UDC 004.93, 004.8

METHOD OF PARALLEL HYBRID SEARCH FOR LARGE-SCALE CODE REPOSITORIES

Boiko V. O. – Assistant of the Department of Software Engineering, Khmelnytskyi National University, Khmelnytskyi, Ukraine.

ABSTRACT

Context. Modern software systems contain extensive and growing codebases, making code retrieval a critical task for software engineers. Traditional code search methods rely on keyword-based matching or structural analysis but often fail to capture the semantic intent of user queries or struggle with unstructured and inconsistently documented code. Recently, semantic vector search and large language models (LLMs) have shown promise in enhancing code understanding. The problem – is designing a scalable, accurate, and hybrid code search method capable of retrieving relevant code snippets based on both textual queries and semantic context, while supporting parallel processing and metadata enrichment.

Objective. The goal of the study is to develop a hybrid method for semantic code search by combining keyword-based filtering and embedding-based retrieval enhanced with LLM-generated summaries and semantic tags. The aim is to improve accuracy and efficiency in locating relevant code elements across large code repositories.

Method. A two-path search method with post-processing is proposed, where textual keyword search and embedding-based semantic search are executed in parallel. Code blocks are preprocessed using GPT-4o model to generate natural-language summaries and semantic tags.

Results. The method has been implemented and validated on a .NET codebase, demonstrating improved precision in retrieving semantically relevant methods. The combination of parallel search paths and LLM-generated metadata enhanced both result quality and responsiveness. Additionally, LLM-post-processing was applied to the top-most relevant results, enabling more precise identification of code lines matching the query within retrieved snippets. Other results can be further refined on-demand.

Conclusions. Experimental findings confirm the operability and practical applicability of the proposed hybrid code search framework. The system's modular architecture supports real-time developer workflows, and its extensibility enables future improvements through active learning and user feedback. Further research may focus on optimizing embedding selection strategies, integrating automatic query rewriting, and scaling across polyglot code environments.

KEYWORDS: hybrid code search, vector search, semantic embeddings, code summarization, LLM-generated metadata, cosine similarity, textual relevance, class and method retrieval, class-based indexing, software engineering.

ABBREVIATIONS

AST is an abstract syntax tree;
LLM is a large language model;
RAG is a retrieval augmented generation;
NLP is a natural language processing;
AI is an artificial intelligence;
GPT is a generative pre-trained transformer;
HNSW is a hierarchical navigable small world;
MRR is an average inverse rank of the first relevant search result.

NOMENCLATURE

B is a set of source code blocks;
b_i is a source code block;
q is a user-supplied natural language query;
S is a set of summaries of code blocks;
E is a set of summary embeddings;
X is input data;
Y' is output data that represents search results;
f is a general representation of a function that performs search based on input parameters;
T_i is a maximum number of input tokens provided to GPT-4o model;
$T_{\max}$ is a maximum tokens GPT-4o model can process;
T_o is a maximum amount of output tokens GPT-4o model can produce;
$C(b_i)$ is a chunking function of a particular code block;
c_{im} is a chunk of a particular code block;
$\ell(c_{ij})$ is a length of a code chunk;
$s_{i(j-m)}$ is a particular summary within a chunk c_{ij};
s_i is a particular summary of a code block;
LLM is the general representation of summarization function;
$\vec{v}_j$ is an embedding vector of the j-th token in the sequence;
k is a number of tokens obtained after tokenizing s_i;
M is an embedding model text-embedding-ada-002;
$\Re^d$ is a space of embeddings;
$\vec{e}_i$ is a raw sentence-level embedding vector;
$\hat{e}_i$ is the L2-normalized embedding vector;
t_i is a token;
$IDF(t_i)$ is an inverse document frequency function;
d is a text-based metadata;
$B(d,q)$ is a term frequency weight;

© Boiko V. O., 2025
DOI 10.15588/1607-3274-2025-3-6

$BM25(d,q)$ is the lexical relevance score between the user's query q and the textual representation d of a code unit;

α is a weighting coefficient (in range [0, 1]) that balances keyword-based vs. semantic-based relevance;

$S(b_i)$ is a hybrid relevance score assigned to code block b_i;

$TopK(\cdot)$ is a function that returns the K highest-scoring elements from the input set;

Y is a set of top-K ranked results;

L is a total loss across all examples, evaluated using a binary cross-entropy loss function;

$GPTR(b_i)$ is a function that refines exact lines of code inside a code block b_i using GPT-4o model;

Y_{top} is a set of the top most relevant search results.

INTRODUCTION

Effective code retrieval is pivotal in modern software development, enabling developers to efficiently locate and reuse existing code snippets. Traditional code search methods have predominantly relied on keyword-based approaches, which, while straightforward, often fail to capture the nuanced semantics of programming languages and the intent behind code implementations. This limitation becomes particularly evident in large-scale codebases, where the sheer volume and complexity of code can hinder accurate retrieval.

