TTDS Group Project - Code Search Engine

s2704516 s2051283 s2716530 s2585198 s2436122

Feb 2025

Abstract

The CodeMe search engine is a specialized information retrieval system designed to help developers efficiently find relevant code snippets and discussions. Built on the Stack Overflow dataset, this system implements domain-specific preprocessing techniques that recognize code structures and programming conventions like camel-case and snake-case tokenization. The search platform employs a custom-built inverted index with delta encoding and variable integer compression, reducing storage requirements by 30% while maintaining efficient query processing. Advanced retrieval features include BM25-based ranking enhanced with query expansion using CodeBERT embeddings, spell-checking, and sophisticated reranking through metadata analysis and language model (LM) similarity scoring. The system supports boolean, phrase and proximity search for precise queries. User testing demonstrates positive clarity scores and reasonable response times (averaging 1.4 seconds), confirming the effectiveness of our approach in providing relevant programming solutions through an intuitive, filter-enabled interface. The search engine can be accessed here http://35.214.27.195:5000/.

1 Introduction

In today's fast-paced software development world, programmers face complex challenges that require quick access to relevant solutions. Despite the abundance of online resources, developers still spend a significant amount of time searching for code examples, documentation, and discussions. Our project aims to address this inefficiency. General-purpose search engines, while useful, lack the specialized ability to interpret programming syntax, recognize code patterns, and differentiate functionally distinct solutions. This gap in search technology hinders developer productivity and knowledge sharing within the programming community.

Our work addresses this challenge by developing a dedicated search engine optimized specifically for coding queries. By leveraging the rich repository of programming knowledge contained in the Stack Overflow dump dataset, we aim to create a system that understands not just keywords, but the semantic meaning behind coding questions and the structural elements of programming solutions. This search platform incorporates domain-specific preprocessing techniques that respect code structures, advanced indexing strategies optimized for programming content, and intelligent query processing methods including context-aware expansion. These elements work in concert to deliver precisely relevant results that align with the programmer's intent, not merely their literal query. In the following sections, we describe our implementation process in detail, beginning with data acquisition and preprocessing, followed by our indexing approach and search optimization techniques. We conclude with a comprehensive evaluation of the system's performance and its implications for future work in this domain.

2 Data

2.1 Data Collection

Stack Exchange releases data dumps of its publicly available content on a quarterly basis at archive.org. This data dump includes all Stack Exchange sites (e.g., math.stackexchange), but we have restricted our collection to the Stack Overflow dataset that focuses on programming and software development. The data is officially distributed as XML files, which is ingested into a PostgreSQL database as a permanent backup of data where the schema is kept consistent with StackOverflow's internal database schema for structured data management, efficient query execution, and scalability. The database serves as persistent storage, but our primary focus is on the *Posts* table, which contains all Stack Overflow content, including questions, answers, and wiki posts. With a total of 59,974,112 posts and approximately 54GB of data,

this table stores not only the title and body of each post but also key metadata such as the author's ID, creation date, score (net votes of upvotes minus downvotes), view count, comment count, favorite count, and associated tags. Our project further leverages this metadata to enhance scoring mechanisms and enable users to filter and sort posts based on various criteria.

2.2 Data preprocessing

2.2.1 Parsing

Stack Overflow posts present a unique challenge for search engines as they contain two fundamentally different types of content: lexical text and code snippets. To address this, we designed a preprocessing pipeline that efficiently handles both formats while preserving their structure and meaning, yielding highly promising results. Each post is originally stored as an HTML document, containing structured elements such as paragraphs, blockquotes, links, and code blocks. To extract relevant content, we built an HTML parser that categorizes each part of a post into three types of content blocks: *TextBlock* - regular text, including explanations and discussions; *CodeBlock* - includes programming code, extracted from `<code>...</code>` tags; and *LinkBlock* – includes links to external documentation or resources. Each post is represented as a list of blocks which are preprocessed by the *Preprocessing* module which contains three distinct stages: *pre-tokenization normalization*, *tokenization* and *post-tokenization normalization*.

2.2.2 Preprocessing

Our tokenization approach combines established natural language processing techniques with specialized code-aware methods to create an optimal representation of programming content. Beginning with *unicode-to-ascii* normalization, we eliminate accents and diacritics from tokens, creating a standardized character set that improves matching efficiency and reduces index complexity. For the crucial tokenization phase, we designed a hybrid approach guided by the comprehensive analysis of code retrieval methods in [8]. Their findings demonstrated that code-specific tokenization strategies, particularly those addressing common programming naming conventions like snake-case and camel-case, significantly enhance the performance of traditional information retrieval models like BM25 and RM3.We therefore apply tokenization in the following order:

1. Whitespace Splitting → Obtain initial tokens with the regex '\s+'
2. Link Removal → Remove any links
3. Digit Removal → Remove any digits
4. Punctuation Splitting → Remove all punctuation and replace with a space aside from contractions (e.g. "don't") which are combined
5. Camel Case Splitting → Splits camel case words (e.g. "maxTime", "MaxTime")
6. Snake Case Splitting → Splits snake case words (e.g. "max_time")

Our token normalization process combines multiple specialized techniques to optimize code-related search performance. After tokenization, tokens undergo stemming using the Porter Stemmer algorithm [5] implemented via PyStemmer, which reduces words to their root forms for improved matching. In our initial analysis this proved to be beneficial, although a more custom Stemmer could have been useful. We then apply a custom stop-word removal strategy leveraging information-theoretic approach as discussed in [1], which uses a specialized list derived from Stack Overflow content rather than generic stop-word lists, preserving programming-relevant terminology while eliminating truly non-discriminative terms. All tokens are then lowercased for consistency, followed by Unicode-to-ASCII normalization to standardize character representation, reduce encoding variance, and ensure compatibility with our index storage format. This carefully calibrated normalization pipeline, developed through extensive iterative testing against real-world programming queries, significantly enhances retrieval precision for code-specific content, demonstrating measurable improvements over generic text processing approaches typically employed by general-purpose search engines.

3 Inverted Index

The inverted index itself is built entirely from scratch and as such, we are not reliant on external technologies or services to provide our data. We build the index in a parallel fashion such that multiple sub-shards

are generated and merged to form optimally sized shards for searching. Each shard is composed of a subset of documents and as such, each shard can be treated as an independent "sub-index". This was done with scalability in mind as when a new Data Dump gets released it will be easily integrated into the current system by incrementally indexing only the new data rather than rebuilding the entire index from scratch. When we search across the index, results are obtained independently and merged with the other shards to form a final result. Each "sub-index" includes the following files and information: a term-level document frequency (DF) file, an offset file which contains the term and its offset into index to quickly locate a term's postings and its positions fields and finally the the position file which contains each terms position information. This structure performed very well in terms of computational efficiency for faster lookups.

Component	**Description**
Document Frequency	Number of docs containing term (Variable-length Integer)
Document-Term Block List	A list of Document-Term Blocks (size = Document Frequency)
Document-Term Block	[DocumentID Delta, Term Frequency, Position Frequency]
DocumentID Delta	Stores document IDs as deltas (Variable-length Integer)
Document Term Frequency (TF)	Number of times the term appears in the document
Position Frequency	Number of positions recorded for the term in the document

Table 1: Index File Components

We also utilized Delta Encoding so that we are able to compress our index more effectively, which proved to be very useful as it reduced our storage by approximately 30%. Likewise, each integer is a variable integer so that we can encode numbers into different sizes and allowing for maximal compression.

File	**Contents**
Positions File	Stores *Position Delta* values to encode term positions efficiently
Offsets File	Stores *UTF-8 term size, term bytes, index offset, and position offset* for fast lookups
Document Files	Contains *metadata, body, and title* of posts for retrieval

Table 2: Additional File Components

For the offset file, each offset serves as a seek position for terms in their respective files. The system loads this file into an FST using the *marisa-trie* library, enabling efficient term lookups and prefix operations. This structure also facilitates future features like wildcard search. For the document-level files; information, metadata, body, and title are stored in a binary files for efficient retrieval. BM25 computation relies on document lengths, which are tracked during indexing and stored in a memory-mapped binary file for fast access during retrieval. Each term in the offset file, indexes into both the index and positions file. These files are separated so that we are not forced to decode positions, which is unnecessary for most queries.

Our collection of 59 million posts is compressed into a 34GB optimized search index: 16GB for the term-level inverted index and 18GB for document-level information. In contrast, storing the raw posts in PostgreSQL takes 54GB, with an unoptimized index reaching 50GB. By applying delta and variable integer encoding, we reduce storage from 50GB to 34GB, significantly improving efficiency over traditional database indexing. The index is also designed with scalability in mind, in case we would use a live indexing system. A persistent index will store the existing data, while a live index will handle new documents from PostgreSQL until it grows large enough for integration. It can be added as a new shard or merged with the last shard, ensuring minimal overwrites since each sub-index functions independently.

The ability to search this index is heavily constrained by the sheer scale of the index and as such, a large number of optimisations were carried out to ensure that we could search through the index reasonably quickly. These include: searching through the shards in parallel and merging to form the final result and using Numba JIT compiled functions for computationally expensive operations.

