresponding code snippet from the `code_content` dictionary.

5.1.4 Database setup

To efficiently handle storage, retrieval, and indexing, we set up our database using Milvus, leveraging Docker Compose for streamlined deployment and configuration. This database setup involved three key services:

- **etcd Service:** We utilized the `etcd` container to manage metadata storage and configuration. It uses the CoreOS etcd image version `v3.5.16`. Configuration settings included client URL advertisement, client listening URLs, data directory specification, auto-compaction, and backend quota management to ensure efficient and stable operation.
- **MinIO Service:** The `minio` container was configured for object storage, essential for storing large data objects. The MinIO server uses a specific release version and provides object storage capabilities with secure access via defined credentials. It exposes two ports: one for the main service (port `9000`) and another for the console interface (port `9001`).
- **Milvus Service (Standalone):** The `milvus-standalone` service, using Milvus version `v2.5.4`, was established for vector indexing and retrieval operations. This container depends on the healthy states of both `etcd` and `minio` services, integrating with them via their respective endpoints. It handles incoming requests and provides monitoring via specified ports.

5.1.5 Database API

The Database API manages documentation data in the Milvus database through seven primary endpoints:

- **List All Supported Versions** `[GET /data/versions]`: Returns a list of all supported Next.js versions with download status.

```
[
{"version_name": "v13.0.0", "downloaded": true},
{"version_name": "v13.0.1", "downloaded": false},
```

```
    ...
    {"version_name": "v15.3.1", "downloaded": true}
    ]
```

Listing 5.3: GET /data/versions output example

- **Retrieve Version Details** `[GET /data/versions/{version}]`: Fetches metadata (file size, last modified time) for a specific version. Returns 404 if unsupported.

```
    {
      "version_name": "v15.3.1",
      "downloaded": true,
      "file_size": 49966271,
      "last_modified": 1745841921.8775475
    ]
```

Listing 5.4: GET /data/versions/version output example

- **List Downloaded Versions** `[GET /data/downloaded]`: Returns a list of all versions downloaded locally.

```
    [
    "v13.0.0",
    "v15.1.2",
    "v15.3.1"
    ]
```

Listing 5.5: GET /data/downloaded output example

- **Retrieve Version Statistics** `[GET /data/stats]`: Provides statistics including total supported and downloaded versions.

```
    {
      "total_supported": 107,
      "total_downloaded": 3,
      "downloaded_versions": [
      "v13.0.0",
      "v15.1.2",
      "v15.3.1"
      ]
    }
```

Listing 5.6: GET /data/stats output example

- **Retrieve Documentation Version** `[POST /version/retrieve]`: Downloads a version from Kaggle (if absent) and inserts it into Milvus with optional indexing parameters.

  Request parameters:

  - `version_name`: Target version (e.g., `v15.0.0`).
  - `m_text` (optional, default = 16): HNSW "M" for text embeddings.
  - `ef_text` (optional, default = 200): HNSW "efConstruction" for text embeddings.
  - `m_code` (optional, default = 16): HNSW "M" for code embeddings.
  - `ef_code` (optional, default = 200): HNSW "efConstruction" for code embeddings.

  Processing steps:

  1. Check if CSV exists locally.
  2. Download CSV from Kaggle if missing.
  3. Create Milvus collection if not exists; apply index parameters (`M`, `efConstruction`) specific to the version.
     - `M`: Controls graph connectivity in HNSW; higher values improve accuracy but use more memory.
     - `efConstruction`: Controls the candidate pool size during indexing; larger values increase precision at indexing cost.

4. Insert parsed CSV data into Milvus.

- **Delete Documentation Version** `[DELETE /version/delete]`: Deletes a version from both local storage and Milvus.

  Request parameters:

  – `version_name`: Target version.
  – Optional indexing parameters (`m_text`, `ef_text`, `m_code`, `ef_code`).

  Processing steps:

  1. Remove local CSV and associated Milvus entries.
  2. Download fresh CSV from Kaggle.
  3. Recreate Milvus indices using specified or default `M` and `efConstruction` values.
  4. Reinsert refreshed data into Milvus.

- **Repair Documentation Version** `[POST /version/repair]`: Repairs a version by deleting old data and reimporting it fresh.

  Request parameters:

  – `version_name`: Target version.

  Processing steps:

  1. Check if CSV exists locally.
  2. Remove associated entries from Milvus.
  3. Delete the local CSV file.

5.2 RAGGIN core services

The RAGGIN core service provides the API functionalities for hybrid retrieval and generation. It is deployed as a containerized application, built from a custom Dockerfile and exposed through port 8000. The backend interacts directly with the Milvus vector database via a specified internal URI, handling retrieval queries and database operations. It also interfaces with an external Ollama API endpoint to access local LLMs during generation. Pretrained embedding models, such as BGE-M3, are cached within a dedicated volume to reduce startup latency and accelerate embedding generation. The backend depends on the Milvus standalone service for healthy initialization, ensuring database availability before serving requests. Volumes are mounted to persist downloaded documentation files and model checkpoints across restarts, allowing efficient reuse of data. This service acts as the main processing layer connecting the vector storage, retrieval logic, and language model inference.

5.2.1 Retriever

The Retriever service performs hybrid searches by combining sparse lexical matching and dense vector similarity using BGE-M3 embeddings. It leverages Milvus for efficient retrieval and supports weighted multi-modal queries.

Hybrid Search Endpoint `[POST /search]`
Executes hybrid searches across sparse and dense (text and code) embeddings, returning the top-`k` most relevant results.
Request parameters:

- `text_query`: Text query for semantic matching.
- `code_query`: Code query for snippet matching.
- `version_name`: Version filter (e.g., `v15.0.0`).
- `sparse_weight` (optional, default = 1.0): Weight for sparse embeddings.
- `dense_text_weight` (optional, default = 1.0): Weight for dense text embeddings.

- `dense_code_weight` (optional, default = 1.0): Weight for dense code embeddings.
- `top_k` (optional, default = 5): Number of results to return.
- `filter_expr` (optional): Additional filter expression[2] for Milvus.
- `radius_sparse`, `radius_dense_text`, `radius_dense_code` (optional, default = 0.5): Search radius[3] for each modality.
- `range_sparse`, `range_dense_text`, `range_dense_code` (optional, default = 0.5): Search range[4] for each modality.

Processing steps:

1. Validate input: Ensure at least one modality weight is nonzero; prepare queries and filters.
2. Generate embeddings: Produce sparse and dense vectors using BGE-M3.
3. Search: Query Milvus separately for each modality (sparse, dense text, dense code).
4. Merge results: Raw similarity distances returned by Milvus are mapped to the range $[0, 1]$ to ensure comparability across modalities. The normalization function is defined as:

$$\text{normalized\_score} = \frac{\frac{\pi}{2} - \arctan(\text{distance})}{\frac{\pi}{2}}$$

   where:

   - A distance close to 0 maps to a score close to 1 (indicating high relevance).
   - A distance approaching $+\infty$ maps to a score close to 0 (indicating low relevance).
   - The output is clamped to $[0, 1]$ to avoid overflow or underflow.

   This mapping ensures that smaller distances (closer matches) correspond to higher normalized scores.

   After normalization, scores from different modalities (sparse, dense text, dense code) are aggregated into a single combined score using a weighted sum:

$$\text{combined\_score} = w_{\text{sparse}} \times s_{\text{sparse}} + w_{\text{dense\_text}} \times s_{\text{dense\_text}} + w_{\text{dense\_code}} \times s_{\text{dense\_code}}$$

   where:

   - $w_{\text{sparse}}, w_{\text{dense\_text}}, w_{\text{dense\_code}}$ are the user-defined weights for each modality (default 1.0).
   - $s_{\text{sparse}}, s_{\text{dense\_text}}, s_{\text{dense\_code}}$ are the normalized scores from the respective search results.
5. Select top-k: Rank by hybrid scores and return top results.

Example query:

```json
{
  "text_query": "Is this command using the canary version?",
  "code_query": "npx create-next-app@latest",
  "version_name": "v15.1.2",
  "sparse_weight": 0.5,
  "dense_text_weight": 1.0,
  "dense_code_weight": 0.9,
  "top_k": 3,
  "filter_expr": "tag == 'documentation'",
  "radius_sparse": 0.08,
  "range_sparse": 1,
  "radius_dense_text": 0.6,
  "range_dense_text": 1,
  "radius_dense_code": 0.6,
  "range_dense_code": 1
}
```

Listing 5.7: POST /search JSON payload example

[2]Filtered Search - `https://milvus.io/docs/filtered-search.md`
[3]Range Search - `https://milvus.io/docs/range-search.md`
[4]Range Search - `https://milvus.io/docs/range-search.md`

Example response:

```json
{
  "results": [
  {
    "title": "02-canary - Upgrade your Next.js Application to canary",
    "metadata": {},
    "version": "v15.3.1",
    "text_content": "...",
    "code_content": "...",
    "tag": "documentation",
    "sparse_distance": 0.115,
    "dense_text_distance": 0.602,
    "dense_code_distance": null,
    "combined_score": 0.745
  },
  {
    "title": "create-next-app - Create Next.js apps",
    "metadata": {},
    "version": "v15.3.1",
    "text_content": "...",
    "code_content": "...",
    "sparse_distance": 0.080,
    "dense_code_distance": 0.901,
    "combined_score": 0.681
  },
  {
    "title": "12-upgrading - Learn how to upgrade Next.js",
    "metadata": {},
    "version": "v15.3.1",
    "text_content": "...",
    "code_content": "...",
    "dense_text_distance": 0.664,
    "combined_score": 0.626
  }
  ]
}
```

Listing 5.8: POST /search output example

5.2.2 Generator

Local LLMs setup

In the development of RAG systems, the choice of a robust and secure LLM is critical for achieving high-quality responses and ensuring data privacy. To support our RAG pipeline, we installed and configured Ollama[5], a platform specifically designed to facilitate the deployment of pre-trained language models on local infrastructure. This section describes the setup and integration of Ollama into our system.

Setup Ollama

The setup process begins by installing the Ollama application, which is compatible with various operating systems including Windows, macOS, and Linux. Once installed, Ollama provides a straightforward interface to download and configure pre-trained models without requiring fine-tuning.

The installation of specific models is conducted via the command prompt or terminal, with available models conveniently listed on `https://ollama.com/search`. The command to install a model is `ollama pull <model_name>`.

Once a model is installed, it is stored locally, ensuring a high level of data privacy and security — an essential feature for applications dealing with sensitive or proprietary information. To test the model, we can use the command `ollama run <model_name>` .

Furthermore, the platform's lightweight design enhances compatibility across a variety of hardware configurations, making it particularly suitable for environments with limited computational resources. This flexibility makes Ollama an optimal choice for deploying local LLMs in diverse use cases.

