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

Committee:	
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) pipeline integrated into a Visual Studio Code (VSCode) extension, serving as a local intelligent assistant for web developers working with the Next.js framework. The system uses Milvus [51] 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 VSCode 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 [47], 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/`

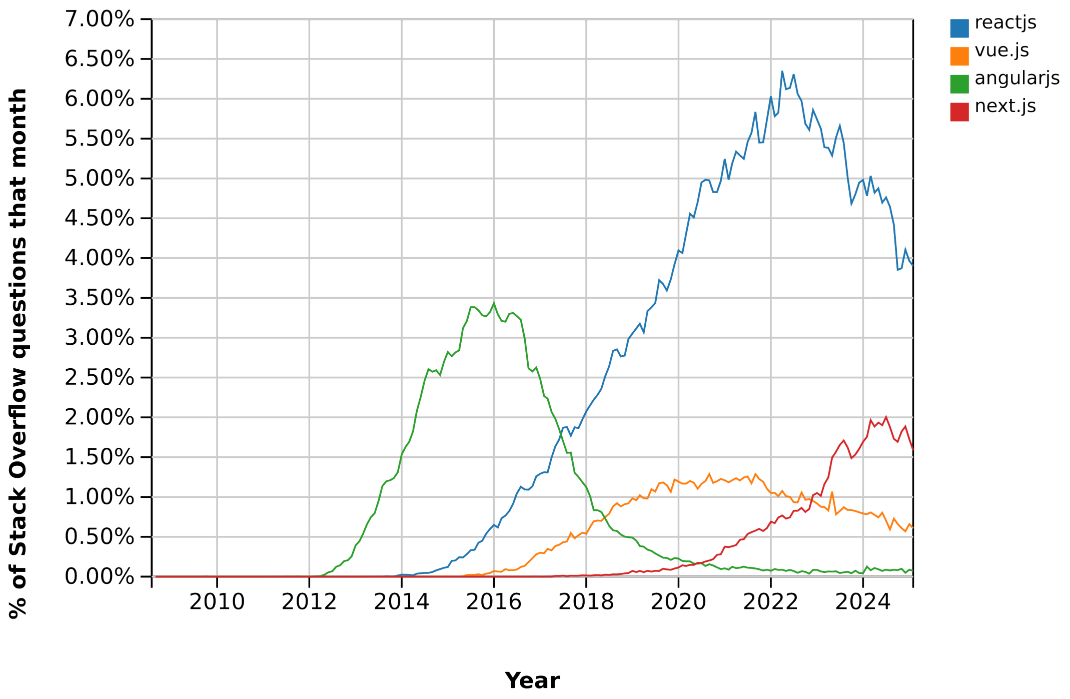


Figure 1.1: The popularity of frameworks based on StackOverflow questions 2008-2024

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 VSCode extension, that facilitates the web development process by addressing common challenges associated with modern frameworks. The tool aims to:

- 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.
- Analyze the user's codebase and suggest modifications for parts that are no longer relevant, particularly when updating the code to newer versions of a framework.

This extension will help streamline the development process, making it easier for developers to maintain, upgrade, and enhance their applications with minimal friction.

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. Key concepts of the Next.js framework and associated development tools are also introduced.

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 VSCode extensions and discusses strategies for retrieval and augmentation. Comparative analyses of existing frameworks and methodologies are also presented.

Chapter 4. Approach. This chapter describes the proposed solution, including the architecture of the RAG-enhanced system. It explains the rationale for selecting specific models like BGE-M3 and Qwen and details the integration of local vector storage with the Next.js framework to ensure efficient retrieval and generation.

Chapter 5. Initial implementation. This chapter outlines the step-by-step development process of the RAG pipeline. It includes data preparation, database setup, and integration with the VSCode extension. Challenges encountered during implementation and their solutions are also discussed.

Chapter 6. Evaluation plan. This chapter evaluates the system using metrics such as accuracy and developer productivity. It describes the testing methodology, compares results with existing solutions, and highlights the strengths and limitations of the proposed approach.

Chapter 7. Future plan. The final chapter summarizes the contributions of the research and discusses potential areas for improvement. It offers recommendations for extending the system's compatibility and incorporating advanced retrieval techniques to enhance performance and scalability.

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) 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.

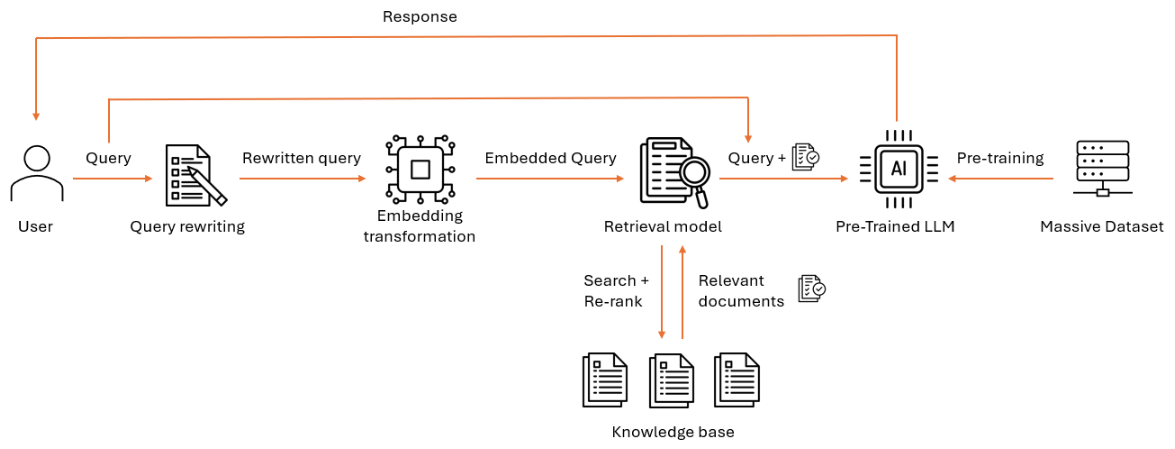


Figure 2.1: RAG Process

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 [14].

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) [23].

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 [14].

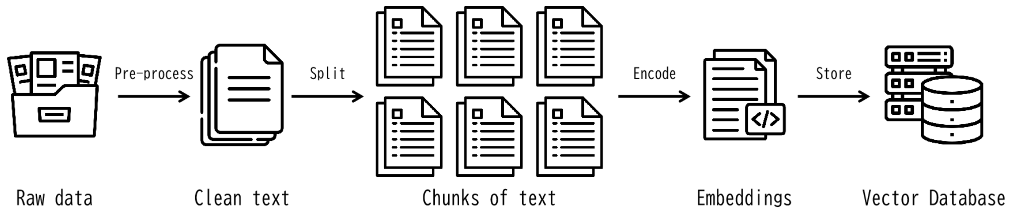


Figure 2.2: Preprocessing process for retriever

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 [11] and Sentence-BERT [39] 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** [30]: 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** [11]: 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) [39]: 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** [12] 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) [34] use this method to learn semantic relationships between words from large corpora.

Fine-tune embedding models: Reinforcement learning [24]: 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) [24], 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 [29] 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.

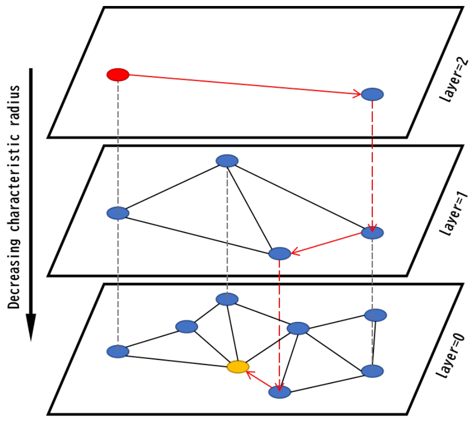


Figure 2.3: HNSW search algorithm visualization. The search starts from entry point from top layer (shown red), red arrows show direction of greedy algorithm to query point (shown yellow) [29].

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

```
1: $W \leftarrow \emptyset$                                         ▷ list for the currently found nearest elements
2: $ep \leftarrow$ get enter point for hsnw
3: $L \leftarrow$ level of $ep$                                     ▷ top layer for hsnw
4: $l \leftarrow -\lfloor \ln(\text{unif}(0,1)) \cdot mL \rfloor$   ▷ new element's level
5: $l \leftarrow \min(l, L+1)$
6: for $lc \leftarrow L$ to $l+1$ do
7:     $W \leftarrow$ SEARCH-LAYER($q, ep$, ef $= 1, lc$)
8:     $ep \leftarrow$ get the nearest element from $W$ to $q$
9: end for
10: for $lc \leftarrow \min(L, l)$ to 0 do
11:     $W \leftarrow$ SEARCH-LAYER($q, ep$, efConstruction, $lc$)
12:     neighbors $\leftarrow$ SELECT-NEIGHBORS($q, W, M, lc$)      ▷ Algorithm 3 or 4
13:     add bidirectional connections from neighbors to $q$ at layer $lc$
14:     for each $e \in$ neighbors do
15:         $eComm \leftarrow$ neighborhood($e$) at layer $lc$
16:         if $|eComm| > M_{\max}$ then                           ▷ shrink connections if needed
17:             if $lc = 0$ then                                   ▷ if $lc = 0$ then $M_{\max} = M_{\max0}$
18:                 $M_{\max} \leftarrow M_{\max0}$
19:             end if
20:             $eComm \leftarrow$ SELECT-NEIGHBORS($e, eComm, M_{\max}, lc$)   ▷ Algorithm 3 or 4
21:         end if
22:         set neighborhood($e$) at layer $lc$ to $eComm$
23:     end for
24: end for
25: $ep \leftarrow W$
26: 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

```
1:  $v \leftarrow ep$                                        ▷ Set of visited elements
2:  $C \leftarrow ep$                                        ▷ Set of candidates
3:  $W \leftarrow ep$                                        ▷ Dynamic list of found nearest neighbors
4:  while $|C| > 0$ do
5:      $c \leftarrow$ Extract nearest element from $C$ to $q$
6:      $f \leftarrow$ Get furthest element from $W$ to $q$
7:      if distance($c, q$) > distance($f, q$) then
8:          break                                            ▷ All elements in $W$ are evaluated
9:      end if
10:     for each $e \in$ neighbourhood($c$) at layer $lc$ do       ▷ Update $C$ and $W$
11:         if $e \notin v$ then
12:             $v \leftarrow v \cup e$
13:             $f \leftarrow$ Get furthest element from $W$ to $q$
14:             if distance($e, q$) < distance($f, q$) or $|W| < ef$ then
15:                 $C \leftarrow C \cup e$
16:                 $W \leftarrow W \cup e$
17:                 if $|W| > ef$ then
18:                     Remove furthest element from $W$ to $q$
19:                 end if
20:             end if
21:         end if
22:     end for
23: end while
24: 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

```
1: 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