Recent advancements in artificial intelligence and natural language processing have introduced semantic search techniques that utilize embeddings to represent code and queries in a continuous vector space. These embeddings facilitate the retrieval of code snippets based on semantic similarity rather than exact keyword matches, thereby enhancing search relevance. However, solely relying on semantic embeddings can overlook the precision offered by traditional keyword searches, especially when specific syntax or identifiers are involved.

The object of study is the process of searching and retrieving relevant code fragments from large-scale and semantically diverse software codebases.

The subject of study is the methods and models for hybrid code search that combine textual keyword matching, semantic embedding-based retrieval, and metadata-enriched indexing.

The known code search approaches and algorithms described by various authors and applied across different domains, including traditional keyword-based [1–2, 4] and structural analysis methods [3], are often limited in capturing the semantic context of code, especially in large and heterogeneous codebases. These methods typically rely on exact textual matches or static syntactic representations, which restrict their effectiveness in scenarios where the user query expresses intent rather than specific code tokens.

However, several recent studies [5–9] have explored semantic search or other not straightforward techniques based on neural models to enhance retrieval relevance. While these approaches demonstrate improvements in understanding code semantics, they generally do not provide a unified method that combines semantic search, keyword filtering, and metadata-based enrichment into a parallel and scalable architecture suitable for practical use in real-world software engineering environments.

The purpose of the work is to develop a method and incorporate it into an efficient and scalable hybrid model for semantic code search, which combines keyword-based filtering, embedding-based retrieval, and LLM-generated metadata. The proposed model is intended to serve as a practical framework for software engineers to search and retrieve relevant code fragments from large-scale codebases based on both natural language queries and structural context.

1 PROBLEM STATEMENT

Suppose we have a set of source code chunks $B = \{b_1, b_2, ..., b_n\}$ from a software codebase, and a user-supplied natural language query q, which represents the search intent. Each chunk $b_i \in B$ contains one or more code blocks.

The task is to develop a method that will perform an accurate hybrid code search method to retrieve a set of relevant source code sections according to the natural language query q.

This can be represented by the following model:

$$X = \{B, S, E, q\},\ S = \{s_i\}, E = \{\hat{e}_i\},\ f : X \to Y', \quad (1)$$

where the function f from input X generates an output Y' which represents the search results.

2 REVIEW OF THE LITERATURE

A typical keyword search is often carried out using algorithms such as Rabin-Karp or Knuth-Morris-Pratt. These algorithms are commonly employed in the development of frameworks designed to detect plagiarism in text documents, as outlined in the study [1]. However, these algorithms are not effective for code search, as they can only detect specific text patterns based on explicitly defined key phrases. As a result, they are not suitable for tasks that demand more advanced search techniques.

Pattern matching search, or Regex search, is a versatile tool that enables flexible string matching by defining complex patterns. It is widely supported across various programming languages, as it is integrated into text processing libraries. In a particular publication [2], RunEx is a code search tool designed for programming instructors to easily identify patterns and mistakes in students' code. It enhances traditional search methods by incorporating runtime values and provides a user-friendly interface for

© Boiko V. O., 2025
DOI 10.15588/1607-3274-2025-3-6
OPEN ACCESS

constructing expressive queries. RunEx outperforms baseline systems in accuracy and introduces a new approach for analyzing student code at scale. However, the industrial program code is a much more complex structure than simple text, so other methods for code search are required.

The methods mentioned earlier are useful only for textual search. However, they cannot be employed for processing difficult code structures. For this, the abstract syntax tree analysis is applicable. For example, paper [3] introduces the similarity detection technique that uses richer structural information while ensuring reasonable execution time. It generates syntax trees from program code, extracts connected *n*-gram structure tokens, and compares them using cosine correlation in the vector space model.

In general, there are a lot of AST-based methods are used and already provided in the most integrated development environments to enhance code detection, correction, syntax highlighting, and search by dependency references. However, understanding the AST is not always needed. For example, publication [4] proposes a model designed to improve code search by combining the advantages of deep learning models like DeepCS with indexing techniques for faster search. This model identifies and removes irrelevant keywords, performs fuzzy search with key query terms using Elasticsearch, and re-ranks results based on sequential token matching.

However, even using Elasticsearch database for indexing, standard search utilities don't support semantic search, which has become even more popular and effective with generative AI development in the last few years. The paper [5] introduces an annotation-based code search engine that addresses information loss by extracting features from code annotations from five perspectives. Unlike current models that treat code annotations as simple natural language, the engine preserves structural information. This approach is much better since it includes deep learning, but it still does not support search queries in a natural language despite its proximity to semantic search.

The paper [6] proposes an efficient and accurate semantic code search framework using a cascaded approach with fast and slow models. The fast model, a transformer encoder, optimizes a scalable index for quick retrieval, while the slow model re-ranks the top K results to improve accuracy. To reduce memory costs, both models are jointly trained with shared parameters. This improves accuracy and efficiency but does not integrate keyword-based filtering or metadata, and the cascaded approach adds complexity.