Integration with the RAG system

To integrate Ollama into the RAG workflow, we use the Ollama API for direct interaction with the locally installed models. The endpoint for the API is:

POST `http://localhost:11434/api/generate`

[5] Ollama - `https://ollama.com/`

The POST request body should be in JSON format, specifying the desired model and the prompt to be processed. An example request body is shown below:

```
{
  "model": "qwen",
  "prompt": "What is RAG?"
}
```

Listing 5.9: Ollama JSON payload example

The API provides a straightforward and efficient method for passing queries and receiving responses from the model.

Generator Endpoint `[POST /prompt/generate]`

When RAGGIN Generator receives the generate request, it will take the query, model, and file list and process them to RAGGIN Retriever to retrieve relevant documents. Then, it will take the retrieved documents, together with the query and process them to Ollama LLM to generate the final response.

Request parameters:

- `version_name`: Version filter (e.g. `v15.3.1`).
- `query`: The input query or prompt that serves as the basis for the system to retrieve relevant information and generate a response.
- `model`: The name of the Ollama LLM selected for generating the response.
- `file_list` (Optional): A collection of `FileModel` instances, where each instance includes the following string fields: `fileName`, `fileExtension`, and `fileContent`.
- `additional_options` (Optional): An optional field comprising two subfields: `retriever_options` — the configurable parameters associated with the Hybrid Search Endpoint — and `generator_options` — the set of options controlling the language model's generation behavior. These settings enable users to customize the system's retrieval and generation phases for enhanced flexibility and performance.

Processing steps:

1. Retrieve documents: Use input prompt and version name to retrieve relevance documents.
2. Enhance prompt: Construct an enhanced prompt using the original prompt with the retrieved documents.
3. Generate: Send a POST request to the Ollama API endpoint, then receive and return the generated response.

The generate request's body should be in JSON format, specifying the desired model and the prompt to be processed. An example request body is shown below:

```
{
    "version_name": "v15.1.3",
    "query": "refactor the files to make it more readable.",
    "model": "llama2:13b",
    "additional_options": {
        "retriever_options": {
          "top_k": 5
        },
        "generator_options": {
          "temperature": 0.8
        }
    },
    "file_list": [
        {
        "file_name": "/my_page.tsx",
        "file_extension": "tsx",
        "file_content": "import { File, UserRound, LogOut } from \"lucide-react\"; ..."
        }
    ]
}
```

Listing 5.10: POST /prompt/generate JSON payload example

The response body will contain the generated response and some other relevant information in JSON format.

```
{
    "model": "llama2:13b",
    "response": "Sure! Here are some suggestions for refactoring the code to make it more
    readable ..."
    ...
}
```

Listing 5.11: POST /prompt/generate output example

5.3 RAGGIN VSCode Extension Development

The RAGGIN (Retrieval-Augmented Generation for Guided Intelligence in Next.js) Visual Studio Code extension is purpose-built to elevate developer productivity within the Next.js ecosystem. RAGGIN operates entirely on the local machine, leveraging Docker and Ollama to remove any reliance on an active internet connection. At its foundation, the extension seamlessly integrates several advanced technologies: local Large Language Models (LLMs) via Ollama for rapid and context-aware response generation; Retrieval-Augmented Generation (RAG) to supplement AI outputs with precise, project-relevant documentation; Docker containers to orchestrate retrieval and computation processes; and the VS Code extension API, which embeds these capabilities natively within the editor. Real-time communication between the VS Code webview and the extension host is achieved through efficient message passing, ensuring a responsive and interactive AI experience.

5.3.1 Installing in VSCode (Private installation)

Install from the VS Code Marketplace

- In your VS Code IDE, go to **Extensions**.
- Search for **Raggin (Self-Hosted)** and select it.
- Install the extension.

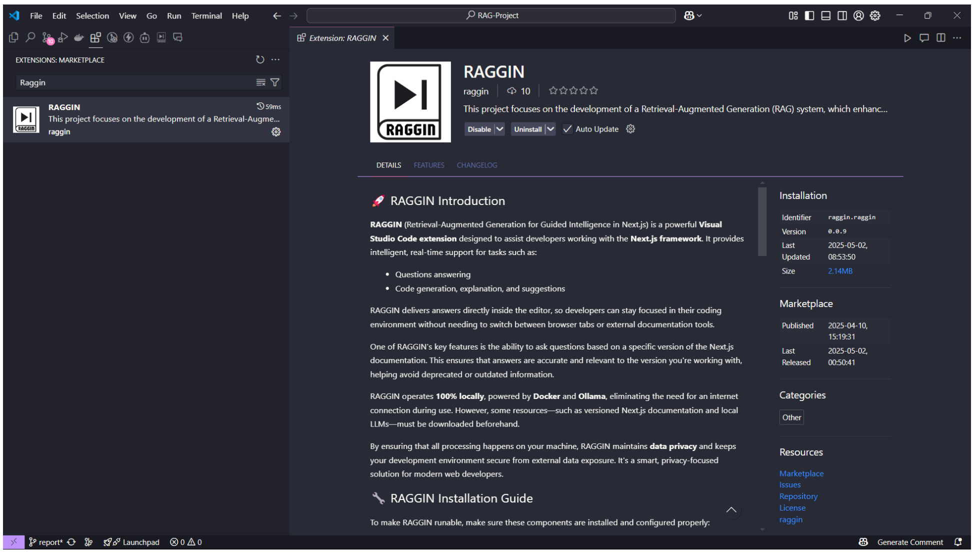

Figure 5.5: Installing Raggin extension in VS Code

You don't need to log in anyway because this is private for user own using.

5.3.2 Icons & labels

Chat page header

The top navigation bar of the IDE plugin displays the selected model (e.g., qwen2.5-coder) and its corresponding version (e.g., `v15.3.0`). To the right, users have quick access to Settings, the option to Add a New Chat, and Chat History, streamlining workflow for configuration, testing, and session management.

Figure 5.6: Chat page header

Chat input

This interface includes a file-plus icon on the left, which allows users to upload files from their code base. Once the file is uploaded, it will appear on the left side of the input field, just above the text box, indicating that the file is successfully attached and ready to be referenced or processed within the current chat session. On the far right, there is a send button (shaped like a cleft arrow) used to submit the message. The submit function becomes active after selecting the appropriate Ollama model, specifying the Next.js version, and completing the prompt input.

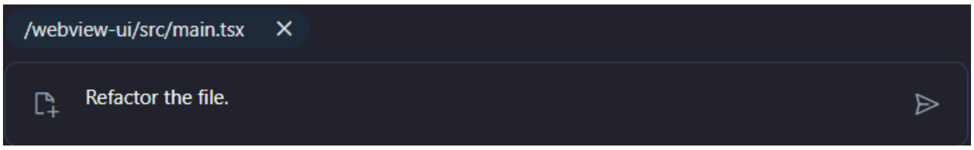

Figure 5.7: Chat input

Added context label

The File Context Label appears prominently beneath the model and version selection bar. It displays the path of the currently active file (e.g., /webview-ui/src/App.tsx), indicating the specific source file that the assistant will use as reference for the current interaction. This context ensures that code-related requests—such as refactoring or debugging—are applied to the correct file within the project structure.

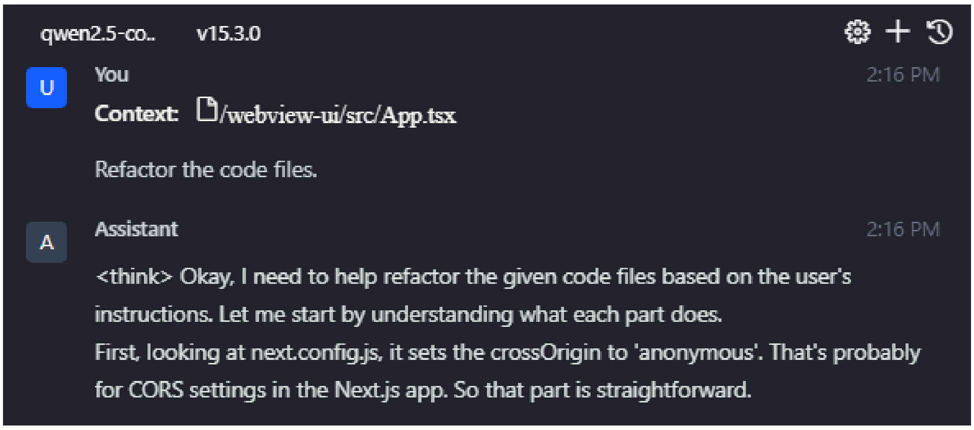

Figure 5.8: Chat input

Chat window

This section of the chat interface displays individual conversation sessions. When a user submits a lengthy question, the interface provides a "Show more / Show less" toggle to optimize readability and minimize vertical scrolling. Responses containing code are rendered using the react-syntax-highlighter library, ensuring clear syntax formatting. Additionally, each code block includes a convenient copy-to-clipboard icon, allowing users to quickly copy the snippet for immediate use in their development workflow.