```
 1: $R \leftarrow \emptyset$
 2: $W \leftarrow C$                                        ▷ Working queue for the candidates
 3: if extendCandidates then                               ▷ Extend candidates by their neighbors
 4:     for each $e \in C$ do
 5:         for each $e_{\text{adj}} \in$ neighbourhood($e$) at layer $lc$ do
 6:             if $e_{\text{adj}} \notin W$ then
 7:                 $W \leftarrow W \cup e_{\text{adj}}$
 8:             end if
 9:         end for
10:     end for
11: end if
12: $W_d \leftarrow \emptyset$                              ▷ Queue for the discarded candidates
13: while $|W| > 0$ and $|R| < M$ do
14:     $e \leftarrow$ Extract nearest element from $W$ to $q$
15:     if $e$ is closer to $q$ compared to any element from $R$ then
16:         $R \leftarrow R \cup e$
17:     else
18:         $W_d \leftarrow W_d \cup e$
19:     end if
20: end while
21: if keepPrunedConnections then             ▷ Add some of the discarded connections from $W_d$
22:     while $|W_d| > 0$ and $|R| < M$ do
23:         $R \leftarrow R \cup$ Extract nearest element from $W_d$ to $q$
24:     end while
25: end if
26: 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

```
1: $W \leftarrow \emptyset$                                 ▷ Set for the current nearest elements
2: $ep \leftarrow$ Get enter point for hsnw
3: $L \leftarrow$ Level of $ep$                             ▷ Top layer for hsnw
4: for $lc \leftarrow L \ldots 1$ do
5:     $W \leftarrow$ SEARCH-LAYER($q, ep, \texttt{ef} = 1, lc$)
6:     $ep \leftarrow$ Get nearest element from $W$ to $q$
7: end for
8: $W \leftarrow$ SEARCH-LAYER($q, ep, ef, lc = 0$)
9: 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 [14].

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 [14]. 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.

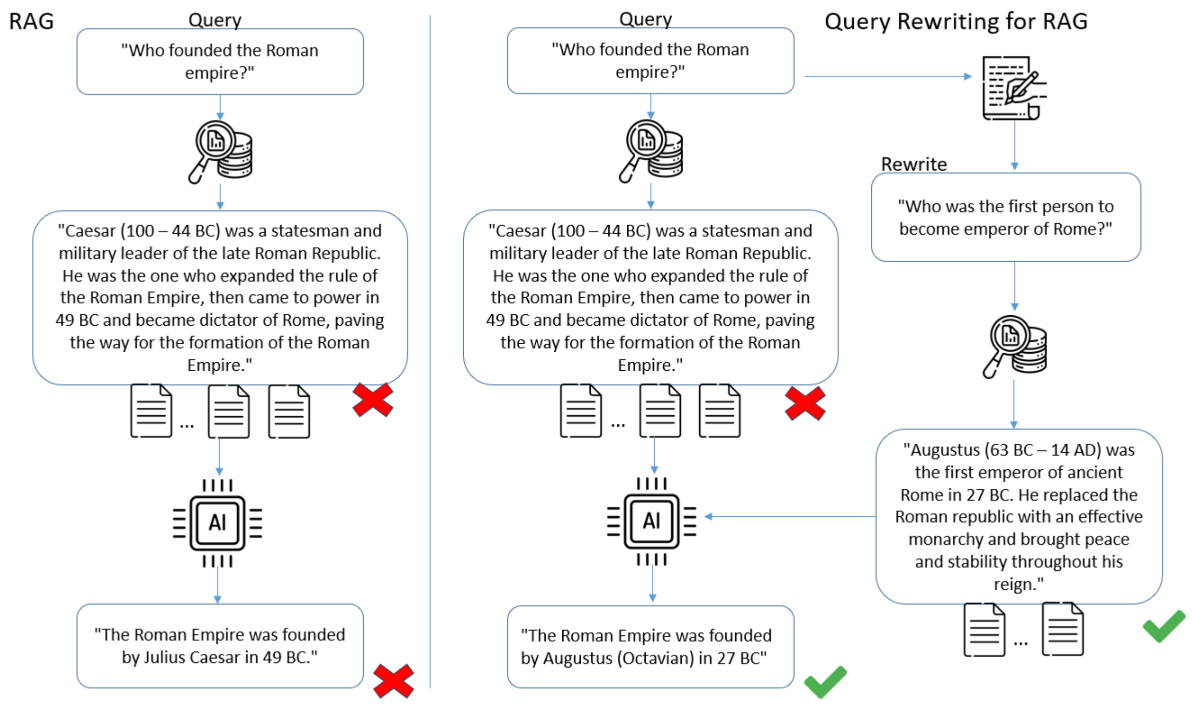

RAG
Query
"Who founded the Roman empire?"
"Caesar (100 – 44 BC) was a statesman and military leader of the late Roman Republic. He was the one who expanded the rule of the Roman Empire, then came to power in 49 BC and became dictator of Rome, paving the way for the formation of the Roman Empire."
AI
"The Roman Empire was founded by Julius Caesar in 49 BC."
Query
"Who founded the Roman empire?"
Query Rewriting for RAG
Rewrite
"Who was the first person to become emperor of Rome?"
"Augustus (63 BC – 14 AD) was the first emperor of ancient Rome in 27 BC. He replaced the Roman republic with an effective monarchy and brought peace and stability throughout his reign."
"The Roman Empire was founded by Augustus (Octavian) in 27 BC"

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 [28]. 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 [59], 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 [14].

In this section, we provide an overview of the architecture of Transformer models [50] 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 [50], a model architecture eschewing recurrence and instead relying entirely on an attention mechanism to draw global dependencies between input and output [50]. It has revolutionized natural language processing (NLP) by enabling parallel processing of sequences. Unlike recurrent architectures such as RNNs [42] or LSTMs [17], 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 [35], BERT [11], and T5 [37].

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 [15] 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 [50].
- **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 [50].

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 [50].

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.

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

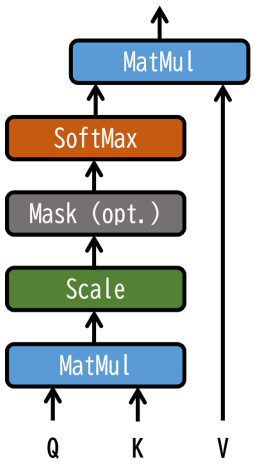


Figure 2.5: Scaled dot-product attention [50].

- **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 [44]. 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.

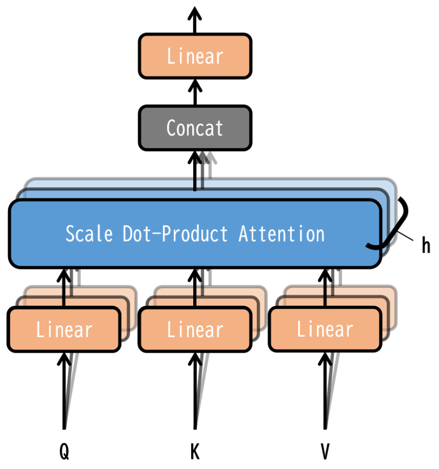


Figure 2.6: Multi-Head Attention (consists of several attention layers running in parallel) [50].

- **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 [56]. 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 [38], 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 [50] 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:

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

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:

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

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:

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

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

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

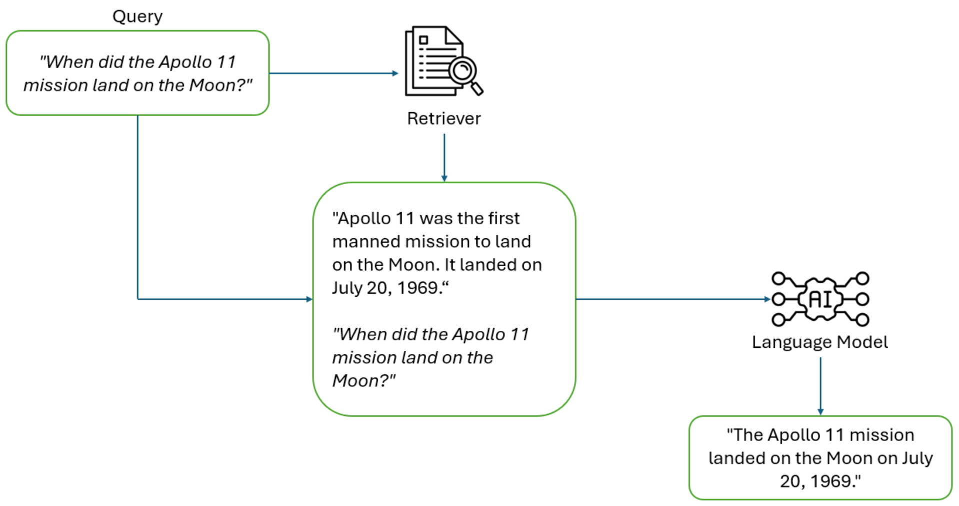
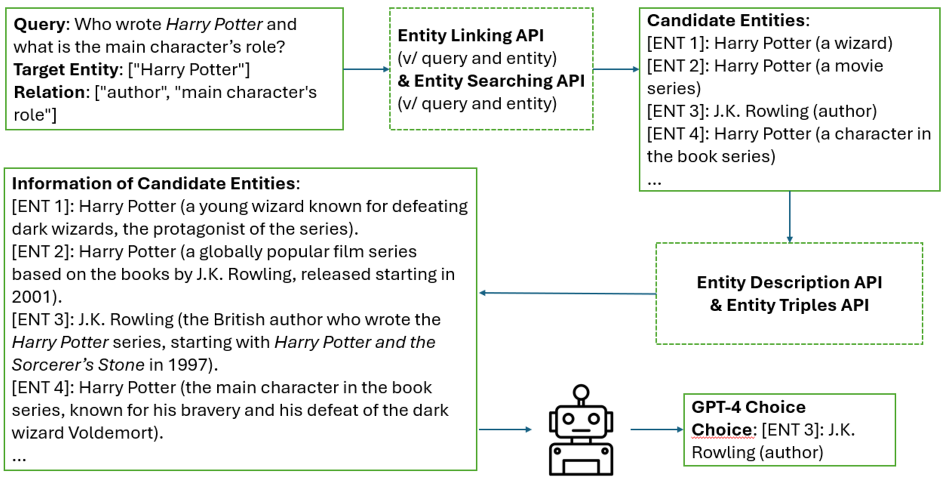

Query
"When did the Apollo 11 mission land on the Moon?"
Retriever
"Apollo 11 was the first manned mission to land on the Moon. It landed on July 20, 1969."
"When did the Apollo 11 mission land on the Moon?"
AI
Language Model
"The Apollo 11 mission landed on the Moon on July 20, 1969."

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 [54] 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.



Query: Who wrote *Harry Potter* and what is the main character's role?
Target Entity: ["Harry Potter"]
Relation: ["author", "main character's role"]
Entity Linking API (v/ query and entity)
& Entity Searching API (v/ query and entity)
Candidate Entities:
[ENT 1]: Harry Potter (a wizard)
[ENT 2]: Harry Potter (a movie series)
[ENT 3]: J.K. Rowling (author)
[ENT 4]: Harry Potter (a character in the book series)
...
Entity Description API & Entity Triples API
Information of Candidate Entities:
[ENT 1]: Harry Potter (a young wizard known for defeating dark wizards, the protagonist of the series).
[ENT 2]: Harry Potter (a globally popular film series based on the books by J.K. Rowling, released starting in 2001).
[ENT 3]: J.K. Rowling (the British author who wrote the *Harry Potter* series, starting with *Harry Potter and the Sorcerer's Stone* in 1997).
[ENT 4]: Harry Potter (the main character in the book series, known for his bravery and his defeat of the dark wizard Voldemort).
...
GPT-4 Choice
Choice: [ENT 3]: J.K. Rowling (author)

Figure 2.8: Workflow of augmentation with structured data sources [54].

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 $\mathcal{T}$.