4 Searches

4.1 Ranked Retrieval

The most frequent type of search query in our system will involve users seeking solutions or discussions to coding-related questions. To address this type of search, we initially experimented with TF-IDF and its probabilistic extension, BM25, with the latter proving to be more effective based on our initial manual evaluations, perhaps due to its document length normalization and stronger emphasis on rare but important terms. BM25 [6] ranks documents based on term frequency (TF) and inverse document frequency (IDF) while incorporating document length normalization to ensure fair ranking. It ultimately computes a cumulative score for each document based on all query terms, determining its relevance ranking. Higher relevance scores are assigned to documents where query terms appear more frequently, while overly common terms are downweighted. It also prevents longer documents from gaining an unfair advantage simply due to higher word counts, resulting in a more balanced and effective retrieval process.

$$\text{BM25}(q,d) = \sum_{t \in q} IDF(t) \cdot \frac{f(t,d) \cdot (k_1 + 1)}{f(t,d) + k_1 \cdot \left(1 - b + b \cdot \frac{|d|}{\text{avg}|d|}\right)}$$

$$IDF(t) = \log\left(\frac{N - n_t + 0.5}{n_t + 0.5} + 1\right)$$

where q is the query containing terms t, d is the document being scored, $f(t,d)$ is the term frequency of t in document d, $|d|$ is the length of document d, N is the total number of documents in the corpus and finally n_t is the number of documents containing term t. The parameters k_1 and b were set to their default values of 1.5 and 0.75, respectively.

One key aspect of our retrieval approach is scoring query terms based on their importance, ensuring that rarer terms contribute more to a document's ranking. We achieve this by processing terms in decreasing inverse document frequency (IDF) order, meaning less common words have a greater impact on the final score. To refine the search results, we first attempt to return only documents that contain all query terms (intersection). However, if this results in too few matches, we expand the search to include documents that contain at least one query term (union), ensuring a sufficient number of results while maintaining relevance.

One challenge we initially faced was handling sparse query matches, either due to rare terms with limited occurrences in our index or words that were absent. To address this, the system automatically backfills results by including the results of a semantic-style search where we use the model developed for Section 5.4.2 to find semantically-relevant documents. If this fails, we use random additional documents to ensure more comprehensive results list. Another challenge was processing very long queries, which significantly slowed down retrieval. To mitigate this, the search engine selects a subset of terms from the query and ranks documents based on those, improving efficiency while maintaining relevance.

Beyond BM25, we implemented additional ranked retrieval enhancements, including *query expansion* using precomputed word embeddings to improve recall by adding semantically related terms, a *spell-checking* module to handle common user typos and two *learning-to-rank (L2R)* modules leveraging firstly language models (LMs) and secondly metadata-based heuristics. A thorough description of these can be found in Section 5.4.

4.2 Advanced Search

Understanding our users' needs, we anticipate that many will require more advanced search functionalities beyond simple ranked retrieval. This includes Boolean search using the AND, OR, and NOT operators, as well as phrase search and, to a lesser extent proximity search, which will be described below. Users can enable these features through the "Advanced Search" button. Boolean search is automatically activated when a query includes Boolean operators or it gets redirected to phrase or proximity search if it follows their predefined commonly used syntax, namely using quotes ("enter phrase") and #N(term1,term2) respectively. To ensure correct interpretation, we implemented a stack-based parsing method for efficient query processing.

4.2.1 Boolean Search

Boolean search is a core component of our system, allowing users to define query logic explicitly using AND, OR, and NOT operators. We implemented a tree-based parser that processes queries by retrieving

terms at the leaf nodes and computing set operations (intersection, union, and complement) at the internal nodes. To ensure flexibility and accuracy, our Boolean search module supports nested expressions (e.g., "x AND y OR z") while respecting operator precedence (NOT > AND > OR). For correctness, the system does not apply filtering or pruning optimizations. Instead, it fully evaluates each query, ensuring that results strictly follow Boolean logic. However, this means that query performance degrades as complexity increases, a tradeoff that advanced users familiar with Boolean search in other systems will recognize. Again, if limited results are found then the system automatically backfills results.

4.2.2 Phrase and Proximity Search

For more precise retrieval our system also includes phrase and proximity search. For the phrase search users can wrap terms in quotes (e.g., "data science") to find documents containing the exact phrase in the specified order. For the proximity Search users can use the syntax '#N(term1 term2)' to search for terms appearing within N words of each other. For example, '#5(python tutorial)' retrieves documents where "python" and "tutorial" occur within five words. The implementation of this system is based primarily on the intersection of each term's posting lists and the additional positions for each term in each document.

5 Features Implemented

5.1 Query Expansion

A key challenge in building a search system is bridging the gap between how users phrase their queries and the underlying information need. For example, in a coding context, a user looking for "multithreading in Java" might phrase it as "parallel execution in Java" or "concurrent programming in Java". To bridge this gap, we implemented semantic query expansion using CodeBERT [2], a pre-trained language model trained on both programming and natural language. We generated 768-dimensional embeddings for each word in our vocabulary and precomputed them to avoid on-the-fly calculations. If a user searches for a term not in our vocabulary, it is excluded from query expansion to maintain efficiency, though this is rare. When a query is submitted, the system retrieves embeddings for each term and uses cosine similarity to find semantically related words. For example, a query containing "multithreading" might be expanded to include "parallel".