Another publication [7] introduces RepoRift, a code search approach that leverages RAG-powered agents to improve the accuracy of code retrieval. By enhancing user queries with relevant information from GitHub repositories, the agents provide more contextually aligned and informative inputs to embedding models. The approach also incorporates a multi-stream ensemble technique to further improve retrieval accuracy. It introduces context-awareness but relies on external repositories and augmentation agents, which may not generalize or scale in enterprise environments.

The report [8] presents a novel code retrieval system using the Dense Passage Retrieval technique, which measures functional similarity between code snippets for relevance. By leveraging large-scale pre-trained language models like CodeBERT and Starencoder, the system efficiently retrieves similar code based on natural language descriptions or source code queries. However, it uses pure embedding-based retrieval, without support for text-based filtering, tag-based classification, or enriched metadata.

Another paper [9] proposes a code semantic enrichment approach to improve deep code search by aligning the semantics of code snippets with developers' queries. Recognizing that code represents low-level implementation and queries are high-level, the approach enriches code snippets with descriptions of similar code implementations. Based on a large-scale analysis of a large amount of Java code-description pairs, the method uses syntactic similarity to retrieve similar code for each snippet, enhancing its semantic representation. The model is trained using an attention mechanism to map pairs of enriched code and query into a shared vector space. To further improve representation quality, a multi-perspective co-attention mechanism with Convolutional Neural Networks is applied to capture local correlations. This approach bridges the semantic gap, but it still does not integrate parallel keyword retrieval, nor does it utilize tags or structured metadata for boosting precision.

Chen et al. use both types of search in their retriever and feed results to an LLM. Authors in a conference paper [10] proposed a retrieval-augmented framework for improving code suggestions by combining traditional information retrieval methods and deep learning-based code search with large LLMs. Their system includes a retriever that supports multiple query types (e.g., method headers, natural language), a formulator that constructs prompts using retrieved code, and a generator based on LLMs like ChatGPT. The study demonstrates that incorporating semantically relevant code snippets significantly enhances code generation quality. However, their framework does not explicitly mention summarizing context or performing line-level GPT-based retrieval. Instead, they concatenate retrieved code snippets with the query for the LLM to consume.

While recent advancements in code search emphasize deep learning and semantic retrieval, each of the reviewed approaches addresses only part of the challenge – and none in the research area offers a unified and effective framework for a code-based search.

3 MATERIALS AND METHODS

Modern code search systems are challenged by the semantic gap between a developer's natural language query and the structure and behavior of source code. Traditional keyword-based approaches, while efficient and interpretable, often fail to capture the intent behind a query when relevant code does not share lexical similarity

© Boiko V. O., 2025
DOI 10.15588/1607-3274-2025-3-6
OPEN ACCESS

with the input terms. Conversely, purely embedding-based semantic search can retrieve contextually aligned results but lacks explainability and may yield less precise matches when queries involve specific identifiers or domain terms.

To address these limitations, this work introduces a hybrid code search method that combines the strengths of both paradigms: the precision of keyword-based retrieval and the contextual depth of embedding-based semantic search. This hybrid architecture is further enriched through the integration of LLM-driven summarization and semantic tagging, allowing the system to index not just raw code, but also its abstracted intent and purpose. The method consists of 3 phases: indexing, retrieval, and post-processing. Each is described further.

The indexing phase serves as the preparatory stage in the hybrid code search system, where raw source code files are transformed into structured, queryable data suitable for both keyword-based and embedding-based retrieval. In information retrieval systems, indexing refers to the process of analyzing and organizing source material in a way that enables efficient and accurate search. Within the context of this work, indexing involves parsing source code files, extracting metadata, generating natural language summaries, and creating vector representations of these summaries for storage and subsequent retrieval.

Code summarization is an effective technique for improving code comprehension, maintenance, and reuse by automatically generating natural language descriptions of source code [11], so this method is used to provide descriptive text for code blocks for further search.

Each file in the codebase is processed independently. The content of the file is read and passed through a natural language model to generate a descriptive summary that captures the file's functional role and behavioral semantics. This summary is intended to reflect how a human developer might describe the purpose of the file in natural terms, which enhances its compatibility with natural language queries. For most files, especially those of modest length, the summarization is performed in a single pass using the LLM of the GPT-4o model. The entire file content is provided as input, and a concise, high-level summary is returned.

However, in the case of large files – particularly those exceeding the context window of the language model – a token-based chunking strategy is employed.