```
1: function EMBSIM(str $r$, list[str] $\mathcal{R}$)
2:     $s = -1$
3:     $r =$ embedding($r$)
4:     for $r_i \in \mathcal{R}$ do
5:         $r_i =$ embedding($r_i$)
6:         if $\cos(r, r_i) > s$ then
7:             $s = \cos(r, r_i)$
8:         end if
9:     end for
10:    return $s$
11: end function
12: $e =$ entity_linking($\mathcal{E}$)
13: if $e == NULL$ then
14:     return $NULL$
15: end if
16: $r = NULL$
17: $s_r = -1$
18: for $triple \in triples$ do
19:     $r_i = triple.rel$
20:     $s_i =$ EMBSIM($r_i, \mathcal{R}$)
21:     if $s_i > s_r$ then
22:         $s_r = s_i$
23:         $r = r_i$
24:     end if
25: end for
26: if $r == NULL$ then
27:     return $NULL$
28: end if
29: $triples_r = triples$ with relation $r$
30: $\mathcal{R} =$ target entities or attribute values in $triples_r$
31: return $\mathcal{R}$
```

- 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 $\{d_1, \ldots, d_N\}$
 1: Input: input prompt $x$ and preceding generation $y_{<t}$, Output: next output segment $y_t$
 2: $\mathcal{M}$ predicts Retriever given $(x, y_{<t})$
 3: if Retriever == Yes then
 4:     Retrieve relevant text passages $\mathcal{D}$ using $\mathcal{R}$ given $(x, y_{t-1})$      ▷ Retrieve
 5:     $\mathcal{M}$ predicts IsREL given $x, d$ and $y_{<t}$ for each $d \in \mathcal{D}$      ▷ Generate
 6:     $\mathcal{M}$ predicts IsSUP and IsUSE given $x, d$ for each $d \in \mathcal{D}$      ▷ Critique
 7:     Rank $y_t$ based on IsREL, IsSUP, IsUSE
 8: else if Retriever == No then
 9:     $\mathcal{M}_{\text{gen}}$ predicts $y_t$ given $x$      ▷ Generate
10:     $\mathcal{M}_{\text{gen}}$ predicts IsUSE given $x, y_t$      ▷ Critique
11: 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 [16].

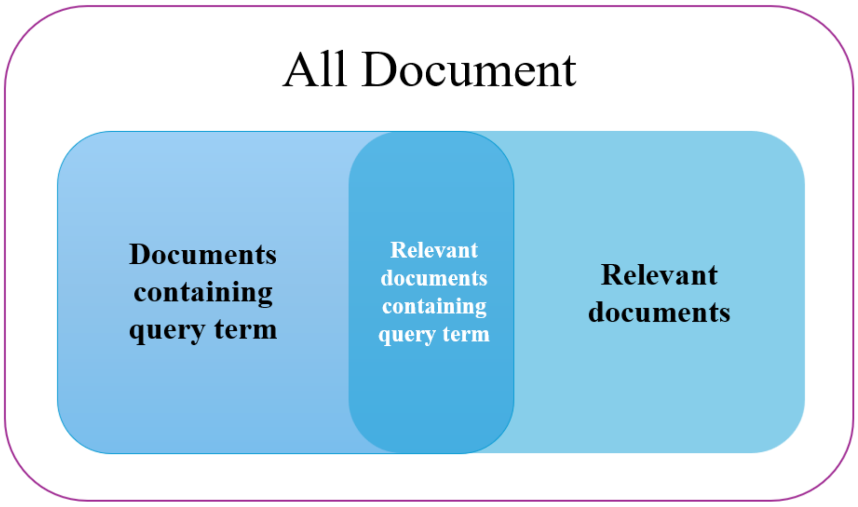

All Document
Documents containing query term
Relevant documents containing query term
Relevant documents

Figure 3.1: BM25 visualization.

Dense retrieval methods: Leveraging semantic embeddings, dense retrieval tools like SBERT [39]'s semantic search and approximate techniques (e.g., HNSW [29], ANNOY [4], and LSH [21]) provide scalable solutions for large knowledge bases. These approaches achieve significant speed improvements with minimal accuracy trade-offs [16].

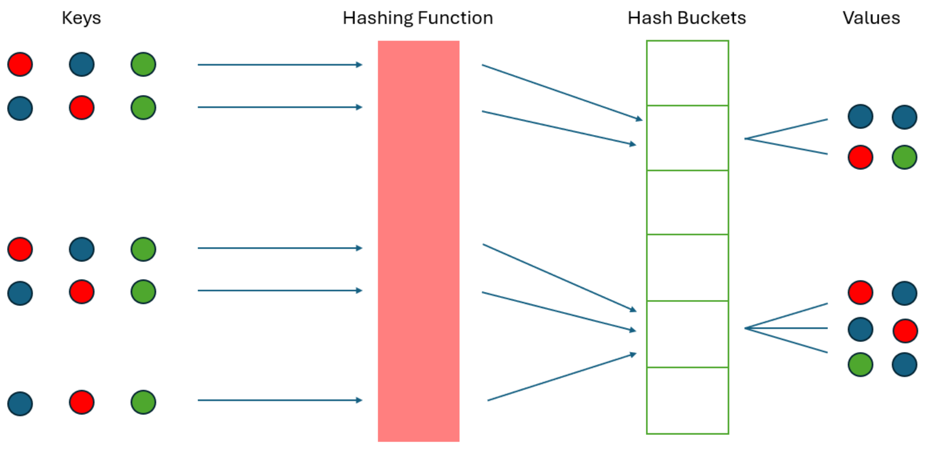
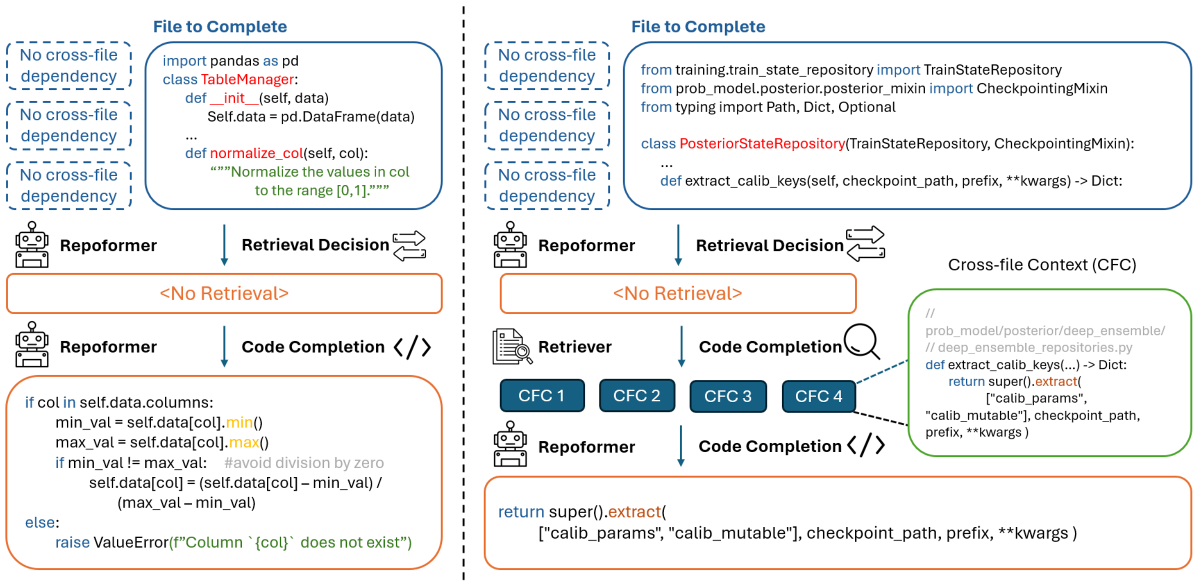


Figure 3.2: LSH visualization.

Selective retrieval: Frameworks such as Repoformer [57] employ selective retrieval, determining the necessity of retrieval based on contextual evaluation. This approach reduces inefficiencies and avoid adding irrelevant context.



```
import pandas as pd
class TableManager:
    def __init__(self, data)
        Self.data = pd.DataFrame(data)
    ...
    def normalize_col(self, col):
        """Normalize the values in col
            to the range [0,1]."""
```

```
if col in self.data.columns:
    min_val = self.data[col].min()
    max_val = self.data[col].max()
    if min_val != max_val:    #avoid division by zero
        self.data[col] = (self.data[col] – min_val) /
            (max_val – min_val)
else:
    raise ValueError(f"Column `{col}` does not exist")
```

```
from training.train_state_repository import TrainStateRepository
from prob_model.posterior.posterior_mixin import CheckpointingMixin
from typing import Path, Dict, Optional

class PosteriorStateRepository(TrainStateRepository, CheckpointingMixin):
    ...
    def extract_calib_keys(self, checkpoint_path, prefix, **kwargs) -> Dict:
```

```
//
prob_model/posterior/deep_ensemble/
// deep_ensemble_repositories.py
def extract_calib_keys(...) -> Dict:
    return super().extract(
        ["calib_params",
"calib_mutable"], checkpoint_path,
prefix, **kwargs )
```