An important consideration is that while query expansion enhances recall, it also increases computational overhead, as BM25 must process a greater number of terms per query. This can lead to performance degradation, especially for longer queries. To balance recall and efficiency, we imposed a limit of 10 additional terms per query, with 1, 2, or 3 semantically relevant tokens being randomly selected per query term. This restriction not only reduces the computational burden but also prevents excessive expansion, which could introduce irrelevant terms and degrade search precision, ensuring that documents remain contextually aligned with user's original intent.

Furthermore, based on our dataset and hypothesized user behavior, we thought that many queries would include programming languages (e.g., "how to do X in Java"). To enhance retrieval accuracy, we introduced a boosting mechanism for key domain-specific terms, such as programming languages and technical concepts (e.g., "hash", "python"). This adjustment increases the ranking weight of documents explicitly mentioning these terms, making them more likely to appear at the top. Qualitative analysis confirmed that this approach significantly improved search relevance.

5.2 Spellchecker

To handle user typos, we integrated TextBlob's spell checker [3] into our query pipeline, ensuring the system interprets intended queries correctly. This approach was chosen for its speed, though more specialized alternatives exist. For example if given the token "arral" the spellchecker will correct it to "array".

5.3 Tags Grouping

As part of the Stack Overflow dataset, the posts already included predefined tags. However, our current collection contained approximately 40,000 unique tags, making it highly inefficient for users to navigate and filter through, and also computationally expensive from the search engine side. To improve efficiency

and usability, we decided to group the tags into more meaningful and structured categories. This was achieved using hierarchical agglomerative clustering after encoding all unique tags with CodeBERT [2] embeddings. Through evaluation, we found that five clusters provided a good balance between silhouette scores and complexity. Since it is highly impractical to read through all the tags, in order to generate descriptive names for these clusters, we leveraged the TinyLlama model [7] for efficient large language model-based naming categorization. The final tag groupings given were: Programming & Development Fundamentals, Software Engineering & System Design , Advanced Computing & Algorithms, Technologies & Frameworks and Other. This categorization significantly improved the usability and efficiency of tag-based filtering in our dataset. Important to note is that filtering is performed post-document retrieval, as maintaining a separate index for each tag category would be inefficient. Instead, documents are first retrieved based on query relevance, and filtering is applied afterward to refine results based on tag groupings.

5.4 Learn to Rank

Evaluating the relevance of retrieved results was challenging due to the large volume of documents and their lack of ranked importance labels. To address this, we reranked the documents retrieved using Ranked of Advanced Retrieval, based on two approaches: metadata reranking and language model (LM) reranking .After obtaining the initial document list, we assigned a weighted combination of both methods, with LM reranking receiving a higher weight (0.6) and metadata reranking a lower weight (0.4). This bias toward LM reranking was chosen because language models better capture semantic relationships and contextual meaning, improving ranking quality for ambiguous or complex queries.

5.4.1 Metadata Reranking

After carefully analyzing our dataset, we thought that the metadata attributes could provide meaningful insights for re-ranking the retrieved documents. We hypothesized that documents with higher views and comment counts, greater upvote scores, more recent creation dates (to avoid for example displaying old documentation for certain functions), and authors with higher reputation scores would be more valuable to users. Through qualitative hyperparameter tuning, we derived the following re-ranking formula, which assigns weighted importance to each metadata attribute, after of course normalizing them to ensure fairness.

$$\text{ranking\_score} = (\text{score} \times 1.5) + (\text{viewcount} \times 1.2) + (\text{reputation\_user} \times 1.5) + (\text{creation\_date} \times 1.5)$$

5.4.2 LM Reranking

An additional L2R feature we implemented was language model (LM)–based reranking, a technique that has recently emerged as one of the most effective methods for improving search result ranking. For this, we again leveraged a sentence-transformers, namely the all-MiniLM-L6-v2, to precompute dense document embeddings for each document in our corpus. The embeddings are being stored using FAISS for efficient similarity search and fast retrieval. When a user submits a query, it is automatically encoded, in parallel with the initial retrieval process to ensure computational efficiency. The reranking module then computes the cosine similarity between the query embedding and the embeddings of the retrieved documents, allowing the reordering of results based on semantic relevance. This method proved to be highly effective in surfacing more semantically relevant documents, allowing for better ranking of results even when some of the exact keyword matches were absent. By capturing the contextual meaning of queries and documents, LM-based reranking significantly improved retrieval quality beyond traditional BM25-based and Advanced search ranking.