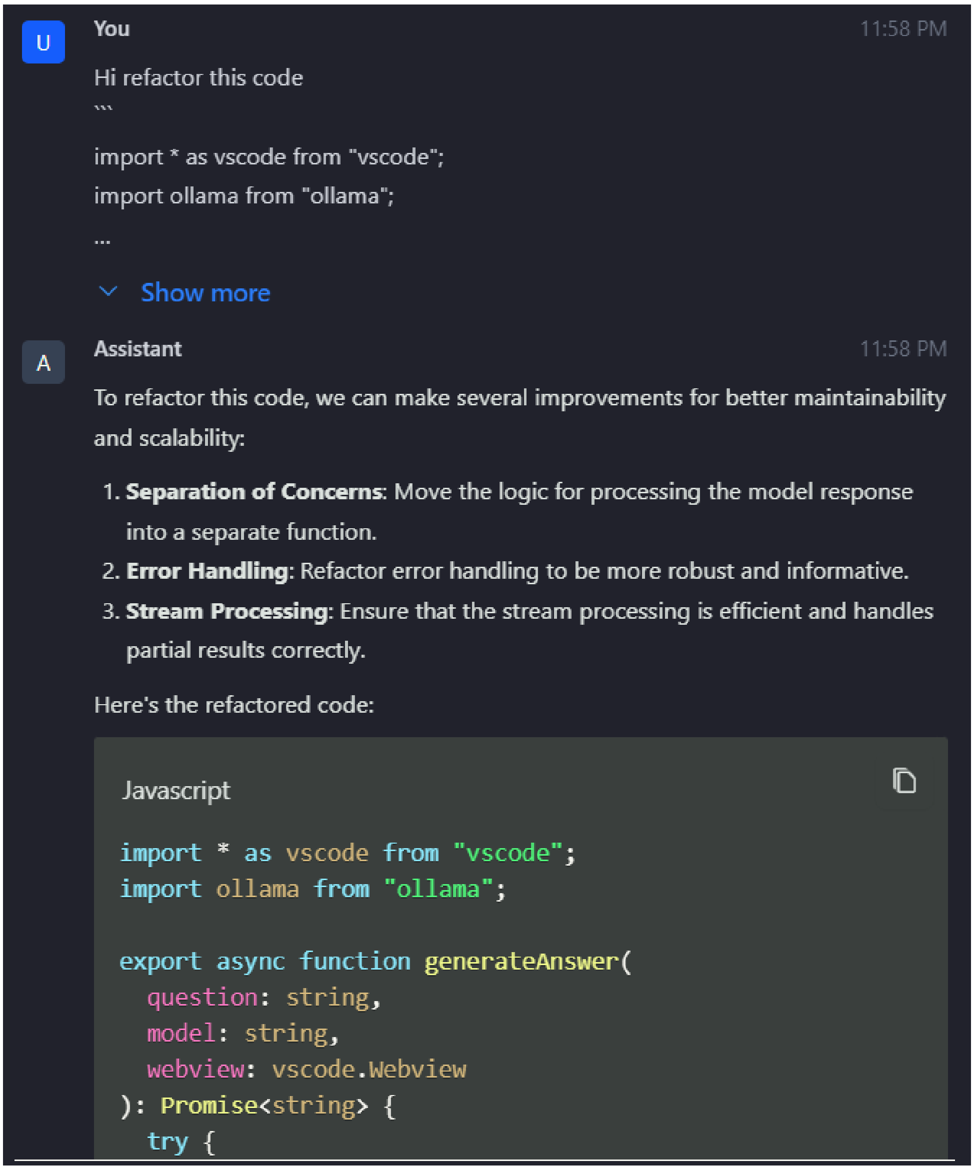

Figure 5.9: Chat input

Chat history header

The Chat History panel serves as a central location for users to access and manage their previous conversation sessions. Positioned at the top of this section is the "Chat History" heading, clearly indicating its purpose. To the right of the heading are two functional icons designed for streamlined interaction. The first icon, represented by a list, allows users to toggle between different view modes or manage existing chat threads more efficiently. Next to it, the clear history icon provides a quick way to delete all saved conversations, helping users maintain a clean workspace.

Figure 5.10: Chat history header

Chat history window

Each chat entry in the chat history panel includes a three-dot settings menu located in the top-right corner of the chat card. This contextual menu provides users with the ability to rename the conversation—allowing for more meaningful and descriptive titles—or to delete the chat entirely when it is no longer needed. These options help users efficiently manage their session history and maintain an organized workspace.

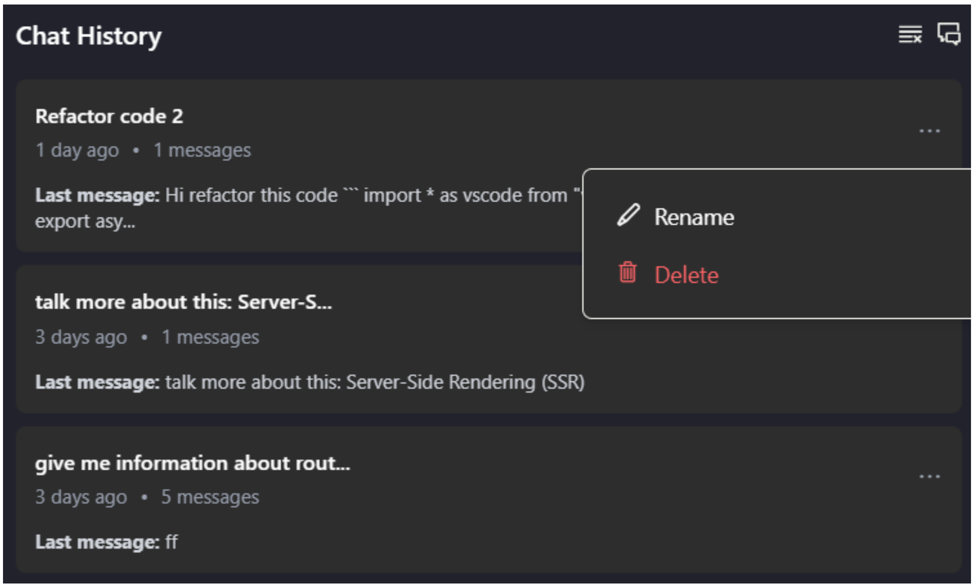

Figure 5.11: Chat history window

Advance settings page

The interface displayed is the Advanced Settings panel, designed to fine-tune parameters for both retrieval and generation in RAG workflow. At the top, there is a tabbed header with two options: Retriever options (currently selected) & Generator options. Each section includes an Instructions link that likely directs users to documentation or usage guidance. At the bottom, users can choose to Cancel changes or Save their configuration using clearly labeled action buttons.

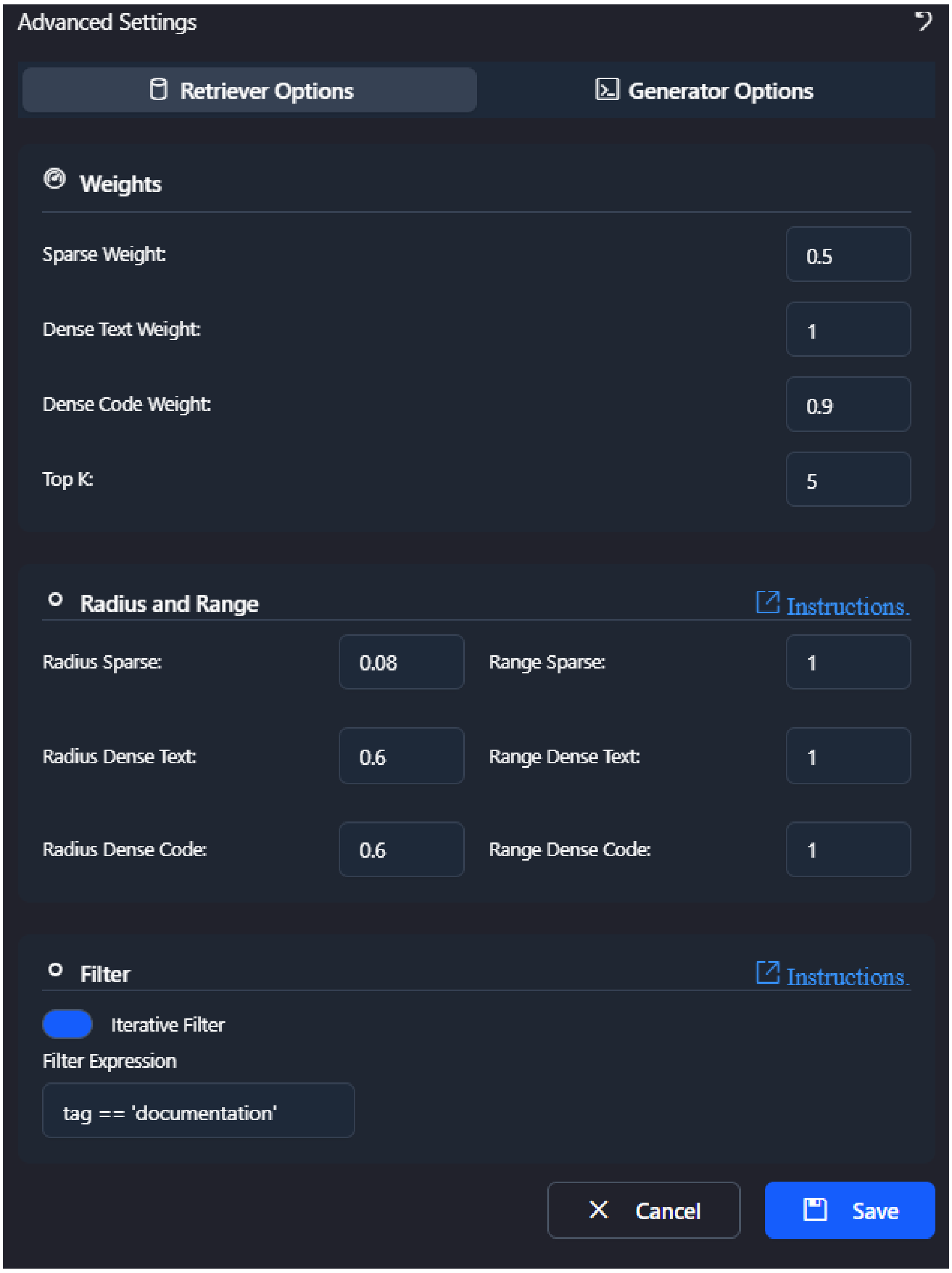

Figure 5.12: Advance settings page

5.3.3 Primary function

RAGGIN is a lightweight, privacy-preserving AI assistant extension built for Visual Studio Code, specifically tailored to support Next.js developers across a wide range of versions. The extension integrates Retrieval-Augmented Generation (RAG) techniques to deliver context-aware, intelligent responses directly within the development environment.

Key features include:

- **Real-time Querying within VSCode:** Developers can interact with the AI assistant directly inside the extension panel. The primary focus is on Next.js-related queries, with dynamic control over the Next.js version (from `v13.0.0` to `v15.3.1`) to ensure framework-specific accuracy. Additionally, users can switch between Ollama-hosted LLM models, offering flexibility in model selection based on task complexity or system capabilities.
- **Local Contextual Intelligence:** Users can load a specific code file from their local project to provide the AI with deeper context. All operations occur within the local development environment, ensuring data privacy and security—no external server calls or remote data exposure

- **Persistent Chat History:** All user interactions are stored locally using VSCode's globalState API, enabling developers to retain conversation history across sessions. The dedicated history view allows quick reference and reuse of past discussions without re-prompting.
- **Advanced Configuration Panel:** For power users and researchers, the Advanced Settings page provides control over key RAG parameters:
  - Retrieval method customization (e.g., top-k, similarity threshold)
  - Generator configuration (e.g., temperature, max tokens) These tunable settings support experimentation and fine-tuning for optimal results.

5.3.4 Performance

Raggin stands out as one of the most lightweight AI-assist extensions on the VSCode Marketplace, with a package size under 2.5MB. Optimized for speed, memory efficiency, and modular development, it consumes minimal system resources—using CPU and memory at levels comparable to or lower than typical VSCode extensions. The extension achieves this efficiency by offloading most computational work to the underlying RAG model via Ollama, ensuring your editor remains responsive even during complex queries. Response quality and speed scale with your hardware capabilities—systems that can run larger LLMs will experience enhanced performance with faster, more accurate answers.