Rather than attempting to identify logical or syntactic boundaries within the file, the content is split into fixed-size chunks that fit within the model's token limit, considering space for system instructions and output. The maximum input token can be represented by the following formula (2):

$$T_i = T_{\max} - T_o. \tag{2}$$

Each chunk is summarized individually, and to maintain context coherence across the file, the summary of the preceding chunk is passed along as part of the input when processing the next chunk. The idea of coherent summarization and semantic continuity is described in the paper [12]. This sequential summarization strategy enables the aggregation of a consistent and comprehensive summary, even for files that cannot be processed in a single request. Once all chunks are processed, their summaries are merged and refined to form a single summary representing the entire file. Formula (3) represents a chunking function:

$$\begin{aligned} &C(b_i) = \{c_{i1}, c_{i2}, ..., c_{im}\}, \\ &\ell(c_{ij}) < T_{\max}. \end{aligned} \tag{3}$$

Each chunk is summarized individually using LLM (4):

$$s_{ij} = LLM(c_{ij}, s_{i(j-1)}, s_{i(j-2)}, ..., s_{i(j-m)}). \tag{4}$$

This is a chained summarization process to maintain coherence across chunks. Then, final code block summary is constructed (5):

$$s_i = LLM(s_{i1}, s_{i2}, ..., s_{im}). \tag{5}$$

The sequential summarization continues on the module or folder level and finally leads to the summary of the whole repository.

Following the natural language summary generation, the system computes a semantic embedding of the summary using a pre-trained embedding model. This embedding is a high-dimensional vector that captures the semantic meaning of the text and enables efficient similarity comparisons with query embeddings during retrieval. The selected embedding model for this process is OpenAI's text-embedding-ada-002, which has demonstrated strong performance in encoding semantic representations across diverse domains. Each vector is associated with the corresponding file and stored in a vector search database to facilitate cosine similarity queries during semantic retrieval [13]. However, we should take into account that the embedding model has a token input limit as well, which is 8191 [14]. Thus, there is a need to pass a small portion of a summary to this model. The embedding process consists of the following steps: tokenization, computing dense representation, and normalization.

Tokenization is the process of breaking down text into smaller units called tokens, which can be words, subwords, or characters. This step is foundational in NLP, enabling models to analyze and understand text data. Effective tokenization is crucial for the performance of subsequent NLP tasks. The following formula (6) represents the tokenization process [15]:

$$s_i \to \{t_1, t_2, ..., t_k\},\ k \le T_e. \tag{6}$$

© Boiko V. O., 2025
DOI 10.15588/1607-3274-2025-3-6
OPEN ACCESS

After tokenization, each token is mapped to a dense vector, known as an embedding. These embeddings are continuous vector representations in a high-dimensional space, capturing semantic and syntactic information about the tokens. Dense embeddings allow models to discern relationships between words based on their contextual usage [16]. OpenAI uses the mean pooling across tokens to obtain the final sentence-level embedding. Formula (7) represents of how a sentence or document embedding is generated:

$$\vec{e}_i = M(s_i) = \frac{1}{k}\sum_{j=1}^{k}\vec{v}_j,\ \vec{v}_j \in \Re^d. \quad (7)$$

Normalization adjusts these dense vectors to ensure consistent scaling and distribution, which is vital for the stability and performance of neural networks. Techniques like layer normalization standardize the inputs across features, facilitating faster convergence during training and improving generalization. Formula (8) represents a process of how the normalized vector (directional embedding) is calculated:

$$\hat{e}_i = \frac{\vec{e}_i}{\|\vec{e}_i\|}. \quad (8)$$

Now, the final formula is a concise and normalized representation of how an embedding vector is computed for a text summary (8):

$$\hat{e}_i = M(s_i) = \frac{1}{k}\sum_{j=1}^{k}M(t_j). \quad (9)$$

Instead of asking the GPT-4o model to generate the overall summary, there is a need to make a prompt and point that the model should return summaries of each meaningful block of code in the file and represent it in a JSON format. Fig. 1 illustrates the template of the completion request to generate the summary of the file.

```
[
  {
    "role": "system",
    "content": "You are a code summarization assistant."
  },
  {
    "role": "user",
    "content": "Analyze the source code provided and output a
JSON array. Each item should include: type: either class, method,
or property; name: the name of the code block; tags: keywords of
a particular code block does; start_line and end_line: start_line
and end_line: estimated line numbers  end_line: estimated line
number where the block ends; summary:detailed explanation line-
by-line or by logical segment
  },
  { "role": "assistant", "content": "Provide me with a source code"},
  { "role": "user", "content": "<source_code>"}
]
```

Figure 1 – Prompts to set up the code summarization assistant

The system employs Qdrant as the underlying vector database to support high-performance semantic code search. Qdrant is selected for its efficient handling of high-dimensional vector spaces, making it particularly suitable for storing and querying dense embedding vectors derived from code summaries. Its ability to perform approximate nearest neighbor search using methods like HNSW ensures fast and scalable similarity retrieval across large codebases.

In addition to vector indexing, Qdrant offers real-time filtering and payload-based queries, allowing search results to be refined using additional metadata without post-processing overhead. This functionality is crucial for hybrid search systems that require filtering by attributes such as file names, method names, or code block positions. As a result, Qdrant supports multi-modal retrieval by combining semantic similarity scores with structured filters.

The system stores not only the normalized embeddings of summarized code chunks, but also custom metadata (payload) including the full file path, the name and type of the code block (e.g., class, method), and the corresponding line range within the file. This enables precise result mapping back to the original source files, as well as advanced use cases such as highlighting specific lines or linking results to developer tools [17]. The structure of the Quadrant data record is shown in Fig. 2.