6 Evaluation

To evaluate the performance of our system, we focused on two key metrics: the speed of returning results and their accuracy. For the ranked retrieval, since our data was not labeled in terms of relevance or ranked relevance, we had to adopt an unsupervised evaluation approach. For this, we initially curated a smaller index containing 1 million tokens and created a set of queries covering various coding-related questions, along with other challenging cases that users might attempt (e.g., misspelled words, excessively long texts). A detailed list of all queries and metrics can be found in Appendix 2. For the unsupervised evaluation, we computed two metrics: firstly the *clarity score* that measures the difference in term

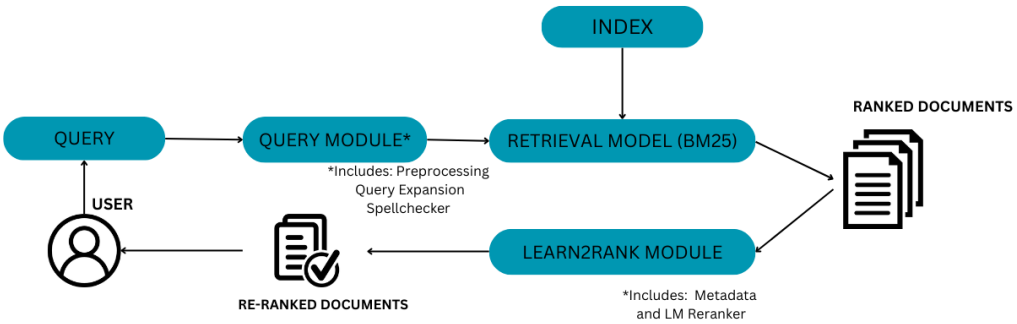

Figure 1: Ranked Retrieval Process

distribution between the retrieved documents and the rest of the corpus and secondly the *Jaccard Score* that measures the similarity among the retrieved documents. These metrics help assess the relevance of the retrieved results, as ideally they would be very different from the rest of the corpus and also very similar among them. Our results showed positive Clarity Scores, confirming that retrieved documents were distinct from the corpus. The average Jaccard Score was lower than expected but still reasonable, given the short queries. As expected, longer queries yielded a bit higher Jaccard Scores. In terms of performance, queries averaged 0.5 seconds when run locally, demonstrating efficient execution.

For the Advanced Search Module evaluation, key factors included accuracy (ensuring correct term matching in AND queries for example) and retrieval time, which averaged 0.7 seconds, though complex queries took longer. Users should expect increased retrieval times for highly detailed searches.

Similar tests were conducted after deployment on the cloud, where the average retrieval speed was 1.4 seconds per document for both ranked and advanced search. This performance is promising, considering the index size. Sometimes, though the initial query will take slightly longer amount of time due to initialization.

7 Front-end Implementation

Our frontend is built with React, providing a simple, responsive UI for both desktop and mobile users. Each UI element—search bars, result lists—is a reusable React component, ensuring maintainability and scalability. For development and production, we use Vite, which offers fast builds and hot module reloading, enabling real-time updates while forwarding API requests locally. Tailwind CSS ensures a consistent, utility-first design, while shadcn/ui provides pre-built components like buttons and inputs. On the data side, RTK Query manages API calls, caching, and error handling, ensuring efficient state management. React Router enables seamless navigation between pages without full reloads. We have designed two pages: the Home Page and the Results Page. The search bar includes buttons for both search types ensuring that the appropriate endpoint is being triggered, while a scrollable list displays posts returned by the backed along with their metadata and allows users to navigate through multiple pages using pagination controls. By separating standard and advanced search at the front end, we ensure each query type gets sent to the appropriate endpoint, while keeping the interface straightforward. The left panel also allows users to filter results by category as described in Section 5.3 and additionally, a button allows users to sort results by date, ensuring they can easily find the most recent posts. The current design can be found in .

8 Back-end

The backend is built using Flask [4], acting as the interface between the database and the frontend, with the API handling all search functionalities. Flask was chosen because it is lightweight, flexible, and efficient for building scalable web applications and APIs. The search system operates in two modes: Basic Search, which uses BM25 to retrieve relevant documents, and Advanced Search, which supports Boolean search, proximity search, and phrase search for more refined results. Before retrieval, the query undergoes preprocessing and query expansion, incorporating similar words to improve recall. The search function first parses the query to extract terms, then checks the cache for existing results—returning

them instantly if found or otherwise retrieving documents based on the search. The retrieved results are then passed to post-search processing, where they are formatted, re-ranked based on metadata, and further refined using a Language Model (LM)-based reranking to ensure the most relevant documents appear at the top. If filters are applied, the results undergo additional processing, where documents can be sorted by date in descending order or filtered by category, displaying only those matching the five predefined tags. The final set of results is then returned to the user via a POST request, ensuring efficiency and relevance in document retrieval.