5.3.5 Known Limitations and Future Roadmap

Known Limitations: Despite its current utility, RAGGIN still has areas for growth. Currently, the extension supports contextual referencing for one file at a time, which may restrict its effectiveness in multi-module projects. Additionally, responses are only rendered after full generation, lacking token-by-token streaming, which may result in perceived delays for long outputs.

Planned Enhancements: The development roadmap for RAGGIN includes several key improvements: implementing streaming responses for real-time feedback, upgrading the user interface for smoother interaction and better usability, enabling multi-file context handling to support complex codebases, and enhancing prompt management and template customization for more structured queries.

6 Evaluation and testing

6.1 Evaluation methodology

To evaluate the performance of both the retriever and the generator, we employ a curated dataset containing queries with predefined relevant documents. The evaluation covers two types:

- **Simple queries.** Direct questions that require retrieving clearly defined information from the documentation.
- **Complex queries.** Queries involving multiple components, conditions, or requiring deeper semantic understanding.

The evaluation was conducted using the default parameters for both the retriever and generator components, with the system targeting Next.js version 15.3.1. All experiments were performed on a machine equipped with an Intel Core i5-12400F processor and an NVIDIA RTX 3060 GPU with 12GB of VRAM.

6.2 Evaluation metrics

6.2.1 Retriever metrics

Recall@K. Recall@K quantifies the proportion of relevant documents successfully retrieved among the top K results for a given query.

$$\text{Recall@K} = \frac{|\text{Relevant documents} \cap \text{Top-}K \text{ retrieved documents}|}{|\text{Relevant documents}|}$$

Where:

- $|\text{Relevant documents}|$ denotes the total number of documents relevant to the query.
- $|\text{Relevant documents} \cap \text{Top-}K \text{ retrieved documents}|$ is the count of relevant documents found in the top K results.

Example. A query returns the top $K = 5$ documents. Among them, 3 are relevant, while the total number of relevant documents is 4.

$$\text{Recall@5} = \frac{3}{4} = 0.75 \quad (75\%)$$

Precision@K. Precision@K measures the accuracy of the top K retrieved documents by evaluating how many of them are truly relevant.

$$\text{Precision@K} = \frac{|\text{Relevant documents} \cap \text{Top-}K \text{ retrieved documents}|}{|\text{Top-}K \text{ retrieved documents}|}$$

Where:

- $|\text{Top-}K \text{ retrieved documents}|$ is the number of documents retrieved within the top K results.

Example. A query retrieves the top $K = 5$ documents, with 3 being relevant. The total number of relevant documents is 4.

$$\text{Precision@5} = \frac{3}{5} = 0.6 \quad (60\%)$$

Latency. Latency represents the response time taken to complete a retrieval operation.

$$\text{Latency (ms)} = \text{Time}_{\text{returned}} - \text{Time}_{\text{submitted}}$$

Example. The query is submitted at 11:00:00.000 and the result is received at 11:00:00.340.

$$\text{Latency} = 340\,\text{ms} - 0\,\text{ms} = 340\,\text{ms}$$

6.2.2 Generator metrics

ROUGE Score. ROUGE evaluates the lexical overlap between generated content and a set of reference texts. ROUGE-N specifically computes n-gram overlaps:

$$\text{ROUGE-N} = \frac{\sum_{\text{ref}\in\text{References}} \sum_{\text{ngram}\in\text{Generated}} \text{Count(ngram)}}{\sum_{\text{ref}\in\text{References}} \sum_{\text{ngram}\in\text{ref}} \text{Count(ngram)}}$$

Example. A reference document has 10 bigrams, and the generated output contains 8 matching bigrams.

$$\text{ROUGE-2} = \frac{8}{10} = 0.8$$

Relevance. Relevance measures how well the generated outputs align with the intent of user queries, typically rated manually and averaged across queries.

$$\text{Relevance} = \frac{1}{N} \sum_{i=1}^{N} \text{RelevanceScore}_i$$

Example. Five queries yield relevance scores of $0.9, 0.8, 0.7, 1.0,$ and 0.6.

$$\text{Relevance} = \frac{0.9 + 0.8 + 0.7 + 1.0 + 0.6}{5} = 0.8$$

Latency. Latency in the generation phase denotes the time elapsed between the completion of retrieval and the generation of the final output.

$$\text{Latency (ms)} = \text{Time}_{\text{generated}} - \text{Time}_{\text{retrieved}}$$

Example. The output is generated at 12:00:00.210 after retrieval completes at 12:00:00.000.

$$\text{Latency} = 210\,\text{ms} - 0\,\text{ms} = 210\,\text{ms}$$

6.3 Evaluation summary

6.3.1 Retriever Performance

Table B.1 and Table B.2 present the retriever performance for simple and complex queries, respectively.

Results show that the retriever achieved high Recall@K in simple queries, with a peak of **0.81** at $K = 5$, demonstrating strong coverage of relevant documents. However, performance declined in complex queries due to their nuanced requirements, with Recall@K dropping to **0.24** at $K = 5$. Precision@K also showed a similar pattern, peaking at **0.57** in simple cases but falling below **0.3** for complex ones.

Latency remained reasonable for simple queries (approximately **1.2 seconds**), while complex queries understandably took longer due to the increased search depth and token length, averaging around **3.4 seconds**.

6.3.2 Generator Performance

The generator ROUGE scores across various models are illustrated in Figure B.1, Figure B.2 and Figure B.3, corresponding to ROUGE-1, ROUGE-2 and ROUGE-3 metrics, respectively. Additionally, model-wise relevance evaluations are presented in Figure B.4 and Figure B.5, capturing the perceived quality of generated outputs across different model sizes. Notably:

- **Codestral-22B**[1], **LLaMA2-13B** [51], and **Qwen2.5-14B** [21] consistently achieved top scores, producing highly relevant and fluent outputs across diverse test cases.
- **Mistral-7B** [24], **CodeLLaMA-7B** [43], and **Qwen2.5-3B** [21] also demonstrated balanced performance, with strong semantic accuracy and reliable response generation.
- Smaller models like **Qwen-1.8B** [3] and **CodeGemma-2B** [49] showed limitations in deeper contextual understanding, reflected in lower performance in complex reasoning tasks.

Overall, generation latency remained within practical limits—averaging approximately **6 seconds** for simple prompts and **12 seconds** for complex queries—though these values varied depending on the size and architecture of the underlying language model.

6.4 RAGGIN VS Code Extension testing

With a package size of less than 2.5MB, RAGGIN Extension ensures a fast installation process and minimal overhead on the VS Code environment. The extension offloads computation to backend services such as Dockerized RAG engines or Ollama models, maintaining low CPU and memory usage during operation.

Performance varies depending on hardware configuration and the size of the selected model. However, during real-world usage, there are instances where the model responds with "I don't know" or fails to generate a meaningful answer—particularly when a smaller LLM is selected. In such cases, switching to a larger, more capable model (e.g., `deepseek-r1:8b` [11]) often leads to significantly more accurate and context-aware responses. This highlights the importance of model selection based on the complexity of the task, as well as the value of scalable performance in RAG systems.

Moreover, for the VS Code extension component, we will test RAGGIN's operability through **scenario testing**.

[1] Codestral - `https://mistral.ai/news/codestral`

ID	Component	Test Scenarios	Expected Result	Status
Test01	Ollama model choosing component	Click "select model" button on chat page's header	A list of models (e.g: Llama3.2, deepseek-r1,..) returned.	Pass
Test02	Next.js version choosing component	Click "select version" button on chat page's header	A list of Next.js version downloaded and version available returned.	Pass
Test03	Ollama model & Next.js version choosing component	Choosing the Ollama model or a Next.js version without starting the RAGGIN Docker container or Ollama service	A VS Code notification should appear informing the user that the backend services are not running	Pass
Test04	Chat input form	Check if the send button is enabled only after selecting model, version, and input prompt	Send button appears only when all required fields are filled, and functions correctly on click	Pass
Test05	File upload	Upload any file through context file input	File uploaded name is displayed upper the chat input area and selectable as chat context	Pass
Test06	Chat generation	Enter a prompt and click send	Message is rendered in the chat window and model returns a valid response	Pass
Test07	File Context Integration	Send a prompt with a selected file context	Response content reflects data from the uploaded file	Pass
Test08	Next.js version setting	Change version in advanced settings page and submit prompt	Response differs appropriately between selected versions (e.g., v13 vs v15)	Pass
Test09	Parameter tuning	Modify temperature or top-k parameters before submitting prompt	Model response behavior changes based on parameter values	Pass
Test10	Chat History – Save	Send multiple prompts in a session	All prompts are stored and shown in correct order under History tab	Pass
Test11	Chat history – Rename	Rename an existing chat from the history panel	Title is updated and persists across sessions	Pass
Test12	Chat history – Delete	Delete a chat from history list	Chat is removed permanently and does not reappear after reload	Pass
Test13	Chat history - Delete all chats	Press "delete all chats" button	All chats are removed permanently and does not reappear after reload	Pass
Test13	Global state save	Reload the extension after using it	Previously saved sessions are restored correctly from global/local state	Pass

Table 6.1: Scenario-Based Manual Testing for RAGGIN VS Code Extension

7 Conclusion

This chapter summarizes the accomplishments of the project over two semesters of research, design, and implementation. It also outlines the current limitations and potential directions for future development of the extension.

7.1 Achieved Results

We have developed an intelligent code assistance system tailored for web development, leveraging RAG to reduce hallucination and support integration of up-to-date information. The system is implemented as a Visual Studio Code extension named RAGGIN, Retrieval-Augmented Generation for Guided Intelligence in Next.js.

The project's source code is publicly available on GitHub under the MIT license[1], ensuring full transparency. The backend services[2] have been containerized and published on Docker Hub, while the extension frontend[3] is released on the Visual Studio Code Marketplace. Additionally, the processed Next.js documentation dataset used for retrieval has been made available on Kaggle[4].

We have delivered a system that assists developers in integrating AI into their workflow, offering an open-source solution that grants the freedom to choose their preferred LLMs according to their system requirements. The system provides both deep configurability and flexibility while maintaining seamless integration within their familiar development environment.

7.2 Limitations

At present, the extension focuses exclusively on the Next.js framework, without support for the broader ecosystem of related technologies commonly used alongside it. While we have manually curated the documentation dataset, defined chunking strategies, and designed embedding pipelines specifically for Next.js, the system lacks the ability to automatically identify and process documentation from other frameworks. As a result, the extension cannot yet be scaled automatically to support additional technologies.