```
return super().extract(
    ["calib_params", "calib_mutable"], checkpoint_path, prefix, **kwargs )
```

Figure 3.3: REPOFORMER framework [57].

Multi-retrieval perspectives: Systems like ProCC [46] utilize diverse perspectives, including lexical semantics, hypothetical line embeddings, and code summarization, to enrich context and improve code completion accuracy.

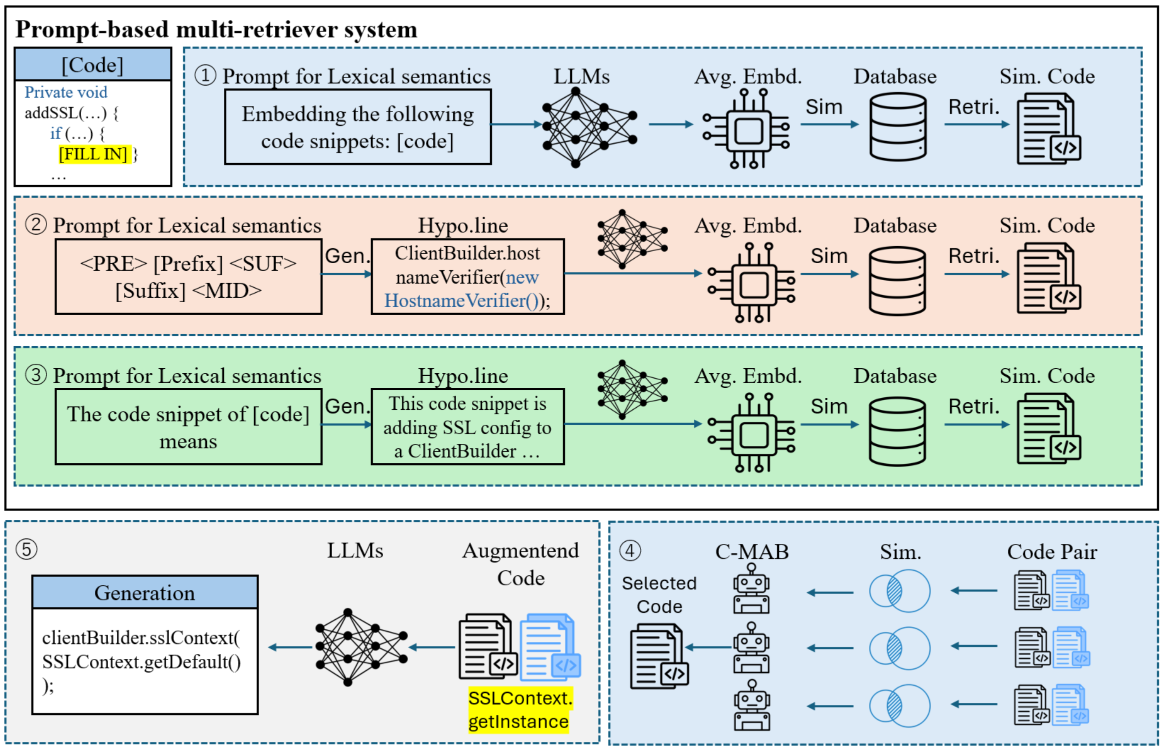

Prompt-based multi-retriever system
[Code]
Private void
addSSL(...) {
if (...) {
[FILL IN] }
...
① Prompt for Lexical semantics
Embedding the following code snippets: [code]
LLMs
Avg. Embd.
Sim
Database
Retri.
Sim. Code
② Prompt for Lexical semantics
<PRE> [Prefix] <SUF> [Suffix] <MID>
Gen.
Hypo.line
ClientBuilder.host nameVerifier(new HostnameVerifier());
③ Prompt for Lexical semantics
The code snippet of [code] means
This code snippet is adding SSL config to a ClientBuilder ...
⑤
Generation
clientBuilder.sslContext(SSLContext.getDefault());
Augmentend Code
SSLContext. getInstance
④ C-MAB
Selected Code
Sim.
Code Pair

Figure 3.4: ProCC framework [46].

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.

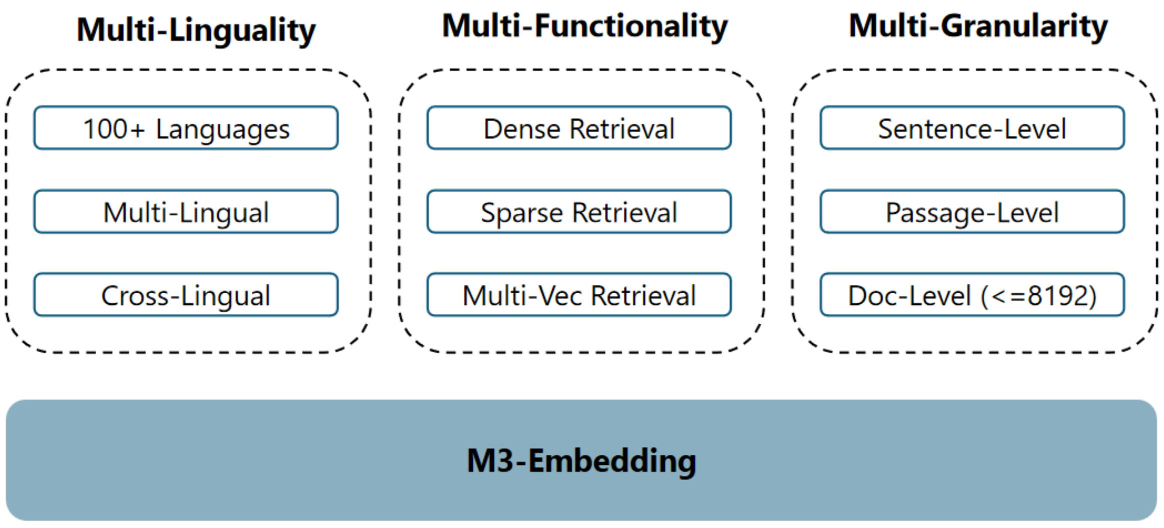


Figure 3.5: Characters of M3-Embedding [8].

3.1.2 Generative enhancements

Generative Pre-trained Transformers: LLMs like GPT-2 [36] and code Llama [41] are trained on extensive code corpora, enabling them to handle diverse tasks such as function completion and API usage.

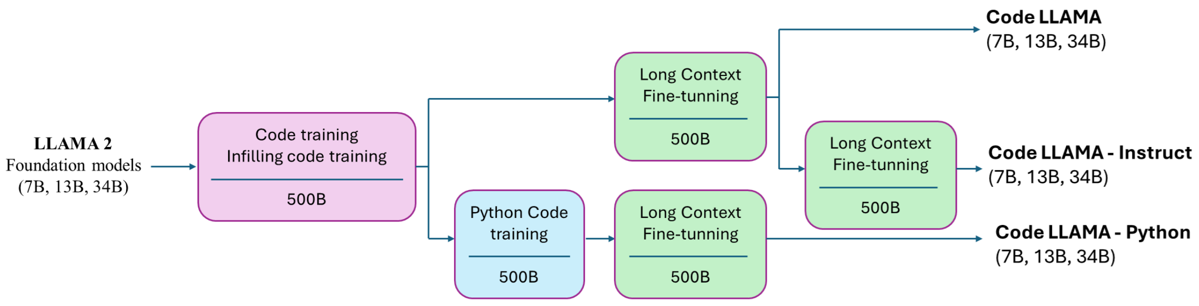


Figure 3.6: The Code Llama specialization pipeline [41].

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 [18].

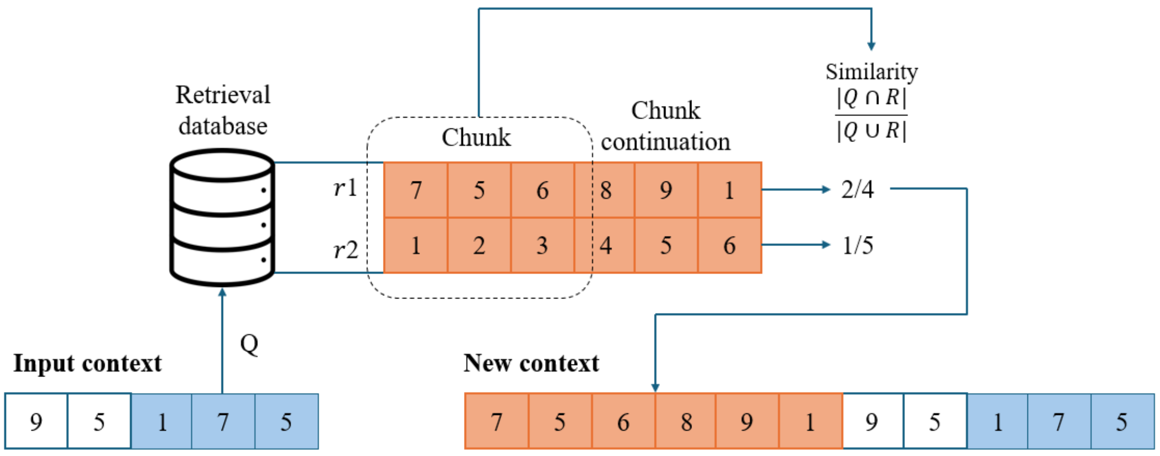


Figure 3.7: The retrieved snippet contains continuation tokens and additional metadata [18].

Specialized models for coding: Models like Qwen2.5-Coder [19] and CodeQwen1.5-7B-Chat [3] are fine-tuned for coding tasks, demonstrating improved performance in code generation and comprehension.

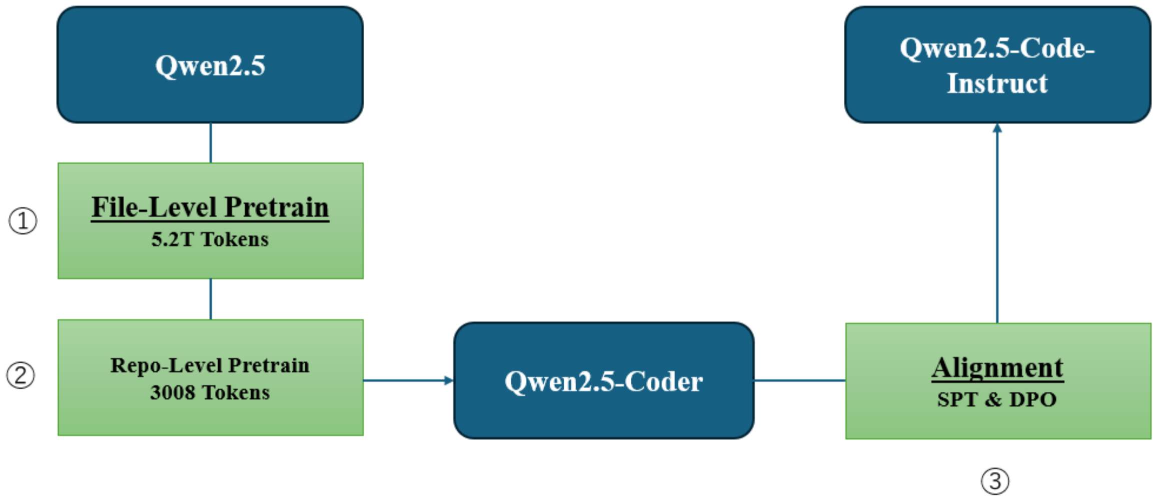
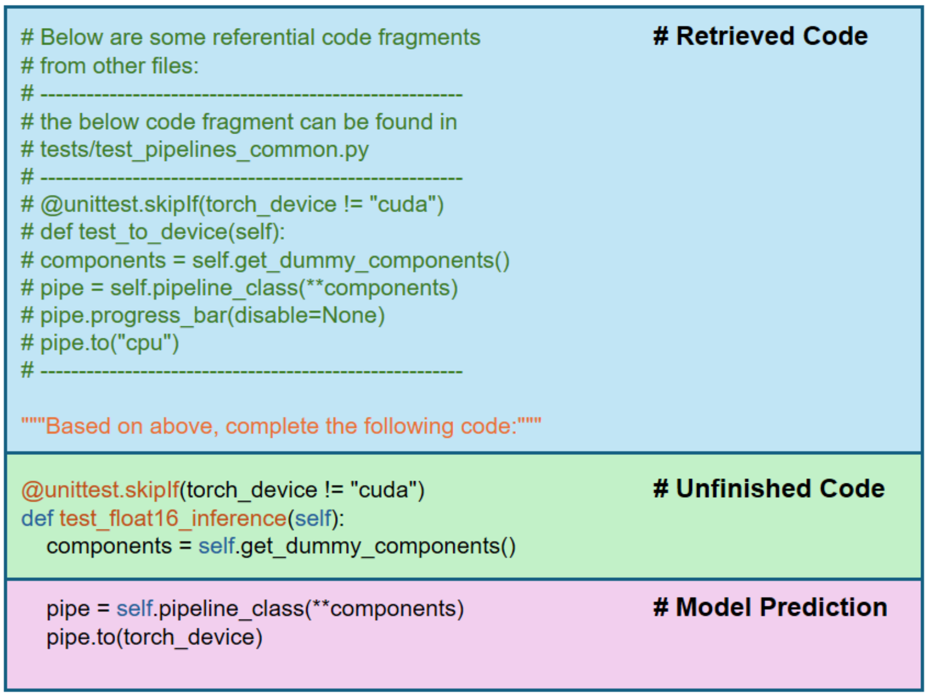


Figure 3.8: The three-stage training pipeline for Qwen2.5-Coder [18].

3.1.3 Specialized Applications

Repository-level code completion: Tools like Repoformer and RepoCoder [60] focus on cross-file dependencies, retrieving and integrating relevant code snippets to improve repository-wide coherence.