9 Integration

Our React front end communicates with the backend through standard HTTP requests, sending user-generated queries to an API endpoint and receiving JSON data in return. During local development, we used a proxy setting in the React package.json so that all requests to the front end would automatically would forward to the Flask server url. This made it possible to test both the front end and backend together on a local machine without worrying about cross-origin issues. Once the React application was tested locally, we replaced the proxy with an environment variable (VITE_API_URL) pointing to the live server's URL, rebuilt the front end, and provided those for production deployment.

10 Cloud Implementation

Our search engine was initially deployed directly on a bare-metal system with minimal storage to test our MVP, allowing for quick validation of core functionality. After proving the concept, we transitioned to Docker containerization for the final product, separating frontend and backend into distinct containers to improve isolation and portability. These containerized components were then pushed to the cloud as and when required.

Our initial cloud configuration utilized a general-purpose compute standard E2 machine (4 vCPUs, 16GB RAM, 100GB persistent disk), but performance testing revealed I/O operations as the primary bottleneck despite our indexing optimizations. We strategically upgraded to a compute-intensive C2D machine (8 vCPUs, 32GB RAM, 150GB persistent SSD), which dramatically reduced search response times by more than 50%. The transition to SSD storage proved particularly impactful, as the reduced latency for random access operations significantly accelerated query processing and result retrieval. Due to credit limitations our search engine will likely not respond well with concurrent access.

11 Limitations and Future Work

Despite time and resource constraints, we believe we have developed a scalable, efficient, and innovative search engine. However, with the given resources the current index and dataset is relatively small and cannot answer every programming question comprehensively, but can serve as a benchmark system implementation for expansion. There are also several areas where further improvements can be made to enhance both performance and usability for our current implementation. One key area for enhancement is advanced preprocessing for code recognition. We originally attempted to build a system that could accurately classify and apply separate preprocessing pipelines for code and non-code content, but we found this to be significantly more complex than anticipated due to inconsistencies in formatting and mixed content within posts. Future work will focus on refining syntax-aware tokenization, structure-based detection, and contextual embeddings to improve precision in handling different types of content. Secondly, to ensure scalability, we initially attempted to implement auto-scaling with an Application Load Balancer (ALB), which would distribute incoming HTTP requests across multiple virtual machines (VMs). The ALB directs traffic to available instances or launches new ones as needed. However, due to limited credits, we were unable to fully deploy this setup. Finally, incorporating user feedback into search ranking could significantly improve the relevance of retrieved results. By leveraging user interaction data, such as click-through rates and dwell time, the system could dynamically adjust rankings based on real-world usage patterns. However, due to the lack of sufficient user data and the need for long-term behavioral analysis, we were unable to implement this feature at this stage. These improvements would enhance the accuracy, usability, and adaptability of our system, paving the way for a more intelligent and user-centric search experience.

Individual Contributions

s2704516 My role in this project centered on developing and evaluating the retrieval system. I began by analyzing the index and proposing additional preprocessing steps, then experimented with various retrieval models. I also optimized query expansion using precomputed embeddings and integrated a spellchecker for improved accuracy. A significant part of my work involved evaluating system performance through metrics like response time and result relevance. To ensure robustness, I generated queries covering all search types, including edge cases that could potentially crash the system, and devised solutions to mitigate those issues.

s2051283 Throughout the project, I was primarily responsible for the collection and processing of data for the index, the development of the index itself and integrating the search funcationality into the index to ensure maximum efficiency. Firstly, I collected the StackOverflow dataset and wrote scripts required to populate our backing PostgreSQL database from the raw XML files. Additionally, I wrote scripts used to generate subset databases that could be used for testing and running our system on local machines. Then I worked on the data preprocessing pipeline to parse, tokenize and normalize the textual information into tokens. I worked on building the custom index and document storage system using encoded binary files. This involved writing the programs used to generate the index and the scripts used to query the index. At later stages, I worked on integrating the retrieval functions, such as BM25, boolean search, phrase search and proximity search, directly into the index for maximum efficiency using optimizations such as Numba JIT compilation. I also generated the document-level embeddings for LM reranking and implementing the FAISS index for efficient search and retrieval. Finally, I worked closely with my team-mates on the back-end to integrate the reranking features into our search procedure.

s2716530 My primary role in this project was front-end development, where I focused on creating an interface that was both intuitive and visually appealing. I worked closely with the backend team to ensure a seamless integration of all components. On the evaluation side, I helped by researching various metrics, such as clarity score, to better evaluate our system's performance. Additionally, I contributed to the tag grouping process by using a pre-trained model to generate meaningful tag representations and then refining them to group similar tags together (to improve our filter system, making it easier to organize and display related content).