Currently, RAGGIN only offers configuration flexibility for the generator model (i.e., the LLM used for response generation). The embedding model for documentation remains fixed, with embeddings precomputed and bundled with the system. Moreover, the extension does not yet support local GPU-based embedding generation for user queries due to the complexities involved in GPU Docker setup and the additional size such support would impose on the container image.

In terms of retrieval performance, our core service exhibits low precision and recall at small Top-K values: Precision@1 is approximately 0.27, and Recall@1 is 0.05. Performance improves only at larger K, indicating the retrieval system is currently not optimized for extracting highly relevant results with minimal context. This limitation negatively impacts downstream LLM performance, as more irrelevant documents are included in the prompt context.

[1] The MIT License — `https://opensource.org/license/mit`
[2] RAGGIN core service — `https://hub.docker.com/repository/docker/melukootto/raggin/general`
[3] RAGGIN Extension — `https://marketplace.visualstudio.com/items/?itemName=raggin.raggin`
[4] Next.js documentation dataset — `https://www.kaggle.com/datasets/jiyujizai/nextjs-documentation-for-raggin`

The full system image occupies roughly 20GB on Docker (17.3GB for the core backend service and 2.7GB for the Milvus database), excluding the language model and Ollama runtime, which challenges our original lightweight deployment goals.

Finally, it is worth noting that RAGGIN is positioned as an open-source alternative to proprietary services like GitHub Copilot[5], which benefits from commercial-scale infrastructure and full integration with the Visual Studio Code ecosystem through its parent company, Microsoft.

7.3 Future development directions

To address the current limitations, we propose several concrete development directions aimed at improving scalability, configurability, performance, and portability of RAGGIN.

Expanding framework support To broaden beyond Next.js, future work will focus on automating the identification and acquisition of documentation from other technologies, such as React[6], Vue[7], or TailwindCSS[8]. This may involve using sitemap parsing, structured data from `schema.org`[9], or scraping via documentation RSS feeds. A modular preprocessing pipeline will also be developed to support customizable chunking and embedding per framework.

Custom embedding model configuration Support for configurable embedding models will be introduced, allowing users to select their preferred models (e.g., BGE-M3, E5, or InstructorXL[10]) during setup. Additionally, GPU-based embedding generation for user queries will be optionally enabled through a separate, GPU-enabled Docker image to prevent bloating the default container.

Improving retrieval accuracy at low Top-K Include fine-tuning the embedding model on task-specific data, introducing learning-to-rank mechanisms for hybrid scoring, and employing lightweight rerankers such as MiniLM[11] or RankT5[12] to refine the top retrieved entries. These methods aim to increase the relevance of results with minimal context.

Reducing system size To meet lightweight deployment goals, we plan to modularize services, separating the backend API, and embedding module into independently deployable containers. Further size reduction can be achieved through multi-stage Docker builds, minimal base images (e.g., Alpine[13]).

Ecosystem integration and tooling To enhance usability and ecosystem compatibility, future development will target integration with the Language Server Protocol (LSP)[14], enabling support beyond Visual Studio Code (e.g., WebStorm[15], Neovim[16]).

These future directions aim to evolve RAGGIN into a robust, extensible, and developer-friendly tool that remains open, transparent, and adaptable to diverse workflows.

[5] GitHub Copilot — `https://github.com/features/copilot`
[6] React — `https://reactjs.org`
[7] Vue — `https://vuejs.org`
[8] Tailwind CSS — `https://tailwindcss.com`
[9] `schema.org` — `https://schema.org`
[10] InstructorXL — `https://huggingface.co/hkunlp/instructor-xl`
[11] MiniLM — `https://huggingface.co/nreimers/MiniLM-L6-H384-uncased`
[12] RankT5 — `https://github.com/nyu-dl/dl4marco-2022`
[13] Alpine Linux — `https://alpinelinux.org`
[14] Language Server Protocol (LSP) — `https://microsoft.github.io/language-server-protocol`
[15] WebStorm — `https://www.jetbrains.com/webstorm`
[16] Neovim — `https://neovim.io`

Bibliography

[1] A. Asai et al. "SELF-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection". In: *arXiv preprint* arXiv:2310.11511v1 (2023). Available at `https://arxiv.org/abs/2310.11511`.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer Normalization". In: *stat* 1050 (2016), p. 21.

[3] Jinze Bai et al. "Qwen Technical Report". In: *arXiv preprint* arXiv:2309.16609 (2023). Available at `https://arxiv.org/abs/2309.16609`.

[4] Erik Bernhardsson. *Approximate Nearest Neighbors Oh Yeah*. Online. Available at `https://github.com/spotify/annoy`.

[5] D. Brown. *Rank-BM25: A Collection of BM25 Algorithms in Python*. Online. Available at `https://doi.org/10.5281/zenodo.4520057`. 2020.

[6] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: `2005.14165 [cs.CL]`. URL: `https://arxiv.org/abs/2005.14165`.

[7] Marcel Bruch, Martin Monperrus, and Mira Mezini. "Learning from Examples to Improve Code Completion Systems". In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Amsterdam, Netherlands, 2009.

[8] Jianlv Chen et al. *BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation*. 2024. arXiv: `2402.03216 [cs.CL]`. URL: `https://arxiv.org/abs/2402.03216`.

[9] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: `2107.03374 [cs.LG]`. URL: `https://arxiv.org/abs/2107.03374`.

[10] Qi Chen et al. *SPTAG: A Library for Fast Approximate Nearest Neighbor Search*. Available at `https://github.com/Microsoft/SPTAG`. 2018.

[11] DeepSeek-AI et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: `2501.12948 [cs.CL]`. URL: `https://arxiv.org/abs/2501.12948`.

[12] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. Available at `https://arxiv.org/abs/1810.04805`. 2019. arXiv: `1810.04805 [cs.CL]`. URL: `https://arxiv.org/abs/1810.04805`.

[13] Matthijs Douze et al. "The Faiss library". In: (2024). arXiv: `2401.08281 [cs.LG]`.

[14] *Elasticsearch*. Online. Available at `https://github.com/elastic/elasticsearch`.

[15] Yunfan Gao et al. "Retrieval-Augmented Generation for Large Language Models: A Survey". In: *arXiv preprint* arXiv:2312.10997 (2023). Available at `https://arxiv.org/abs/2312.10997`.

[16] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: `2407.21783 [cs.AI]`. URL: `https://arxiv.org/abs/2407.21783`.

[17] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[18] Pengfei He et al. “Exploring Demonstration Retrievers in RAG for Coding Tasks: Yeas and Nays!” In: *arXiv preprint* arXiv:2410.09662v1 (2024). Available at https://arxiv.org/abs/2410.09662.

[19] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-term Memory”. In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[20] Marko Hostnik and Marko Robnik-Sikonja. “Retrieval-Augmented Code Completion for Local Projects Using Large Language Models”. In: *arXiv preprint* arXiv:2408.05026v1 (2024). Available at https://arxiv.org/abs/2408.05026.

[21] Binyuan Hui et al. “Qwen2.5-Coder Technical Report”. In: *arXiv preprint* arXiv:2409.12186 (2024). Available at https://arxiv.org/abs/2409.12186.

[22] Gautier Izacard et al. “Unsupervised Dense Information Retrieval with Contrastive Learning”. In: *Transactions on Machine Learning Research* (2022). ISSN: 2835-8856. URL: https://openreview.net/forum?id=jKN1pXi7b0.

[23] O. Jafari et al. “A Survey on Locality Sensitive Hashing Algorithms and Their Applications”. In: *arXiv preprint* arXiv:2102.08942 (2021). Available at https://arxiv.org/abs/2102.08942.

[24] Albert Q. Jiang et al. *Mistral 7B*. 2023. arXiv: 2310.06825 [cs.CL]. URL: https://arxiv.org/abs/2310.06825.

[25] Jeff Johnson, Matthijs Douze, and Hervé Jégou. “Billion-scale Similarity Search with GPUs”. In: *arXiv preprint* arXiv:1702.08734 (2017). Available at https://arxiv.org/abs/1702.08734.

[26] Aadit Kshirsagar. “Enhancing RAG Performance Through Chunking and Text Splitting Techniques”. In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* 10 (Sept. 2024), pp. 151–158. DOI: 10.32628/CSEIT2410593.

[27] Mandar Kulkarni et al. “Reinforcement Learning for Optimizing RAG for Domain Chatbots”. In: *Proceedings of the AAAI 2024 Workshop on Synergy of Reinforcement Learning and Large Language Models*. 2024.

[28] Patrick Lewis et al. “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”. In: *arXiv preprint* arXiv:2005.11401 (2020).

[29] Jie Li et al. “The Design and Implementation of a Real Time Visual Search System on JD E-commerce Platform”. In: *arXiv preprint* arXiv:1908.07389 (2019). Available at https://arxiv.org/abs/1908.07389.

[30] Jie Li et al. “The Design and Implementation of a Real Time Visual Search System on JD E-commerce Platform”. In: *arXiv preprint* arXiv:1908.07389 (2019). Available at https://arxiv.org/abs/1908.07389.

[31] Xinze Li et al. “Structure-Aware Language Model Pretraining Improves Dense Retrieval on Structured Data”. In: *arXiv preprint* arXiv:2305.19912 (2023). Available at https://arxiv.org/abs/2305.19912.

[32] Y. A. Malkov and D. A. Yashunin. “Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.4 (2018), pp. 824–836. DOI: 10.1109/TPAMI.2018.2889473.

[33] Tomas Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: *arXiv preprint arXiv:1301.3781* (2013).

[34] Bhaskar Mitra and Nick Craswell. “Neural Models for Information Retrieval”. In: *arXiv preprint* arXiv:1705.01509v1 (May 2017). Available at https://arxiv.org/abs/1705.01509.

[35] *PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension*. Online. Available at https://www.researchgate.net/publication/341744935.

[36] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. “GloVe: Global Vectors for Word Representation”. In: *Conference on Empirical Methods in Natural Language Processing*. 2014. URL: https://api.semanticscholar.org/CorpusID:1957433.

[37] Alec Radford and Karthik Narasimhan. “Improving Language Understanding by Generative Pre-Training”. In: 2018. URL: https://api.semanticscholar.org/CorpusID:49313245.

[38] Alec Radford et al. *Language Models are Unsupervised Multitask Learners*. Tech. rep. Available at https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf. OpenAI, 2019.

[39] Colin Raffel et al. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. Available at 1910.10683. 2023. arXiv: 1910.10683 [cs.LG]. URL: https://arxiv.org/abs/1910.10683.

[40] O. Ram et al. "In-Context Retrieval-Augmented Language Models". In: *arXiv preprint* arXiv:2302.00083v3 (2023). Available at https://arxiv.org/abs/2302.00083.