```
# Below are some referential code fragments
# from other files:
# -----------------------------------------------------
# the below code fragment can be found in
# tests/test_pipelines_common.py
# -----------------------------------------------------
# @unittest.skipIf(torch_device != "cuda")
# def test_to_device(self):
# components = self.get_dummy_components()
# pipe = self.pipeline_class(**components)
# pipe.progress_bar(disable=None)
# pipe.to("cpu")
# -----------------------------------------------------

"""Based on above, complete the following code:"""
```

```
@unittest.skipIf(torch_device != "cuda")
def test_float16_inference(self):
    components = self.get_dummy_components()
```

```
    pipe = self.pipeline_class(**components)
    pipe.to(torch_device)
```

Figure 3.9: Demonstrating the format of the RepoCoder prompt [60].

Commit message and assertion generation: RAG systems generate commit messages and assertions by retrieving semantically aligned examples, balancing efficiency and effectiveness [16].

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 [63].

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) [43] 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/

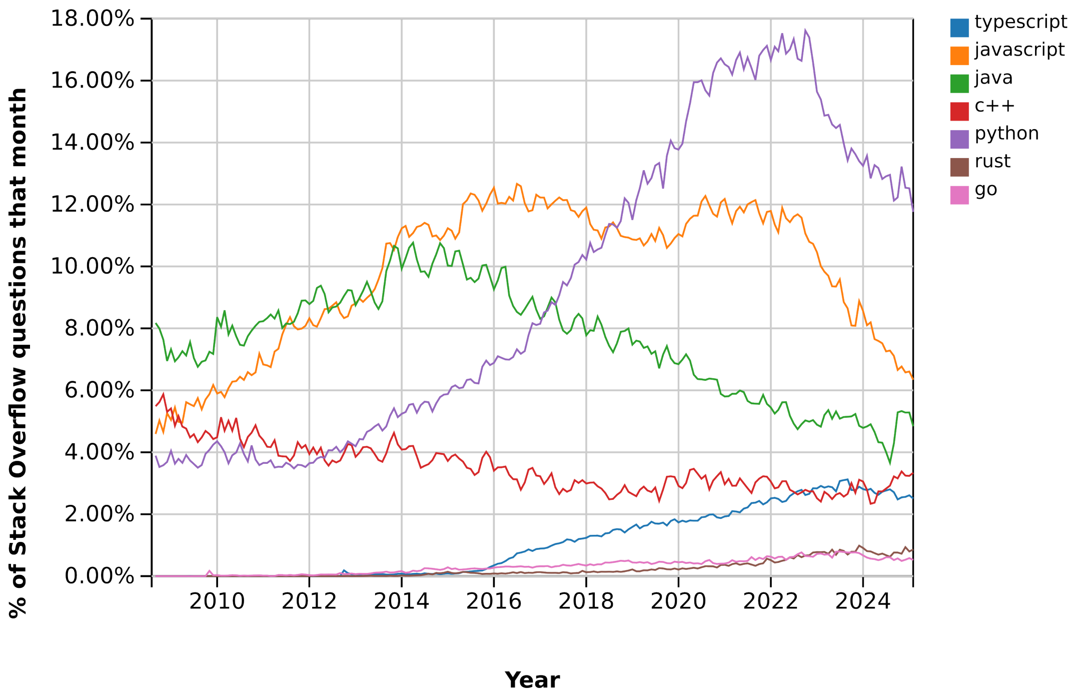


Figure 3.10: Stackoverflow trends chart 2008-2024.

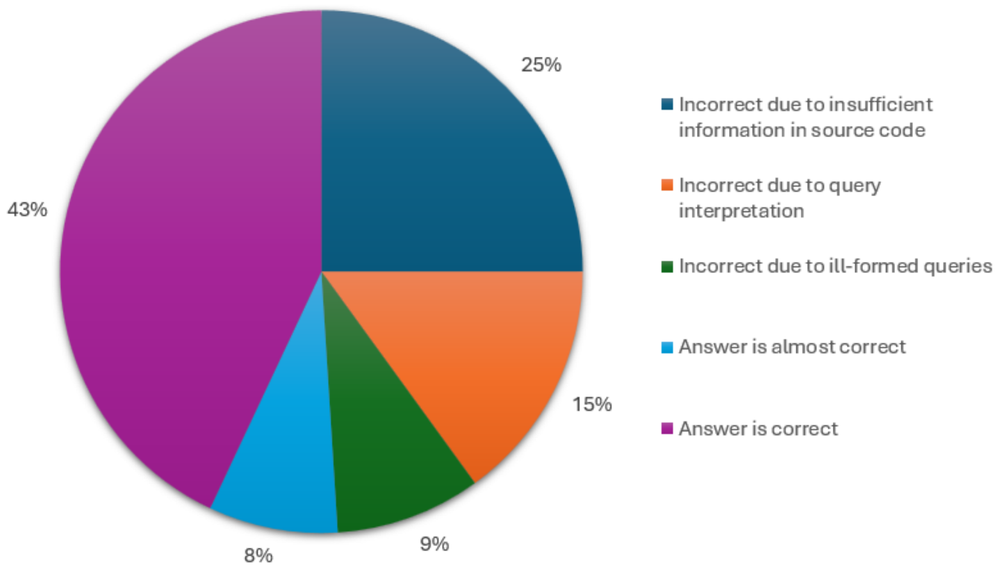


Figure 3.11: NCS evaluation results [43].

Code completion and suggestion systems: Intelligent tools like IntelliCode Compose [45] 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 [25], 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 [31], 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 [50] 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** [43]: 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** [45]: Integrates into VSCode, offering real-time code generation and suggestions.
- **Github Copilot**[4]: Embedded within various IDEs, including VSCode and JetBrains, it leverages AI to provide code completions and suggestions based on the current context.
- **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.

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

Feature	IntelliCode Compose	GitHub Copilot	Tabnine
Functionality	Provides real-time code generation and suggestions within VSCode.	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 VSCode.	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)
```

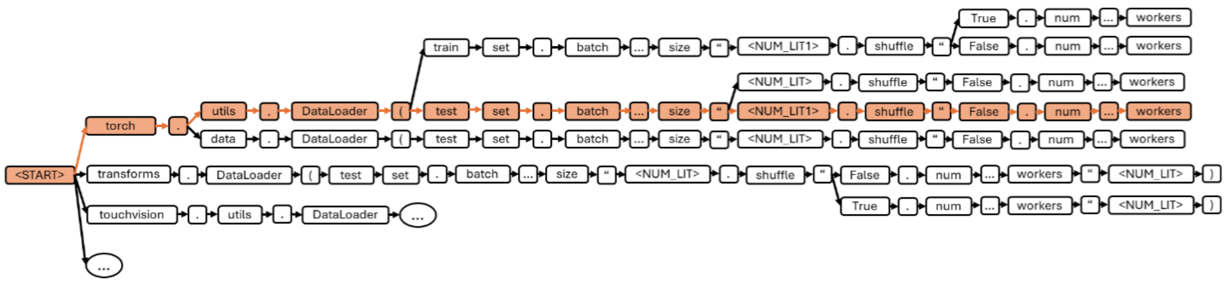


Figure 3.12: IntelliCode Compose's code snippet completion suggestion and completion-tree demonstration [45].

4 Approach

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.

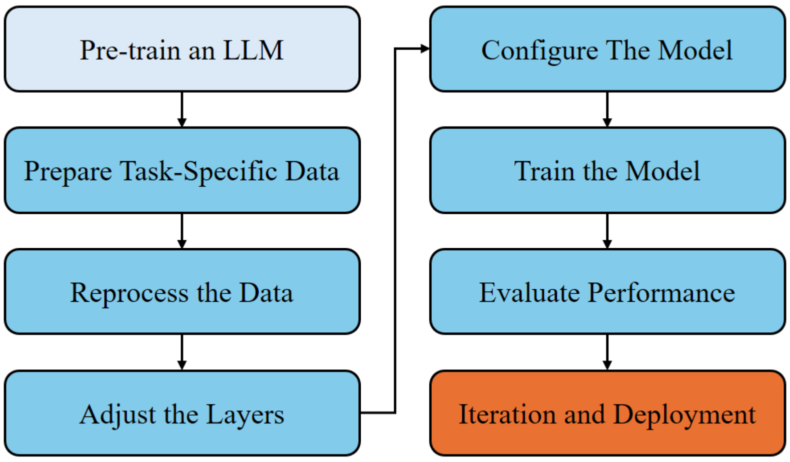


Figure 4.1: Fine-tuning process[1].

[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

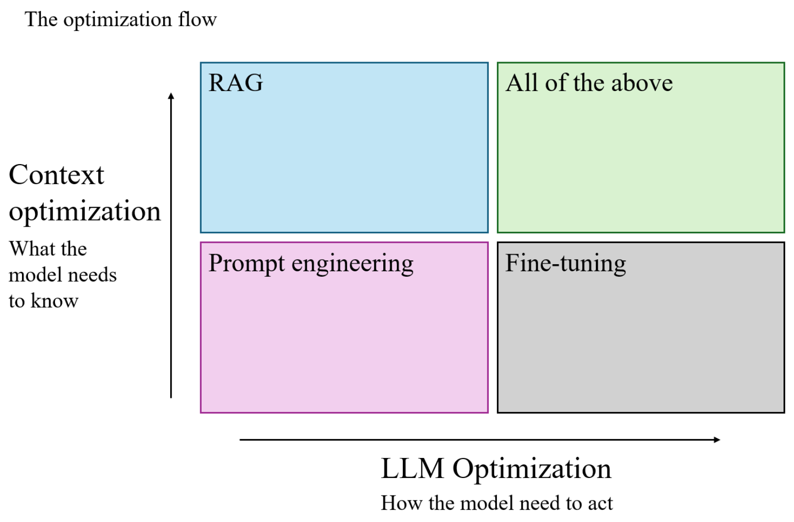


Figure 4.2: The nature of fine-tuning and RAG[2].

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 [40], mDPR [62], mContriever [20], mE5$_{\text{large}}$ [53], E5$_{\text{mistral-7b}}$ [52], BGE-M3 achieves an average retrieval score of 71.5 on the MIRACL dataset [61], 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 [40]	31.9
E5$_{\text{mistral-7b}}$ [52]	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 Qwen?

Qwen [3] is a relatively less recognized yet promising LLM developed by Alibaba Cloud, designed to offer robust natural language understanding and generation capabilities. While its public profile may not be as prominent as some of its contemporaries, the performance metrics in Table 4.3 demonstrate that Qwen is highly competitive in several benchmarks. For instance, the 14B parameter Qwen model achieves state-of-the-art results in tasks such as MMLU (66.3), C-Eval (72.1), and GSM8K (61.3), surpassing many well-known local models like LLaMA [49], Llama2 [48] and Baichuan2 [58]. This highlights Qwen's strong capability to handle diverse tasks effectively despite having fewer parameters compared to some larger models.

The decision to integrate Qwen into our RAG pipeline stems from its balance of efficiency, performance, and adaptability. By leveraging RAG, we aim to showcase how even a "lighter" LLM like Qwen can achieve substantial improvements by incorporating external knowledge sources. Unlike more resource-intensive models, Qwen's relatively lightweight architecture allows for efficient local deployment, making it a practical choice for scenarios where computational resources are limited. This approach not only aligns with our objective of building a lightweight and context-aware system but also emphasizes how RAG techniques can enhance the generative responses and contextual understanding of models that might otherwise be constrained by their architecture.

This comparative analysis provides valuable insights into Qwen's strengths and limitations, illustrating its potential as an effective and efficient solution when paired with external retrieval mechanisms in RAG-based applications.

Model	**Params**	**MMLU**	**C-Eval**	**GSM8K**	**MATH**	**HumanEval**	**MBPP**	**BBH**
		5-shot	**5-shot**	**8-shot**	**4-shot**	**0-shot**	**3-shot**	**3-shot**
Baichuan2	7B	54.7	56.3	24.6	5.6	18.3	24.2	41.6
	13B	59.5	59.0	52.8	10.1	17.1	30.2	49.0
LLaMA	7B	35.6	27.3	11.0	2.9	12.8	17.7	33.5
	13B	47.7	31.8	20.3	4.2	15.8	22.0	37.9
	33B	58.7	37.5	42.3	7.1	21.7	30.2	50.0
	65B	63.7	40.4	54.4	10.6	23.7	37.7	58.4
Llama 2	7B	46.8	32.5	16.7	3.3	12.8	20.8	38.2
	13B	55.0	41.4	29.6	5.0	18.9	30.3	45.6
	34B	62.6	42.6	42.2	6.2	22.6	33.0	44.1
	70B	69.8	50.1	63.6	13.5	29.9	45.0	64.9
QWEN	1.8B	44.6	54.7	21.2	5.6	17.1	14.8	28.2
	7B	58.2	63.5	51.7	11.6	29.9	31.6	45.0
	14B	66.3	72.1	61.3	24.8	32.3	40.8	53.4

Table 4.3: Performance comparison of some open-source base models on different benchmarks [3].