s2585198 My primary role in the project was to implement key components of the retrieval system and back end. I focused on search functionality and integrating a language model for re-ranking documents. I researched and implemented evaluation metrics to assess the effectiveness of re-ranking, ensuring that the ranking adjustments improved result relevance. For the back end, I first developed the basic search functionality and then implemented filtering mechanisms to allow document filtering by date and tags. Additionally, I designed and implemented a mapping system to align user-selected tags with those generated by the language model. My work primarily involved extending the retrieval model's capabilities, implementing back end functionalities, and collaborating with the front-end team to ensure seamless integration.

s2436122 I played a key role in integrating the frontend and backend with the cloud, ensuring seamless connectivity and deployment. I implemented Docker on both the frontend and backend individually, which made the entire deployment process more efficient and scalable. This approach also allowed me to easily experiment with the deployed application and optimize the choice of virtual machine and disk combinations. In addition, I collaborated with s2051283 to quickly develop a minimum viable product (MVP) index, enabling others to start working on the retrieval models without delay. Moreover, a significant part of my work was research and contributed ideas for better data preprocessing like the removal of code-related stopwords and helped in incorporating such ideas to our indexor.

References

References

[1] Yaohou Fan, Chetan Arora, and Christoph Treude. "Stop words for processing software engineering documents: Do they matter?" In: *2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE)*. IEEE. 2023, pp. 40–47.

[2] Zhangyin Feng et al. "Codebert: A pre-trained model for programming and natural languages". In: *arXiv preprint arXiv:2002.08155* (2020).

[3] Steven Loria. *TextBlob: Simplified Text Processing*. 2023. URL: `https://textblob.readthedocs.io/`.

[4] Pallets Team. *Flask Documentation*. Accessed: 2024-03-03. 2024. URL: `https://flask.palletsprojects.com/`.

[5] M. F. Porter. "An Algorithm for Suffix Stripping". In: *Program: Electronic Library and Information Systems* 14.3 (1980), pp. 130–137. DOI: `10.1108/eb046814`.

[6] Stephen Robertson, Hugo Zaragoza, et al. "The probabilistic relevance framework: BM25 and beyond". In: *Foundations and Trends® in Information Retrieval* 3.4 (2009), pp. 333–389.

[7] TinyLlama Team. *TinyLlama: A 1.1B Llama Model Optimized for Efficiency*. `https://github.com/TinyLlama`. 2024.

[8] Xinyu Zhang et al. "Bag-of-Words Baselines for Semantic Code Search". In: *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*. Ed. by Royi Lachmy et al. Online: Association for Computational Linguistics, Aug. 2021, pp. 88–94. DOI: `10.18653/v1/2021.nlp4prog-1.10`. URL: `https://aclanthology.org/2021.nlp4prog-1.10/`.