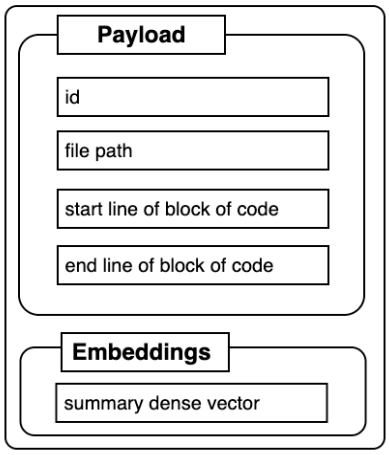

Figure 2 – A data structure for vector database Qdrant

In parallel to vector indexing, the system prepares the indexed data for keyword-based search. Summaries of code blocks, as well as additional metadata, are indexed using Elasticsearch. This database is selected due to its support for full-text search, fuzzy matching, BM25 ranking, and scalable indexing capabilities. These features make it particularly suitable for handling keyword queries where precise or partial text matches are desired [18]. The data structure is mostly the same as for vector database, but instead of embeddings – summaries and tags are saved.

By separating the vector-based search functionality and the text-based search pipeline, the indexing phase ensures that both retrieval modes can be executed inde-

© Boiko V. O., 2025
DOI 10.15588/1607-3274-2025-3-6
OPEN ACCESS

pendently and efficiently. When the code base is changed, the system detects changes and re-generates summaries with their dense vectors only for an updated file. It can happen in the background once per some time range or while a search happens if the system detects that the code base has changed.

After this phase, the indexed codebase contains structured, searchable entries for each file, consisting of its metadata, summaries and their dense vectors, and keyword-indexed content. This structured representation supports fast and accurate retrieval in the subsequent stages of the system. Fig. 3 illustrates the activity diagram of an indexing phase.

The retrieval phase constitutes the core of the hybrid search mechanism and is responsible for identifying relevant source code blocks based on user-supplied natural language queries. This phase integrates two parallel retrieval processes: traditional keyword-based search and semantic vector-based search. Both processes operate over the indexed representation of the codebase generated during the previous phase to maximize the relevance and completeness of the search results.

Upon receiving a natural language query, the system performs keyword-based retrieval by submitting the query to a full-text search engine, such as Elasticsearch. This engine operates over the textual content indexed during the indexing phase, particularly focusing on the code block summaries and the generated semantic tags. Standard ranking techniques, including the BM25 scoring function (10), are employed to identify documents that contain direct lexical overlap with the query terms. This retrieval path provides high precision, especially for queries that include domain-specific keywords, identifiers, or terminology that match the indexed content directly.

$$BM25(d,q)=\sum_{i=1}^{n} IDF(t_i)\cdot B(d,q). \tag{10}$$

In parallel, the system conducts semantic retrieval by encoding the user query into a high-dimensional embedding using the same embedding model employed during indexing. In this case, the model used is OpenAI's text-embedding-ada-002, which produces vector representations that reflect the semantic meaning of the input text. The resulting query embedding is then compared against the stored file embeddings in a vector database using cosine similarity as the distance metric (11):

$$\cos(\hat{e}_q,\hat{e}_i)=\hat{e}_q\cdot\hat{e}_i,\quad \left\|\hat{e}_q\right\|=\left\|\hat{e}_i\right\|=1. \tag{11}$$