Model	**Params**	**HumanEval**	**MATH**	**GSM8K**
		0-shot	**4-shot**	**8-shot**
Baichuan 2 [58]	7B	18.3	5.6	24.6
LLaMA [49]	7B	12.8	2.9	11.0
Llama 2 [48]	7B	12.8	3.3	16.7
QWEN [3]	1.8B	17.1	5.6	21.2

Table 4.4: Performance comparison of Qwen 1.8B and other 7B models on selected benchmarks.

4.4 Main architecture

Our RAG system is designed to leverage external knowledge sources to enhance the performance and contextual understanding of LLMs. The architecture consists of the following main 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 VSCode 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.

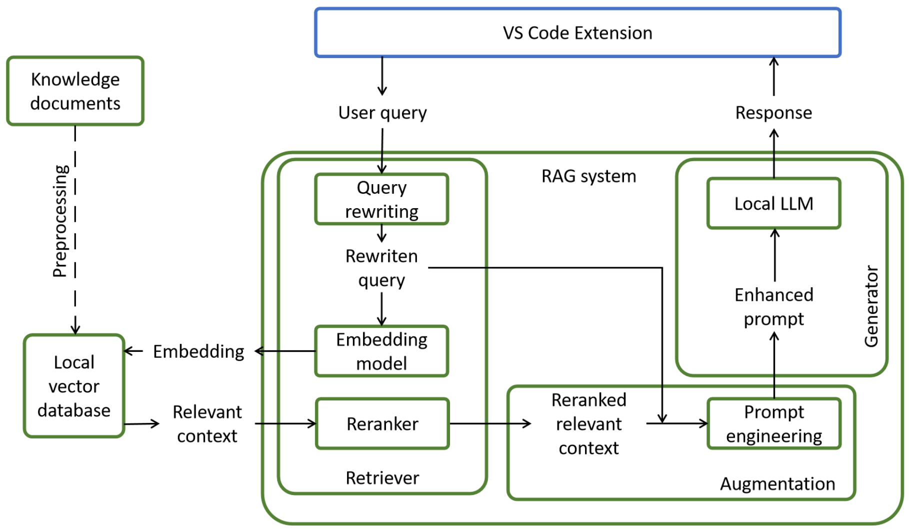


Figure 4.3: Our RAG architecture design.

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 RAG system solves the problems

The proposed RAG 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 VSCode environment, allowing developers to access it without leaving their coding workspace.
- Queries can be sent directly from VSCode, 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.

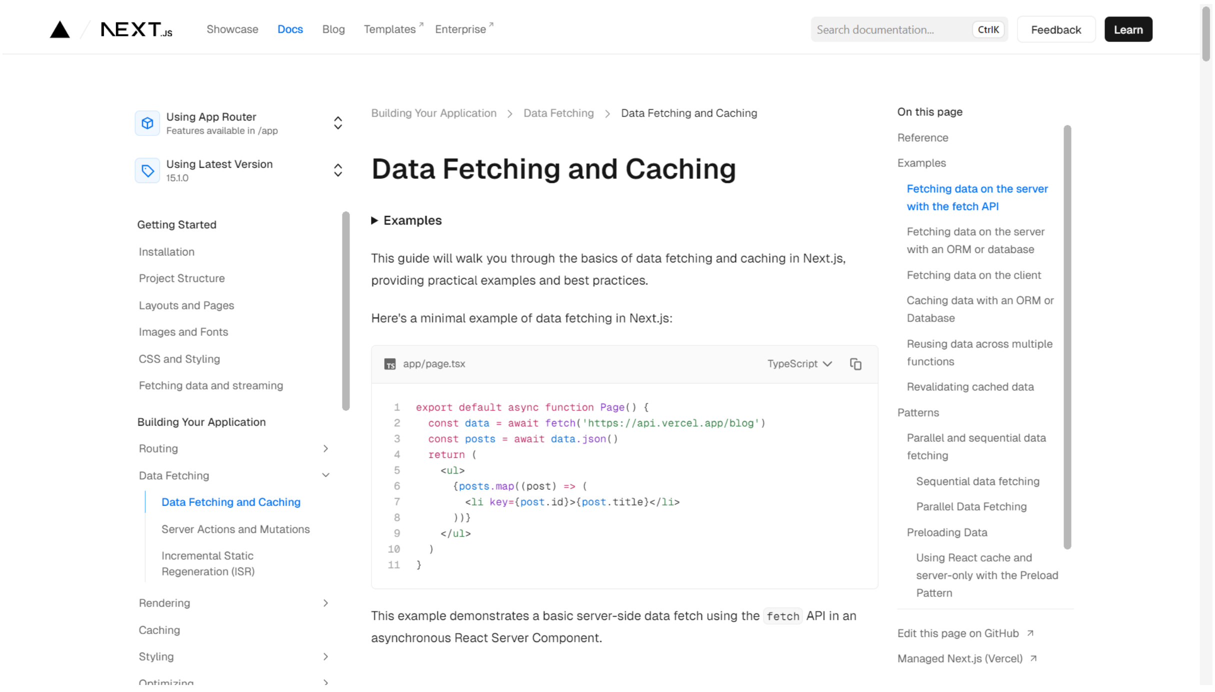
Solutions:

- 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.

4.4.1 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.



NEXT.js
Showcase
Docs
Blog
Templates
Enterprise
Search documentation...
CtrlK
Feedback
Learn
Using App Router
Features available in /app
Using Latest Version
15.1.0
Getting Started
Installation
Project Structure
Layouts and Pages
Images and Fonts
CSS and Styling
Fetching data and streaming
Building Your Application
Routing
Data Fetching
Data Fetching and Caching
Server Actions and Mutations
Incremental Static Regeneration (ISR)
Rendering
Caching
Styling
Optimizing
On this page
Reference
Examples
Fetching data on the server with the fetch API
Fetching data on the server with an ORM or database
Fetching data on the client
Caching data with an ORM or Database
Reusing data across multiple functions
Revalidating cached data
Patterns
Parallel and sequential data fetching
Sequential data fetching
Parallel Data Fetching
Preloading Data
Using React cache and server-only with the Preload Pattern
Edit this page on GitHub
Managed Next.js (Vercel)

Building Your Application > Data Fetching > Data Fetching and Caching

Data Fetching and Caching

▶ Examples

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:

app/page.tsx — TypeScript

```
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 4.4: 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 VSCode. 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 [12, 22]	✓	✗	✓	✗	✗	✗
Microsoft SPTAG [10]	✓	✗	✓	✗	✗	✓
ElasticSearch [13]	✗	✓	✗	✓	✗	✓
Jingdong Vearch [26, 27]	✓	✓	✓	✓	✗	✓
Alibaba AnalyticDB-V [55]	✓	✓	✗	✗	✗	✓
Alibaba PASE (PostgreSQL) [33]	✗	✓	✗	✓	✗	✓
Milvus [51]	✓	✓	✓	✓	✓	✓

Table 4.5: Vector database comparison.

4.4.2 VSCode 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 VSCode 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 VSCode 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 VSCode 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 VSCode 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 VSCode 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.

4.4.3 Feedback loop

Even the most advanced models, such as GPT-4 [32] or local LLMs like LLaMA [49], are not immune to errors or biases. While they excel at generating high-quality responses, these models lack intrinsic mechanisms for discerning truth, consistency, or fairness without external guidance. To implement robust feedback loops in our system, we leverage a combination of automated and human-guided approaches, ensuring comprehensive oversight and continuous refinement.

- Human-in-the-Loop (HITL) Feedback: Users interacting with the VSCode extension provide explicit feedback on the relevance and accuracy of responses, particularly for sensitive or complex queries. For example, when a user submits a query with a code snippet, they can flag issues such as incorrect logic, incomplete explanations, or unhelpful suggestions. These inputs are used to refine the model's understanding and response generation iteratively.
- Reinforcement learning from human feedback (RLHF): RLHF is employed to align the model's outputs with user preferences and ethical guidelines. By introducing a reward system based on user feedback, the model learns to prioritize accuracy, clarity, and helpfulness. This method ensures that the system adapts to the specific needs of the developer community.

5 Initial implementation

For this implementation, we specifically prepared the data for version 15.1.0 of Next.js. In our initial efforts, we concentrated on establishing the foundational components of the system, which included designing the data structure, preparing and preprocessing the data, and configuring both the database and local Large Language Models (LLMs) to develop a functional prototype.

5.1 Database design

Before delving into the structure and preparation of the data, it is important to discuss the data source. Naturally, the official documentation pages of the Next.js framework were our first consideration. Upon examination, the HTML files of the documentation were found to be well-formed and extractable. However, a critical limitation emerged: the official documentation does not maintain a comprehensive version history. The version selection section often changes its listed versions, making it unreliable for our intended use case.

Our goal is to provide users with the ability to select their current framework version and the version they wish to upgrade or downgrade to. This feature requires reliable and comprehensive documentation for all versions.

Fortunately, we identified a more reliable data source: the official GitHub repository of the Next.js framework. This repository provides comprehensive documentation for all versions of the framework in markdown format. These markdown files are well-structured, containing valuable metadata, clearly defined sections, and organized code snippets, making them perfectly suited for our use case. The stable release of version 15.1.0 alone includes 578 markdown files. To ensure broad compatibility, we plan to support versions from 13.0.0 to 15.1.0, encompassing 95 stable versions. Canary versions are excluded from our dataset, as their changes are eventually consolidated in the corresponding stable releases.

5.1.1 Data structure design

The purpose of our data structure design is to store the Next.js documentation effectively. After analyzing the structure of the markdown files retrieved from the repository, the following key components were identified:

- **Metadata:** Contains information such as:
  - `title`: The title of the documentation page.
  - `nav_title`: The title displayed in the navigation bar.
  - `description`: A brief description of the page's content.
  - `related`: An optional field containing links to other related documentation pages.
- **Text:** The main content of the documentation, divided into sections and subsections using markdown syntax.
- **Code snippets:** Includes:

  - `language`: Programming language used in the snippet (e.g., JavaScript or TypeScript).
  - `file`: Indicates the file or component to which the code belongs (Next.js follows a naming convention for files).
  - `switcher`: Indicates whether the code snippet has a counterpart in another language (e.g., JavaScript/TypeScript switcher).
  - `content`: The actual code content.

- **Version history:** A special section explicitly listing major updates to the documentation across versions. However, this section is present in only a small portion of the documentation.

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]).

Based on these components, we designed a schema to organize and store the documentation data efficiently.

Collection: `nextjs_docs`

This single collection will store all data, including documents, sections, and code snippets. Each record will represent one of the following:

- A **document**: Represents a top-level page in the documentation.
- A **section**: Represents a subsection within a document.
- A **code snippet**: Represents a code example belonging to a section.

5.1.2 Fields in the schema

Primary scalar fields

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

Field name	Description
`entry_id` (Primary Key)	Unique identifier for each entry.
`tag`	A tag derived from `nav_title` to indicate the subject or topic of the entry. Useful for categorizing and filtering entries by subject.
`entry_type`	Type of entry (`document`, `section`, or `code_snippet`).
`parent_id`	The `entry_id` of the parent entry. For: • **Sections:** Links to the document or other section it belongs to. • **Code snippets:** Links to the section it belongs to.
`title`	Title of the document, section, or code snippet.
`description`	Description from `description` metadata.
`metadata`	JSON or string field for custom metadata (e.g., programming language, or related links).
`version`	Framework version associated with the entry (e.g., `"15.1.0"`).

Table 5.2: Primary scalar fields in the schema.

Text fields

Field name	Description .
`text_content`	The raw text content (e.g., document or section body).
`code_content`	The actual code snippet (only for entries of type `code_snippet`).

Table 5.3: Text field in the schema.

Embedding vectors

Field name	Index	Metric
`sparse_title_description`	`SPARSE_INVERTED_INDEX`	Inner Product
`dense_text_content`	`HNSW`	Cosine Similarity (Dimension: 1024)
`dense_code_snippet`	`HNSW`	Cosine Similarity (Dimension: 1024)

Table 5.4: Embedding vectors in the schema.

Partitioning Field: `version`
This field is used to partition entries by `version`. Partitioning aligns with Milvus's capability for managing partitions, ensuring efficient queries when users specify the framework version.

5.2 Data preparation

As described in the previous section, our primary data source is the GitHub repository of the Next.js framework. Initially, we developed a crawler that recursively navigated and downloaded resources from the `docs` and `errors` folders of the repository. However, this approach faced significant limitations due to GitHub's API rate limits. After careful evaluation, we adopted a more efficient and native method to retrieve data directly from the GitHub repository.

Steps in data preparation:

- **Data acquisition:** To extract the necessary files, we implemented a Python script that utilizes Git's sparse-checkout feature. This approach enabled us to selectively download specific folders, such as `docs` and `errors`, corresponding to a particular version tag. The script automates tasks including cloning the repository, configuring sparse-checkout, and checking out the desired version tag.

- **Data organization:** The retrieved folders and their contents were stored in version-specific directories. For instance, for the version `v15.1.0`, the directory structure appeared as follows:

  This structure provides an organized format, facilitating seamless access and retrieval of documentation resources for different framework versions.

- **Data integrity check:** To ensure the integrity and relevance of the dataset, the script included steps to validate and clean the retrieved data. Non-essential files and directories were removed, leaving only the targeted folders and their respective contents. This process ensured that the dataset was well-structured and ready for subsequent preprocessing.

5.3 Data preprocessing

Once the data was acquired and organized, the next critical step was preprocessing it to ensure compatibility with the schema design for effective storage and retrieval. This process involved converting raw markdown files into structured formats through a preprocessing pipeline implemented in Python. The pipeline utilized markdown parsing libraries, embedding models, and structured data processing to extract meaningful information from the documentation.

Key preprocessing steps

- **Metadata extraction:** The metadata fields, such as `title`, `description`, and `nav_title`, were extracted from markdown files using the `frontmatter` library. Additional metadata, such as `related` links, was included if available. Each document was assigned a unique `entry_id` to enable hierarchical referencing within the schema.

  Example code for generating a unique entry ID:

```
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.

- **Data structuring:** The extracted sections were structured as dictionaries containing fields such as section title, textual content, and associated code snippets. This data organization was designed to align with the schema in the database for seamless integration.

- **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.

- **Validation and cleanup:** Parsed sections and their content were validated for completeness and correctness. Empty or incomplete sections were flagged for review, and non-essential formatting artifacts were removed to ensure clean and reliable data.

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.
- **Batch processing and storage:** Processed entries were saved in batches as CSV files, with fields such as `entry_id`, `entry_type`, `text_content`, and embeddings. This modular strategy supported scalability and efficient handling of large datasets.

The preprocessing resulted in 4,714 data entries stored in a CSV file. A total of 481 files were processed, while 97 files were skipped because their content was generated by other files.

5.4 Database setup

Setting up the database involved configuring Milvus for effective data storage, indexing, and retrieval. This process included deploying Milvus, defining the schema, creating indices, and inserting as well as retrieving data.

5.4.1 Deploying Milvus with Docker Compose

Milvus was deployed using Docker Compose. After installing Docker, the configuration file was downloaded, and the required containers were initialized. This setup resulted in three key containers:

- **milvus-etcd:** Responsible for managing metadata storage.
- **milvus-minio:** Manages object storage for the system.
- **milvus-standalone:** Handles vector indexing and retrieval services.

5.4.2 Installing PyMilvus

The `PyMilvus` library was installed to allow programmatic interaction with the Milvus server. To ensure compatibility, the version of `PyMilvus` matched the Milvus server version (2.4).

Schema definition and collection creation

As outlined in the data structure design, the database schema was implemented with the collection name `nextjs_docs`.

5.4.3 Creating indices

To optimize retrieval performance, indices were created for each of the three vector fields:

- **Sparse index:** Built for the `sparse_title_description` field using an inverted index with inner product (`IP`) as the metric type.
- **Dense text index:** Applied to the `dense_text_content` field using the `HNSW` algorithm with cosine similarity.
- **Dense code index:** Created for the `dense_code_snippet` field, also utilizing the `HNSW` algorithm with cosine similarity.

5.4.4 Inserting data into the Collection

The preprocessed data, stored as a CSV file, was read and validated to ensure all data types matched the schema requirements.

5.4.5 Hybrid search implementation

The hybrid search function was implemented as a prototype to illustrate the retrieval process by integrating sparse and dense embeddings.

Workflow of hybrid search:

1. **Input and initialization:** Sparse and dense embeddings are provided as inputs, along with weights to balance their influence. Sparse embeddings are processed using the `sparse_search` function, while dense embeddings are handled by the `dense_text_search` function.

2. **Sparse search:** This step retrieves entries based on keyword relevance. The output includes the entry's metadata and a distance metric.

3. **Dense search:** This step retrieves entries based on semantic similarity, outputting the entry's metadata and a distance metric.

4. **Combining results:** The results from sparse and dense searches are merged using `entry_id` as the common key. A combined score is calculated for each entry:

   $$\text{Combined score} = (\text{Sparse distance} \times \text{Sparse weight}) + (\text{Dense distance} \times \text{Dense weight})$$

   If an entry is missing in either result, its corresponding distance is set to zero.

5. **Sorting and limiting results:** The entries are ranked by their combined scores in descending order. The top `limit` results are returned, demonstrating the hybrid retrieval process.

5.5 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[2], 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.

5.5.1 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.

5.5.2 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`

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:

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

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

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

Alternatively, Ollama models can also be integrated using the Python library LangChain[3]. With LangChain, the locally installed Ollama models can be accessed and utilized as part of a more comprehensive pipeline, allowing for advanced customization and ease of interaction. This flexibility enables developers to build robust RAG systems that leverage the full potential of Ollama's capabilities while adapting to diverse use cases.

[3]LangChain - `https://python.langchain.com/docs/introduction/`

6 Evaluation plan

6.1 Evaluation metrics

6.1.1 Retriever metrics

Recall@K: Recall@K measures the proportion of relevant documents retrieved within the top K results for a query.

$$\text{Recall@K} = \frac{|\text{Relevant documents} \cap \text{Retrieved top K documents}|}{|\text{Relevant documents}|}$$

where:

- $|\text{Relevant documents}|$ is the total number of relevant documents for the query.
- $|\text{Relevant documents} \cap \text{Retrieved top K documents}|$ is the number of relevant documents retrieved in the top K results.

Example:

- Scenario: A query retrieves the top $K = 5$ documents. Out of these, 3 are relevant, and the total number of relevant documents is 4.
- Calculation:
$$\text{Recall@5} = \frac{3}{4} = 0.75, (75\%)$$

Precision@K: Precision@K evaluates the accuracy of the retrieved documents within the top K results.

$$\text{Precision@K} = \frac{|\text{Relevant documents} \cap \text{Retrieved top K documents}|}{|\text{Retrieved top K documents}|}$$

where:

- $|\text{Retrieved top K documents}|$ is the total number of documents retrieved in the top K results.

Example:

- Scenario: A query retrieves the top $K = 5$ documents. Out of these, 3 are relevant, and the total number of relevant documents is 4.
- Calculation:
$$\text{Precision@5} = \frac{3}{5} = 0.6\,(60\%)$$

Latency: Latency measures the time taken to retrieve documents for a query.

$$\text{Latency (ms)} = \text{Time when results are returned} - \text{Time when query is submitted}$$

Example:

- Scenario: LLM receives a prompt at 11:00:00.000 and receives the results at 11:00:00.340.
- Calculation:

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

Coverage: Coverage assesses the percentage of queries for which at least one relevant document is retrieved.

$$\text{Coverage} = \frac{\text{Number of queries with relevant results retrieved}}{\text{Total number of queries}} \times 100$$

Example:

- Scenario: A total of 10 queries are submitted, and the system retrieves relevant results for 8 queries.
- Calculation:

$$\text{Coverage} = \left(\frac{8}{10}\right) \times 100 = 80\%$$

6.1.2 Generator metrics

ROUGE score: ROUGE evaluates the overlap between the generated text and reference outputs. For example, ROUGE-N is defined as:

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

Example:

- Scenario: A reference document contains 10 bigrams, and the system retrieves 8 bigrams that match with the reference.
- Calculation:

$$\text{ROUGE-2} = \frac{\text{8 bigram overlaps}}{\text{10 bigrams in reference}} = 0.8$$

Relevance: Relevance is assessed manually by rating how well the generated outputs align with the intent of user queries. Scores are averaged across all queries:

$$\text{Relevance} = \frac{\sum_{i=1}^{N} \text{Relevance score for query } i}{N}$$

Example:

- Scenario: There are 5 queries, and their relevance scores are: $0.9, 0.8, 0.7, 1.0,$ and 0.6.
- Calculation:

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

Factual accuracy: Factual accuracy measures the percentage of generated outputs that are factually correct based on the retrieved data:

$$\text{Factual accuracy} = \frac{\text{Number of factually correct outputs}}{\text{Total number of outputs}} \times 100$$

Example:

- Scenario: Out of 20 outputs, 16 outputs are factually correct.
- Calculation:

$$\text{Factual Accuracy} = \left(\frac{16}{20}\right) \times 100 = 80\%$$

Latency: Latency for the generator is calculated similarly to the retriever:

$$\text{Latency (ms)} = \text{Time when output is generated} - \text{Time when retrieval is completed}$$

Example:

- Scenario: LLM receives a prompt at 12:00:00.000 and receives the results at 12:00:00.210.
- Calculation:

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

6.2 Evaluation methodology

6.2.1 Retriever evaluation

We will use the pre-defined dataset with known relevant results to benchmark the retriever's performance across query types:

- **Simple queries:** Basic queries requiring straightforward retrieval of documents.
- **Complex queries:** Multi-faceted queries involving multiple conditions or ambiguities.
- **Context-dependent queries:** Queries that rely on contextual understanding of the current repository.

6.2.2 Generator evaluation

Provide the generator with retriever outputs and evaluate the quality of generated content using:

- **Automatic metrics:** ROUGE scores to quantify similarity and overlap with reference outputs.
- **Human evaluation:** Involve domain experts to rate the relevance, accuracy, and quality of generated outputs on a predefined scale.

Include edge cases in testing, such as incomplete or ambiguous retriever outputs, to ensure the generator's robustness.

6.3 Test plans

Since our RAG model will be integrated into a VSCode extension, we will utilize VSCode's support for running and debugging tests through integration tests. The system's performance will be evaluated using the following methods:

6.3.1 Unit testing

Validate individual functions and components of the extension.

Focus areas Ensure all API calls return expected results and handle errors gracefully. Verify that data processing components correctly transform input data into the desired output format. Additionally, confirm the correctness of the generated output for a variety of given inputs, including edge cases.