[41] Nils Reimers and Iryna Gurevych. "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Nov. 2019. URL: https://arxiv.org/abs/1908.10084.

[42] Stephen E. Robertson and Hugo Zaragoza. "The Probabilistic Relevance Framework: BM25 and Beyond". In: *Found. Trends Inf. Retr.* 3 (2009), pp. 333–389. URL: https://api.semanticscholar.org/CorpusID:207178704.

[43] Baptiste Rozière et al. "Code Llama: Open Foundation Models for Code". In: *arXiv preprint* arXiv:2308.12950 (2023). Available at https://arxiv.org/abs/2308.12950.

[44] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (1986). Available at https://doi.org/10.1038/323533a0, pp. 533–536.

[45] Saksham Sachdev et al. "Retrieval on Source Code: A Neural Code Search". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*. Philadelphia, PA, USA, 2018.

[46] Pranab Sahoo et al. *A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications*. 2024. arXiv: 2402.07927 [cs.AI]. URL: https://arxiv.org/abs/2402.07927.

[47] Alexey Svyatkovskiy et al. "IntelliCode Compose: Code Generation using Transformer and Retrieval-based Models". In: *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*. 2020.

[48] Hanzhuo Tan et al. "Prompt-based Code Completion via Multi-Retrieval Augmented Generation". In: *arXiv preprint* arXiv:2405.07530v1 (2024). Available at https://arxiv.org/abs/2405.07530.

[49] CodeGemma Team et al. *CodeGemma: Open Code Models Based on Gemma*. 2024. arXiv: 2406.11409 [cs.CL]. URL: https://arxiv.org/abs/2406.11409.

[50] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. 2024. arXiv: 2312.11805 [cs.CL]. URL: https://arxiv.org/abs/2312.11805.

[51] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL]. URL: https://arxiv.org/abs/2307.09288.

[52] Ashish Vaswani et al. "Attention Is All You Need". In: *arXiv preprint* arXiv:1706.03762 (2017). Available at https://arxiv.org/pdf/1706.03762.

[53] Jianguo Wang et al. "Milvus: A Purpose-Built Vector Data Management System". In: *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. Virtual Event, China, 2021, pp. 2624–2638. DOI: 10.1145/3448016.3457550.

[54] Liang Wang et al. *Improving Text Embeddings with Large Language Models*. 2024. arXiv: 2401.00368 [cs.CL]. URL: https://arxiv.org/abs/2401.00368.

[55] Liang Wang et al. *Text Embeddings by Weakly-Supervised Contrastive Pre-training*. 2024. arXiv: 2212.03533 [cs.CL]. URL: https://arxiv.org/abs/2212.03533.

[56] X. Wang et al. "KnowledGPT: Enhancing Large Language Models with Retrieval and Storage Access on Knowledge Bases". In: *arXiv preprint* arXiv:2308.11761v1 (2023). Available at https://arxiv.org/abs/2308.11761.

[57] Chuangxian Wei et al. "AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data". In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3152–3165. DOI: 10.14778/3415478.3415541.

[58] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

[59] Di Wu et al. "Repoformer: Selective Retrieval for Repository-Level Code Completion". In: *Proceedings of the 41st International Conference on Machine Learning*. Vol. 235. Vienna, Austria: PMLR, 2024.

[60] Chanwoong Yoon et al. "COMPACT: Compressing Retrieved Documents Actively for Question Answering". In: *Korea University and AIGEN Sciences* (2024). Authors: Chanwoong Yoon, Taewhoo Lee, Hyeon Hwang, Minbyul Jeong, and Jaewoo Kang.

[61] Fengji Zhang et al. "RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation". In: *arXiv preprint* arXiv:2303.12570 (2023). Available at `https://arxiv.org/abs/2303.12570`.

[62] Xinyu Zhang et al. "MIRACL: A Multilingual Retrieval Dataset Covering 18 Diverse Languages". In: *Transactions of the Association for Computational Linguistics* 11 (Sept. 2023), pp. 1114–1131. ISSN: 2307-387X. DOI: `10.1162/tacl_a_00595`. eprint: `https://direct.mit.edu/tacl/article-pdf/doi/10.1162/tacl\_a\_00595/2157340/tacl\_a\_00595.pdf`. URL: `https://doi.org/10.1162/tacl%5C_a%5C_00595`.

[63] Xinyu Zhang et al. "Toward Best Practices for Training Multilingual Dense Retrieval Models". In: *ACM Trans. Inf. Syst.* 42.2 (Sept. 2023). ISSN: 1046-8188. DOI: `10.1145/3613447`. URL: `https://doi.org/10.1145/3613447`.

[64] Ziyao Zhang et al. "LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation". In: *arXiv preprint* arXiv:2409.20550v1 (2024). Available at `https://arxiv.org/abs/2409.20550`.