Appendix 1

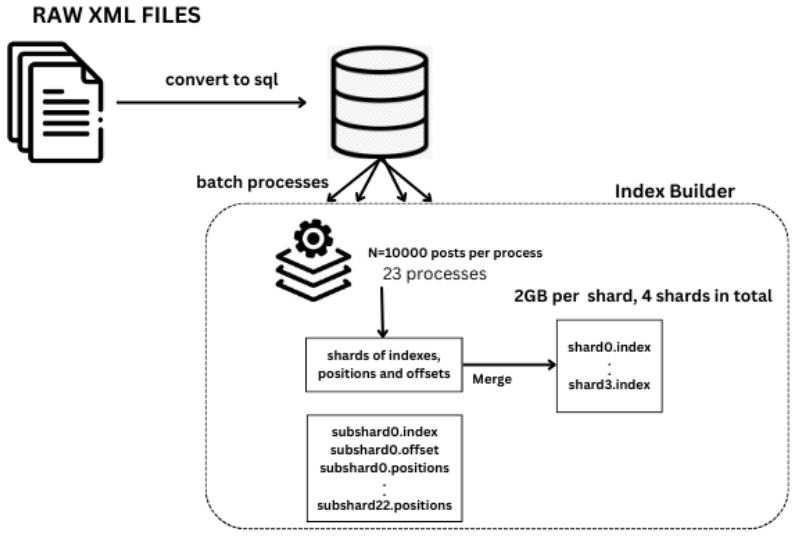

Figure 2: Indexing and processing

Appendix 2: Evaluation of Ranked Retrieval

Query	Clarity Score	Jaccard Index
TypeError: 'NoneType' object is not subscriptable in Python	2.2356	0.0563
NullPointerException handling in Java	2.5564	0.0927
ModuleNotFoundError: No module named 'requests'	2.6116	0.0667
Segmentation fault in C++	2.4958	0.0387
SyntaxError: invalid syntax near 'elif'	2.5540	0.0733
Difference between DFS and BFS algorithms	3.0122	0.0907
How does garbage collection work in Java?	2.6428	0.0361
What is tail recursion, and how does it optimize memory	3.0685	0.0221
What are strong and weak references in Python?	2.3415	0.0418
Explain dynamic programming with an example	2.8331	0.0673
Best way to concatenate strings in Python	2.5238	0.0491
How to optimize SQL queries for large datasets?	2.6725	0.0437
When to use pointers in C++?	2.4395	0.0337
Why is binary search faster than linear search?	2.8197	0.0726
How to improve performance of nested loops in Java?	3.0371	0.0597
How to use Pandas groupby with multiple columns?	2.7609	0.0314
What does std::move do in C++?	2.7783	0.0422
Difference between apply() and map() in Pandas	3.3921	0.0185
How to use React hooks for state management?	3.0654	0.0369
How to make an API request with Axios in JavaScript?	3.0447	0.0751
How to make a website?	3.4507	0.0282
I am craving pasta for dinner	1.9274	0.0866
Coding is very hard, I hate my assignment	2.3859	0.0419
aisjdidhbdd	2.9844	0.0472
What are the differences between breadth-first search (BFS) and depth-first search (DFS) in terms of time complexity, space complexity, and practical applications in graph traversal, and how do these algorithms behave in weighted and unweighted graphs, considering edge cases such as cycles, disconnected components, and large-scale graphs used in AI pathfinding algorithms?	2.2312	0.0982

Table 3: Clarity Score and Jaccard Index for Different Queries

Appendix 3: Front-End and UX

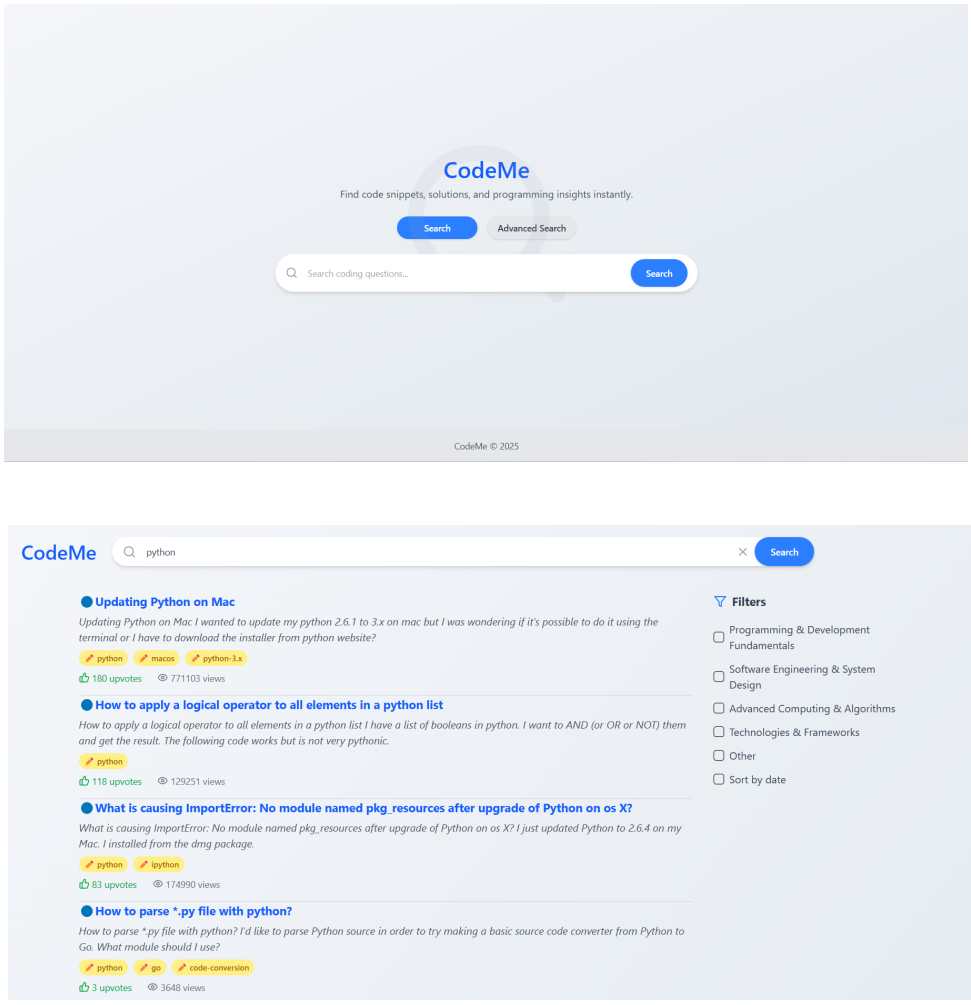

Figure 3: Pages

Appendix 4: Github Link

The Github repository for this project can be found here : https://github.com/annitziak/CodeMe.