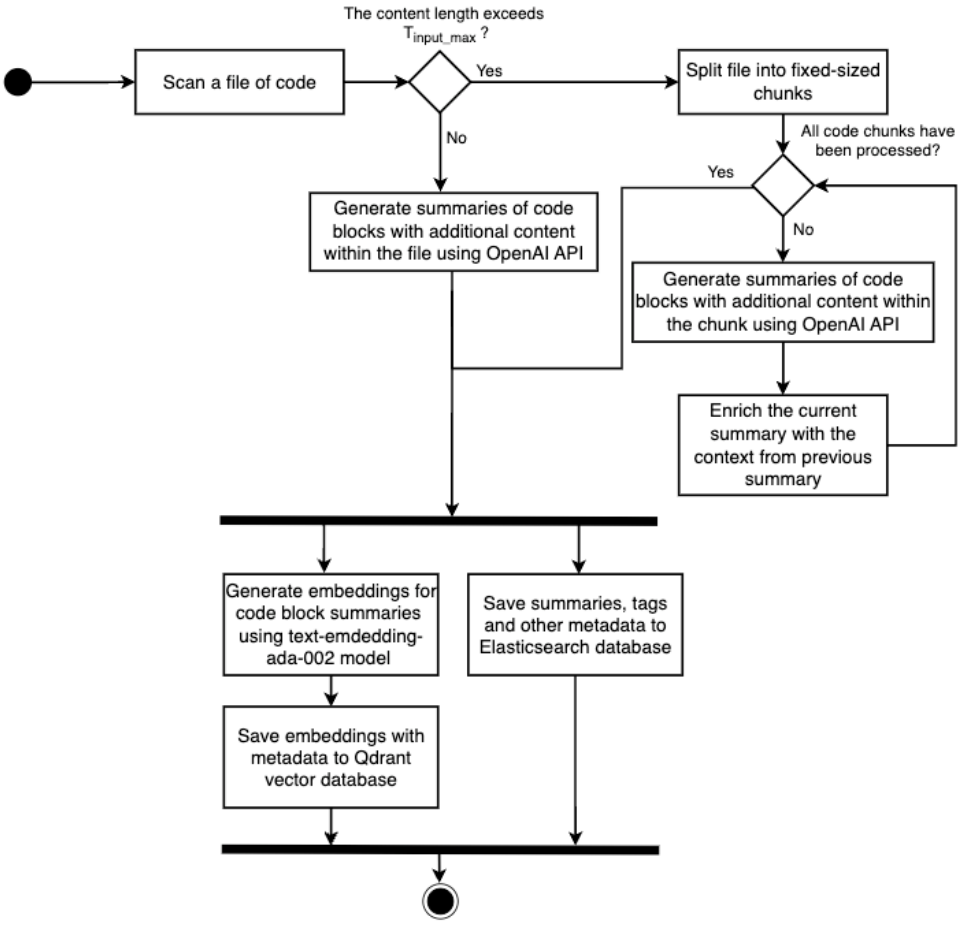

Figure 3 – Activity diagram of indexing phase

© Boiko V. O., 2025
DOI 10.15588/1607-3274-2025-3-6

This retrieval path enables the system to capture conceptual and contextual similarities between the query and the code summaries, even when there is no direct lexical correspondence.

The results from both retrieval paths are processed independently and can be returned as separate ranked lists or integrated into a unified ranking. In cases where both engines yield results for the same file, these results can be merged, with ranking adjusted based on predefined weighting or scoring strategies. Each retrieved file entry includes associated metadata such as the file name, path, summary, and optionally the top-matching tags or score explanations from both retrieval methods (12):

$$\begin{aligned} &S(b_i) = \alpha \cdot BM25(q, d_i) + \\ &+ (1-\alpha) \cdot \cos(\hat{e}_q, \hat{e}_i),\ \alpha \in [0,1], \\ &Y = TopK(\{b_i \in B \mid S(b_i)\}). \end{aligned} \tag{12}$$

The dual retrieval strategy enables the system to respond effectively to a wide range of queries, from those requiring strict keyword matches to those involving abstract or concept-driven search intent. It also supports fallback mechanisms in cases where one of the retrieval paths returns no results or results of low relevance. This phase concludes with the identification of candidate files that are passed to the next stage of the pipeline for more granular analysis and line-level code identification. Fig. 4 illustrates the activity diagram of a retrieval phase.

Following the retrieval of candidate source code blocks through text-based and semantic search mechanisms, the post-processing phase is responsible for narrowing down the search results to the most relevant segments of code at a finer granularity. This stage is particularly important when user queries pertain to specific functional behavior or logic that is confined to specific code lines, rather than the code block as a whole.

To enable this refinement, each retrieved code block is further analyzed using an LLM, which operates on the full content of it, its summary, and the original user query. The objective of the model is to determine which parts of the code are most likely to satisfy the semantic intent of the query. This is achieved by prompting the model with both the query and the block-level context, asking it to identify and return the specific lines or regions of interest within the code. If training or evaluation with ground truth matches, we may define a binary relevance label $y_i \in \{0,1\}$ indicating whether code block f_i is relevant to query q, and predicted score $\hat{y}_i = S(f_i)$. We can evaluate with binary cross-entropy loss:

$$L = -\sum_{i=1}^{n} \left[y_i \log(\hat{y}_i) + (1-y_i) \log(1-\hat{y}_i) \right] \tag{13}$$

The model receives as input the user's natural language query, the code block summary that was previously generated and indexed, and the source code in a line range.

Then, hybrid retrieval with a refinement is shown in formula (14):

$$Y' = \{GPTR(b_i) \mid b_i \in Y\}. \tag{14}$$

Thus, the entire formula expresses that every code block retrieved by the initial hybrid search is further refined using LLM, and the resulting set contains the final, context-aware, and human-readable answers that the system returns to the user. This operation ensures that the output is not just a ranked list of code blocks but a targeted extraction of meaningfully relevant code fragments.

The prompt structure, which is represented in Fig. 5, encourages the model to scan the code in context and locate logic segments that exhibit semantic alignment with the query. In some cases, the model may return an exact set of line numbers that correspond to the requested functionality.