Appendices

Appendix A — Interface of the RAGGIN extension

Figure A.1: Main interface when in Next.js workspace

Figure A.2: Main interface when not in Next.js workspace

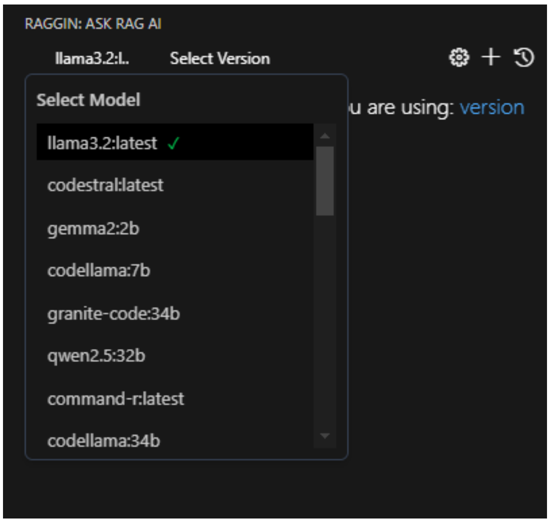

Figure A.3: Select LLM

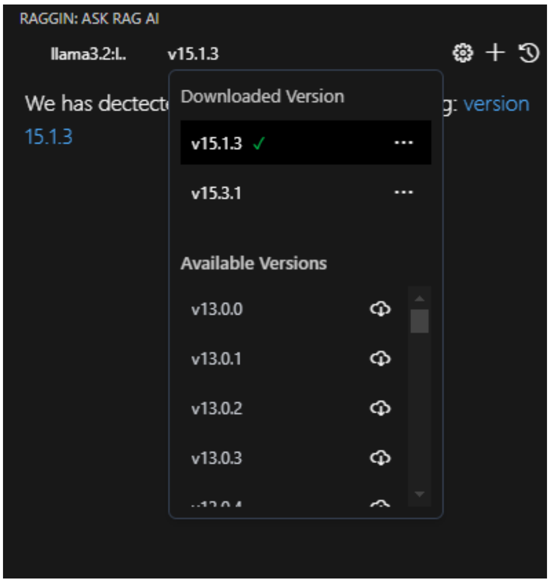

Figure A.4: Select Version

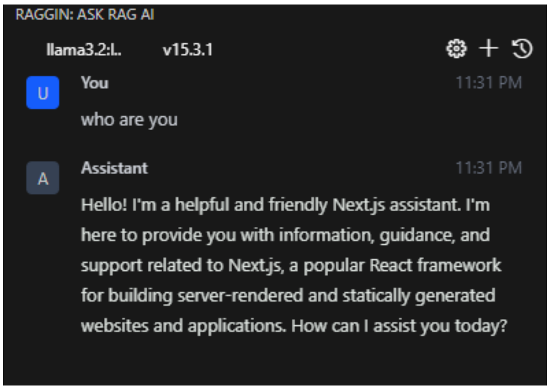

Figure A.5: RAGGIN's response to simple question

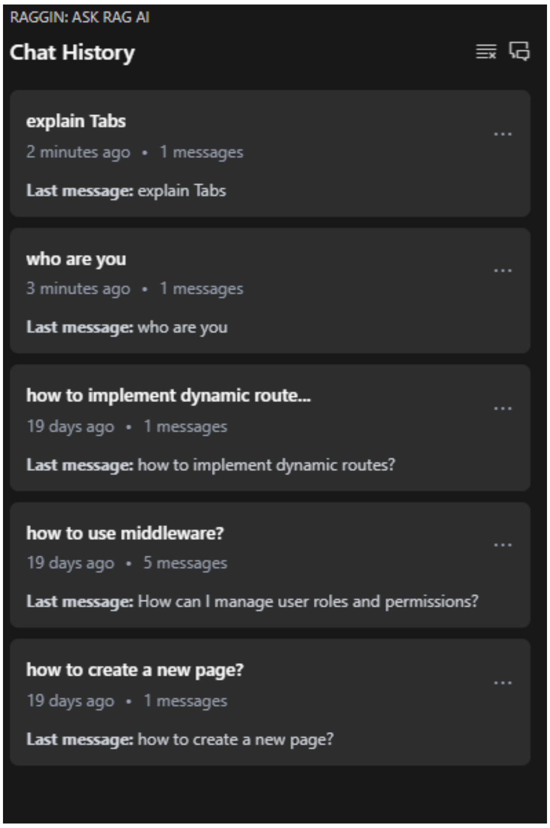

Figure A.6: Chat history

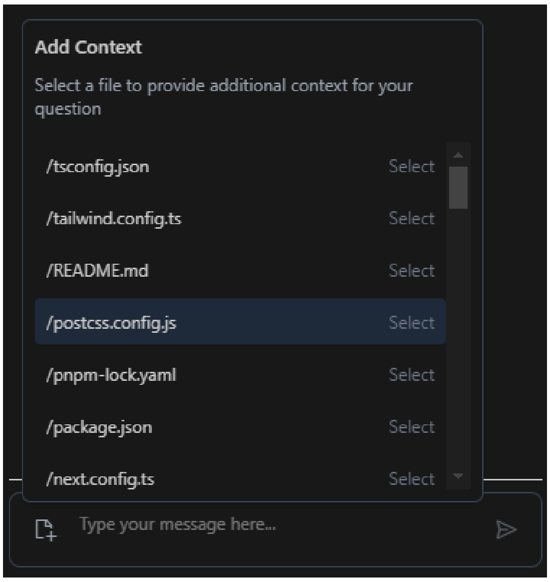

Figure A.7: Add file to context

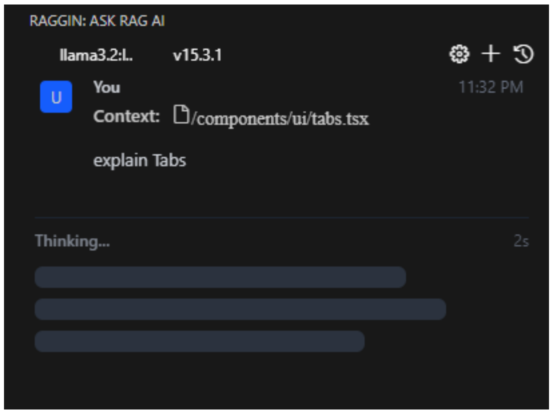

Figure A.8: Send question with selected file

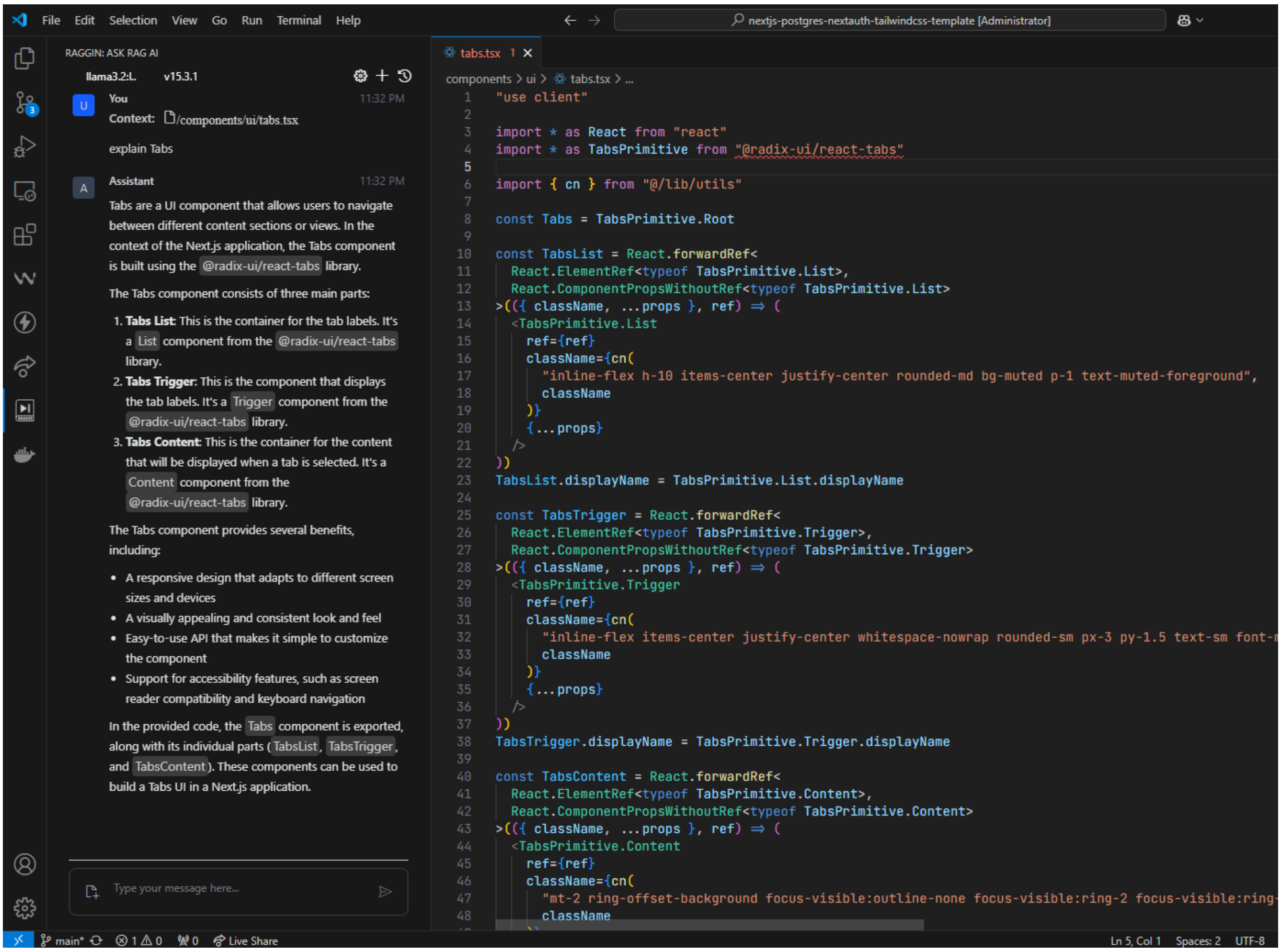

Figure A.9: RAGGIN's response to question with selected file

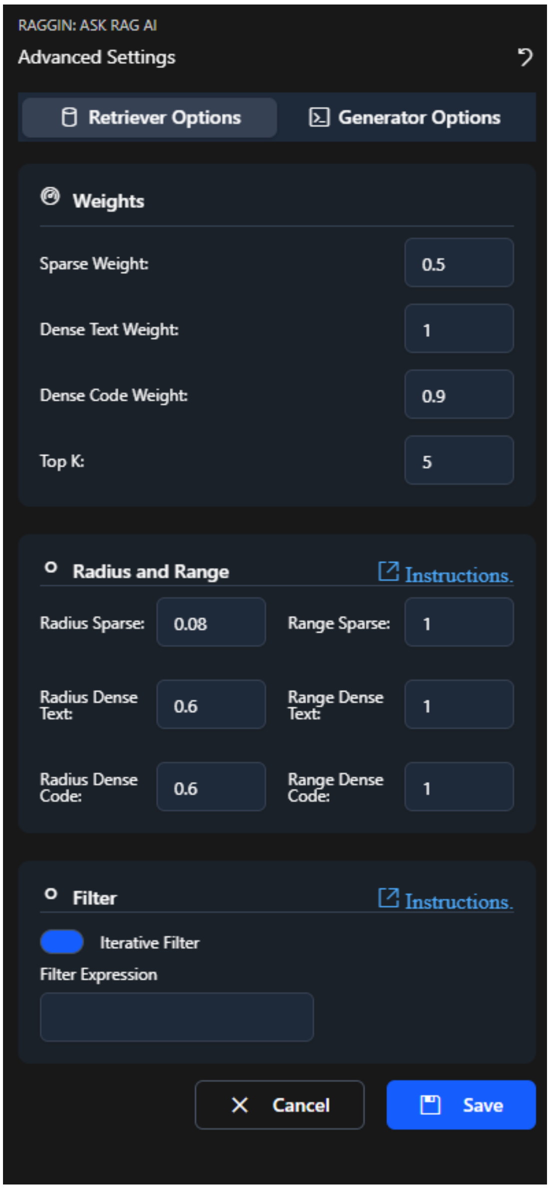

Figure A.10: Retriever options

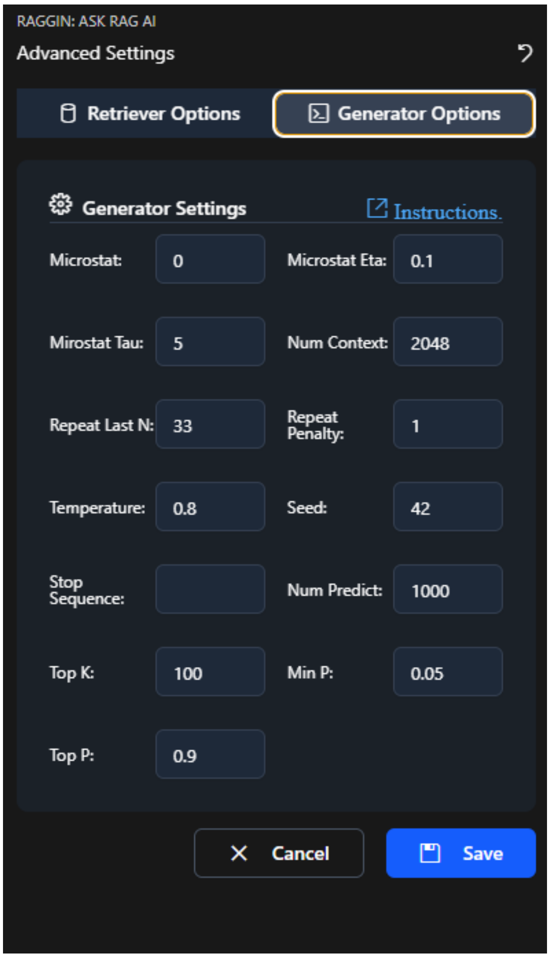

Figure A.11: Generator options

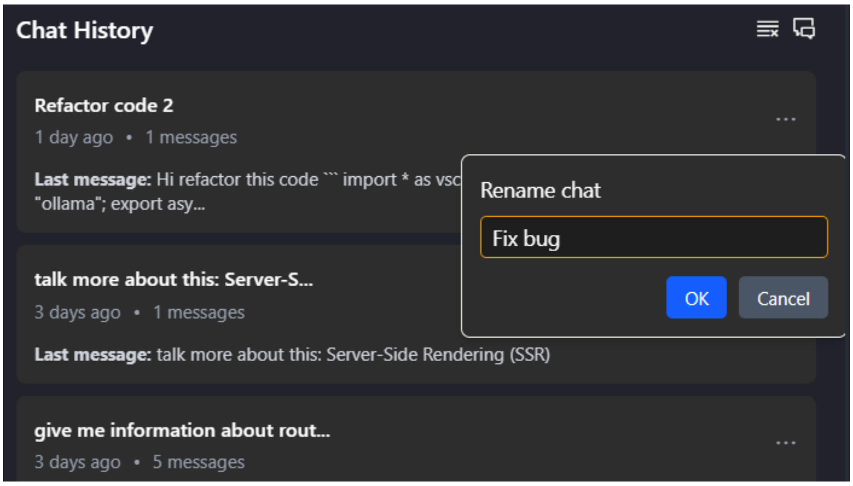

Figure A.12: Rename a conversation

B Evaluation and test results

K	Precision@K	Recall@K	Latency (s)
1	0.27	0.05	1.18
3	0.44	0.36	1.17
5	0.57	0.81	1.21
7	0.37	0.74	1.18

Table B.1: Retriever evaluation results for simple queries with default parameters

K	Precision@K	Recall@K	Latency (s)
1	0.07	0.02	3.40
3	0.24	0.12	3.48
5	0.29	0.24	3.49
7	0.21	0.24	3.49

Table B.2: Retriever evaluation results for complex queries with default parameters