Methods: Write unit tests using Mocha or Jest to ensure each function behaves as expected. Mock the VSCode API to isolate and test individual components without dependencies.

6.3.2 Integration testing

Ensure proper interaction between extension components and the VSCode environment.

Focus areas: Validate commands and features exposed by the extension. Ensure that commands are correctly registered and executed, and that features such as code completion, linting, and formatting work as intended. Test the extension in a simulated VSCode instance to verify its behavior in a real-world scenario.

Methods: Use the @vscode/test-electron API for integration testing. Write integration tests to simulate user interactions with the extension. For example, test the execution of commands, the response to user inputs, and the interaction with the VSCode UI and then configure launch.json to enable debugging of integration tests.

6.3.3 System testing

Test the extension as a whole to evaluate its usability and performance in real scenarios.

Focus areas: Perform comprehensive end-to-end tasks such as activating commands, editing files, and interacting with the VSCode UI. Ensure the extension works seamlessly with other third-party extensions and does not cause any conflicts or unexpected behavior.

Methods: Use automated test suites to re-test core functionalities. Automated test suites help streamline the regression testing process by executing pre-defined test cases efficiently and consistently.

They ensure that recent updates or changes do not negatively impact existing features. Additionally, maintain a comprehensive log of identified bugs, fixes, and test outcomes. This log serves as a valuable reference for tracking recurring issues, validating the effectiveness of applied solutions, and guiding future development efforts. Consistently reviewing and updating these logs ensures better understanding and management of the extension's quality over time.

6.3.4 Performance testing

Assess the extension's speed, resource usage, and responsiveness.

Focus areas: Measure activation time and execution of commands. Monitor memory and CPU usage during operations. Evaluate response times under different network speeds to simulate various user environments. Load test the extension under normal and peak conditions to analyze its behavior. Conduct stress tests to identify breaking points when pushed beyond normal workloads during peak times. Assess recovery mechanisms and evaluate how the extension handles high load or failure events, ensuring it can recover gracefully.

Methods: Use performance monitoring tools and built-in VSCode performance diagnostics. Profile heavy operations using the VSCode Performance Profiler. Simulate different network conditions and user workflows for comprehensive testing.

7 Conclusion and future works

7.1 Conclusion

Our work began in early September 2024, where we selected the problem to address based on our interests and familiarity with the domain. The problem definition phase focused on exploring how we can utilize LLMs to facilitate the development of a rapidly evolving framework while adhering to the latest guidelines. Concurrently, we conducted research on background knowledge.

We started by reading surveys on RAG systems [64, 14] and delved into its core components:

- **Retriever:** We began by determining how to represent data for semantic search. BGE-M3 [8] emerged as a suitable choice for English, supporting multiple search methods including sparse, dense, multi-vector, and hybrid search. To facilitate the search process efficiently, we studied KNN and its limitations, particularly regarding scalability in large databases. To address this, we adopted HNSW [29], which offers a balanced trade-off between accuracy and scalability for vector search.
- **Generator:** For the generator, we built upon the Transformer architecture [50], which serves as the foundation for LLMs, decoder-only transformer. We opted for Qwen [3] as our model for generating text and code within the system.

Following this foundation, we investigated how augmentation could be integrated. We chose to augment the system during the inference stage, inspired by RALM [38]. In essence, this stage occurs before the prompt is passed into the LLM, where we inject documentation content into the prompt via string concatenation.

The augmentation source consists of structured and unstructured data derived from the Next.js documentation. The structured data pertains to the organization of the documentation, while the unstructured data includes code samples. To manage this data effectively, we adopted the KnowledgeGPT framework [54], organizing the data into a knowledge base while supporting semantic search for unstructured content. For the augmentation process, we applied adaptive methods to determine when external information is necessary to generate accurate answers, utilizing reflection tokens as implemented in SELF-RAG [1].

Based on our supervisors' suggestions, we decided to deliver our solution as a VSCode extension. To inform this decision, we analyzed related tools by examining their retrieval mechanisms, code generation capabilities, and integration within the development process. After a thorough comparison, we decided that our solution would run entirely locally on the user's machine, focusing on performance with limited resources. This approach allows us to evaluate our solution against existing cloud-based, paid alternatives like GitHub Copilot[1].

Subsequently, we finalized our approach and designed a high-level architecture (Figure 4.3). For the database, we chose Milvus [51] due to its open-source nature and comprehensive SDK support.

We proceeded with the initial implementation of the system, starting with data preparation. This involved identifying the data source, extracting content from the official Next.js GitHub repository, and implementing preprocessing scripts to clean and process markdown files. We then embedded the data and stored it in the database. As a preliminary milestone, we developed a demonstration script capable of taking an input string and retrieving relevant documents from the database. This phase did not yet include prompt engineering or advanced retrieval mechanisms.

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

We then outlined our evaluation strategy, which focuses on two key components:

1. Evaluating the RAG components: retriever and generator performance.
2. Testing the VSCode extension as a software product.

Report writing began in October 2024, based on slides prepared for weekly supervisor meetings. We continued to write and refine the report following feedback and guidance from our supervisors.

7.2 Upcoming works: January 2025 and beyond

1. **Preparing local LLM and embedding system (November 2024 – January 2025):** Setup and configure the local LLM and embedding system to handle efficient information retrieval.
2. **Collecting and organizing the dataset (December 2024 – February 2025):** Data collection and preprocessing efforts will focus on organizing high-quality and relevant datasets.
3. **Configuring vector database (December 2024 – February 2025):** Establish the vector database to efficiently store and retrieve embeddings for core functionalities.
4. **Implementing query handling and prompt management (December 2024 – February 2025):** Mechanisms for managing user queries and prompt workflows will be implemented for smooth operations.
5. **Evaluating retriever and generator (February – March 2025):** Assessment of retrieval and generation components for accuracy, efficiency, and robustness.
6. **Integrating into VSCode (February – April 2025):** The system will be integrated into the VSCode environment as an extension to improve developer accessibility.
7. **Implementing user interface for interactions (January – April 2025):** Develop an intuitive and interactive user interface to ensure seamless user experience.
8. **Evaluating the extension's performance (March – April 2025):** Performance testing of the VSCode extension will focus on usability, responsiveness, and reliability.
9. **Optimizing the system (March – May 2025):** Optimize the system to enhance efficiency, accuracy, and overall performance.
10. **Deployment (April – May 2025):** Fully develop, test, and deploy the optimized system for production use.
11. **Continue writing report (December 2024 – May 2025):** Prepare the final report summarizing the problem definition, research, implementation, evaluation, and findings.

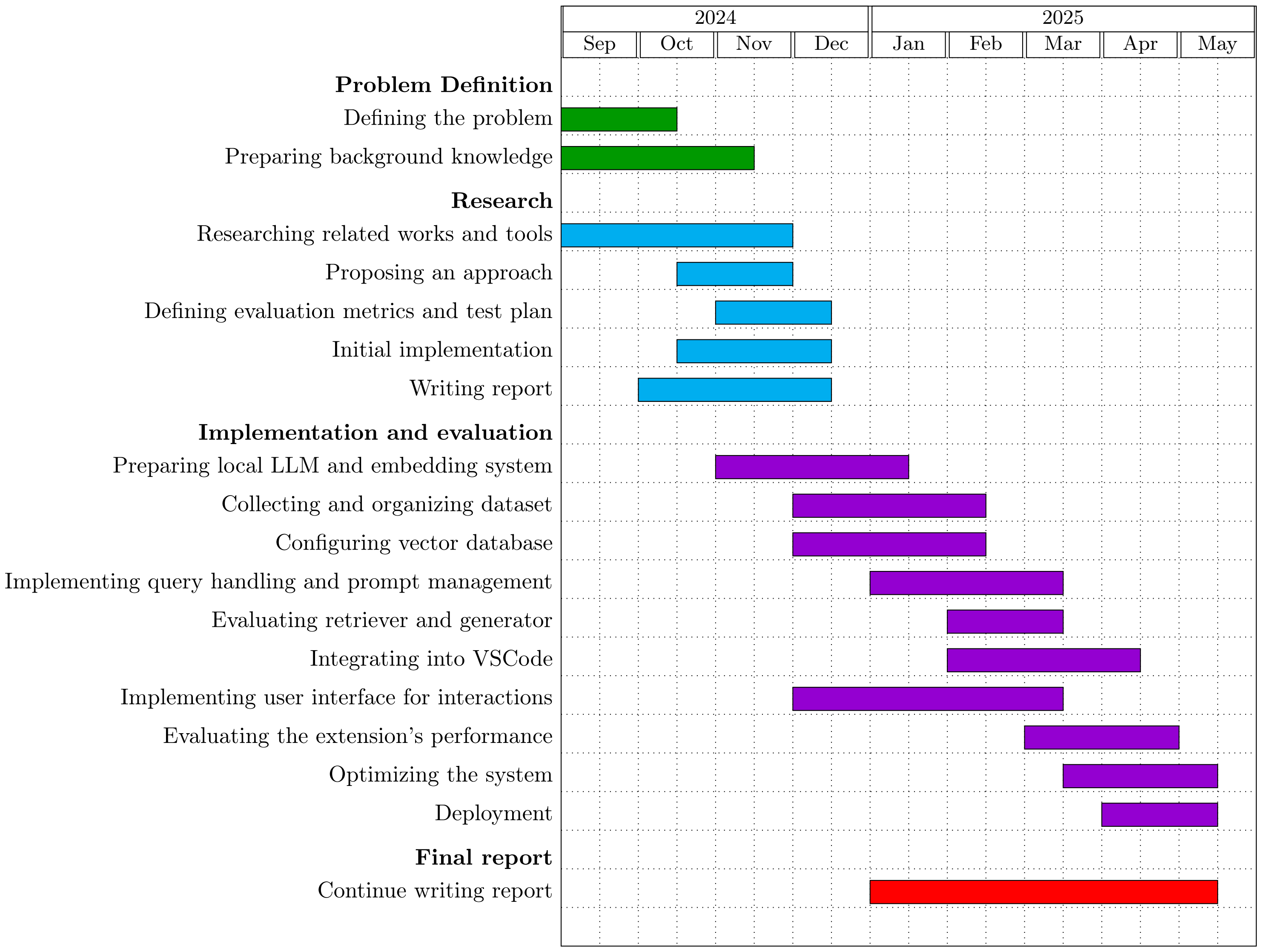


Figure 7.1: Future works.

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] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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`.

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

[20] 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`.

[21] 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`.

[22] 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`.

[23] 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`.

[24] 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.

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

[26] 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`.

[27] 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`.

[28] 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`.

[29] 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`.

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

[31] 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`.

[32] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: `2303.08774 [cs.CL]`. URL: `https://arxiv.org/abs/2303.08774`.

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

[34] 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`.

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

[36] 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.

[37] 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`.

[38] 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`.

[39] 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`.

[40] 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`.

[41] 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`.

[42] 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.

[43] 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.

[44] 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`.

[45] 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.

[46] 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`.

[47] 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`.

[48] 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`.

[49] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: `2302.13971 [cs.CL]`. URL: `https://arxiv.org/abs/2302.13971`.

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

[51] 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`.

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

[53] 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`.

[54] 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`.

[55] 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`.

[56] 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`.

[57] 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.

[58] Aiyuan Yang et al. *Baichuan 2: Open Large-scale Language Models.* 2023. arXiv: 2309.10305 [cs.CL]. URL: https://arxiv.org/abs/2309.10305.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] Penghao Zhao et al. *Retrieval-Augmented Generation for AI-Generated Content: A Survey.* 2024. arXiv: 2402.19473 [cs.CV]. URL: https://arxiv.org/abs/2402.19473.

Appendices

A
Interface of the RAGGIN Extension

B
Test Results