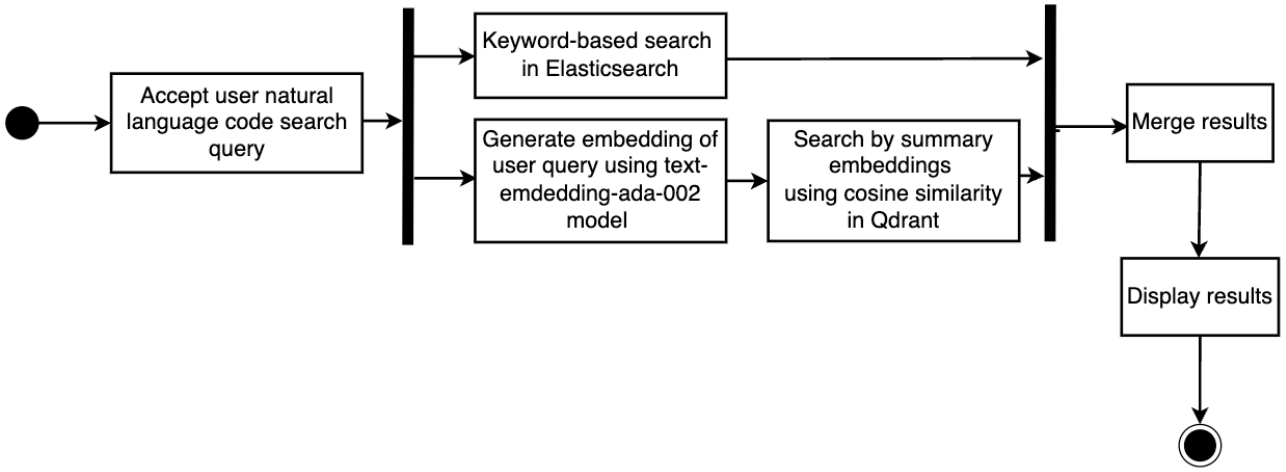

Figure 4 – Activity diagram of retrieval phase

© Boiko V. O., 2025
DOI 10.15588/1607-3274-2025-3-6

```
[
  {
    "role": "system",
    "content": "You are a code analysis assistant."
  },
  {
    "role": "user",
    "content": "Ananlyze a code block to identify which parts of
the code most closely match the following user query based on
the summary and give the line numbers of possible match"
  },
  {
    "role": "assistant",
    "content": "Provide me with a summary and source code"
  },
  {
    "role": "user",
    "content": "Summary: <code_block_summary>;
               Source code: <source_code>
               Line range is: <start_line_number> -
               <end_line_number>"}
]
```

Figure 5 – Example of prompts for code lines retrieving within a post-processing stage

To optimize performance and reduce inference time, only the most relevant results (typically the top five according to the hybrid scoring function) are selected for automatic refinement. The remaining results are excluded from immediate processing and may be refined on demand, based on user interaction. This selective approach ensures scalability while still allowing detailed semantic analysis when needed. Thus, the final formula of hybrid search with refinement of top 5 the most relevant results is presented (15):

$$\begin{aligned} &Y_{top} = TopK(Y,5), \\ &Y' = \{GPTR(b_i) \mid b_i \in Y_{top}\} \cup \\ &\cup \{b_i \mid b_i \in (Y \setminus Y_{top})\}. \end{aligned} \quad (15)$$

The granularity of analysis in this phase is intended to improve the specificity and relevance of search results. While previous stages identify files likely to contain relevant content, this phase identifies and highlights the exact implementation points within those files. The output of this step may be a ranked list of method names, code excerpts, or line ranges, depending on how the model is instructed and the formatting required for downstream presentation. Fig. 6 illustrates the post-processing phase.

The proposed hybrid code retrieval method integrates structured indexing, dual-mode retrieval, and LLM-assisted post-processing to address the challenges of semantic code search in large codebases. Beginning with context-aware summarization and embedding during the indexing phase, the system enables flexible querying through parallel keyword-based and vector-based search mechanisms. The retrieval process balances lexical precision and semantic understanding, while the final line-level refinement phase leverages large language models to isolate the most relevant code fragments based on user intent. Together, these components form a scalable and context-sensitive search pipeline that supports both broad discovery and fine-grained code navigation.

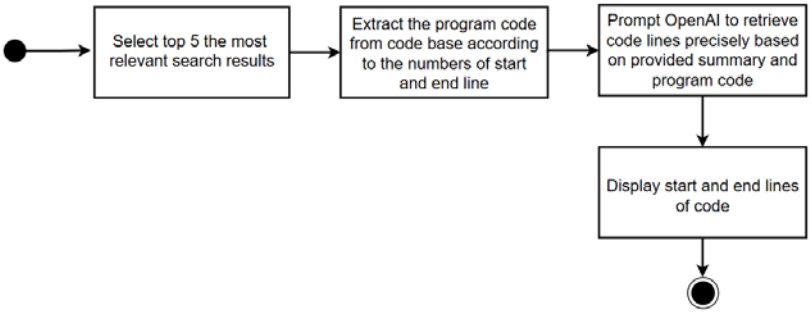

Figure 6 – Post-processing phase

4 EXPERIMENTS

To evaluate the applicability and effectiveness of the proposed hybrid code search method, a set of experiments was conducted using a real-world software codebase. The system was implemented in a .NET-based environment, incorporating components for indexing, vector storage, text search, and OpenAI API integration for both summarization and post-retrieval code refinement.