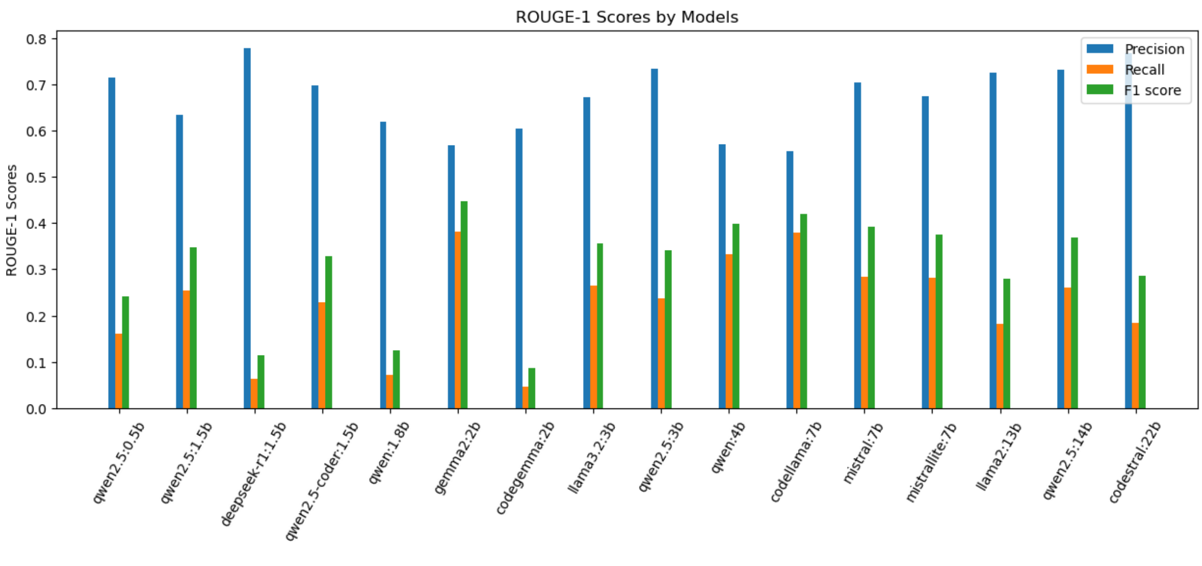

Figure B.1: ROUGE-1 Scores by LLMs

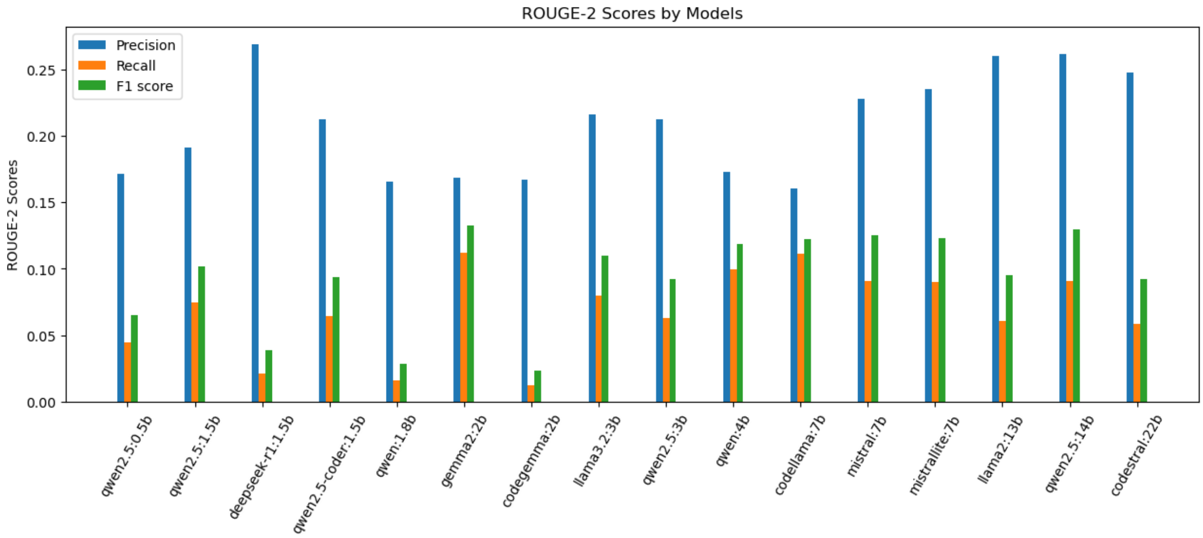

Figure B.2: ROUGE-2 Scores by LLMs

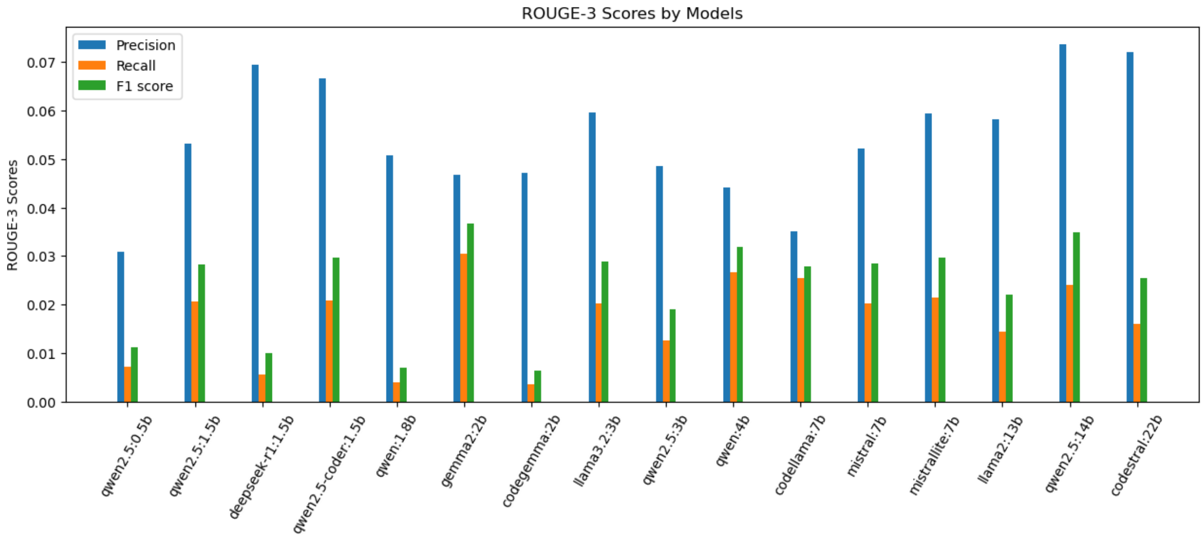

Figure B.3: ROUGE-3 Scores by LLMs

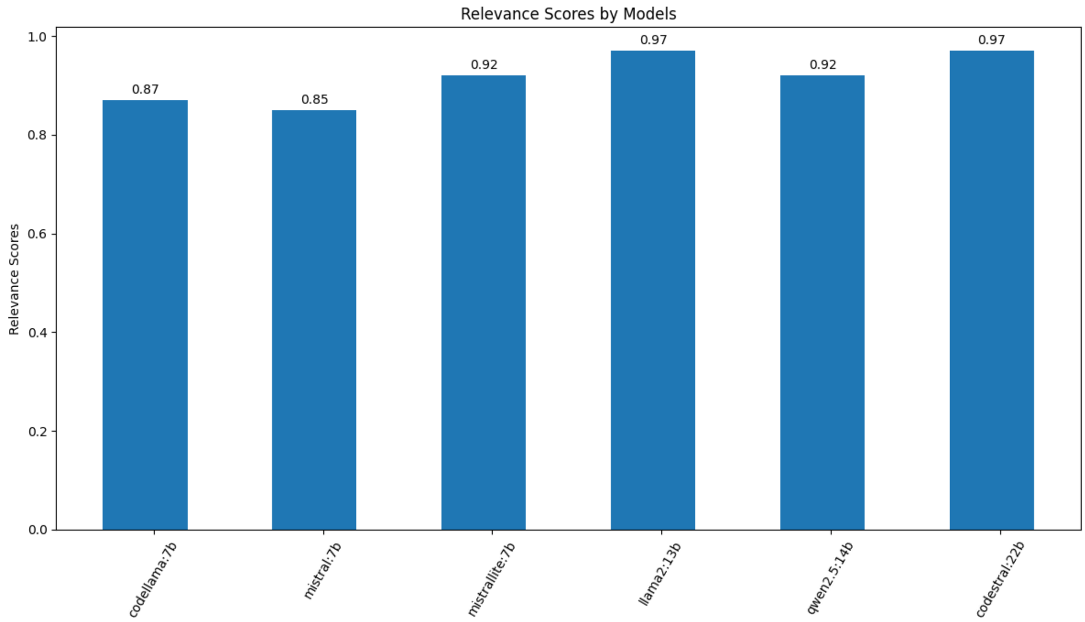

Figure B.4: Relevance Scores by some LLMs

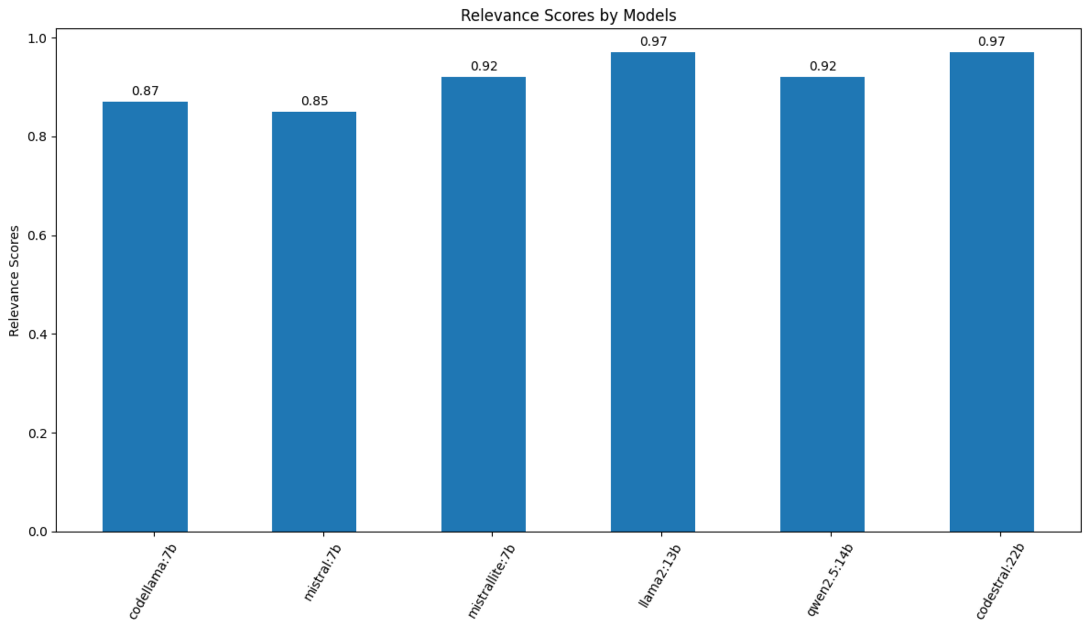

Figure B.5: Relevance Scores by larger LLMs