The primary goal of the experiments was to assess the ability of the system to retrieve semantically relevant code segments based on natural language queries. A test codebase in the domain of service management was selected, containing multiple classes and methods written in C#. This environment enabled the evaluation of the pipeline across different levels of abstraction – from file-level indexing to line-level code extraction.

The indexing phase was implemented as a standalone preprocessing utility that analyzed all .cs files in the selected codebase. Each file was passed through a summarization module based on OpenAI's GPT-4o model. In cases where a file exceeded the input length limitations, a recursive chunking strategy was applied, and the final summary was constructed by aggregating context-enriched chunk-level summaries.

The generated summaries were then embedded using the text-embedding-ada-002 model, and the resulting vectors were stored in a local instance of Qdrant, a high-performance vector database. Simultaneously, the summaries and metadata – including generated semantic tags – were indexed using a locally hosted Elasticsearch instance to support keyword-based retrieval.

© Boiko V. O., 2025
DOI 10.15588/1607-3274-2025-3-6
OPEN ACCESS

Natural language queries were submitted through a simple web interface. Upon query submission, both text-based search and embedding-based retrieval were performed in parallel. Retrieved results were displayed to the user along with cosine similarity scores and matched summaries.

Following initial retrieval, the full source code of selected files was passed to GPT-4o for code analysis. The model was prompted with the original query and the code block summary and instructed to return the most relevant methods or code blocks in structured JSON format, including estimated line numbers and explanatory notes. This allowed for refined pinpointing of logic relevant to the user's intent.

A representative example query "Find places where service orders are filtered by ID" was tested on the indexed codebase. The system successfully retrieved a method named SearchOrders located in the OrderService.cs file. The post-processing phase highlighted the exact lines performing filtering based on the CustomerId field. The result was presented with the matched method, its position in the file, and a step-by-step explanation of the code logic.

5 RESULTS

To evaluate the performance and responsiveness of the implemented hybrid code search pipeline, a series of experiments were conducted using a codebase consisting of 25 C# source files. The average file size was approximately 30–40 KB, and the files varied in structural complexity, encompassing service classes, data repositories, and utility methods. The objective of the evaluation was to confirm the system's ability to index, retrieve, and refine relevant code fragments using the described keyword-based, embedding-based, and LLM-assisted methods.

During the indexing phase, all files were successfully processed without failure. Summarization of each file or chunk (for larger files) was completed using the OpenAI GPT-4o model, followed by embedding generation with the text-embedding-ada-002 model. Vector data was stored in a Qdrant instance running locally, while summaries and tags were indexed using a local Elasticsearch server.

To test retrieval performance, 10 queries were selected, each representing typical developer requests such as "Find where service orders are filtered by ID" or "Show methods that generate auth tokens." For each query, both search paths were executed in parallel. On average, the system returned relevant results within 0.90 to 1.0 seconds, including post-processing by GPT for line-level code matching only for the most relevant results.

The average time for OpenAI API calls during post-retrieval refinement was approximately 0.94 seconds, while the combined time for vector search and keyword search was under 500 milliseconds. These results indicate that the system can operate within practical response times suitable for interactive use. During testing, the standard Tier 1 subscription was used. In high-throughput scenarios, batch processing or queuing would be necessary to avoid exceeding request quotas.

To better understand the effectiveness of the proposed hybrid code search method, a benchmarking comparison (Table 1) was conducted against two baseline approaches:

– baseline A – traditional keyword-based retrieval using Elasticsearch with BM25 scoring;

– baseline B – embedding-only semantic search using text-embedding-ada-002 vectors in Qdrant without keyword filtering or refinement;

– proposed hybrid approach – parallel execution of both search strategies, followed by GPT-based post-processing for line-level code matching.

The evaluation dataset consisted of 10 developer-like queries formulated in natural language. Relevance was manually assessed by analyzing whether the retrieved code matched the intended logic or functionality described in the query. The evaluation was performed using the following metrics:

– top-1 precision – if the top result was relevant;

– top-5 recall – the proportion of relevant items among the top 5;

– MRR – average inverse rank of the first relevant result [19];

– average response time – total time to produce the final result, including post-processing.

The hybrid approach significantly outperforms both baselines in terms of retrieval quality, particularly for queries with complex or abstract semantics. The use of LLM-based summarization and refinement contributes to higher Top-1 precision and MRR scores, demonstrating the system's ability to retrieve not only relevant files but also the most accurate code segments within them.

While the hybrid method with refinement introduces additional latency due to post-processing, the average response time of approximately 0.94 seconds remains within acceptable bounds for interactive search tasks and it has been optimizing by handling a refinement of the most relevant result and the rest of results are intended to be processed on demand by user interaction. In comparison, embedding-only search is faster but occasionally less precise due to the absence of textual disambiguation and LLM refinement.

The results confirm the system's ability to produce accurate, context-sensitive, and developer-usable code search outputs across a heterogeneous codebase while maintaining acceptable execution times for all processing stages.

Table 1 – Benchmarking comparison

Method	Top-1 precision	Top-5 recall	MRR	Average time (s)
Baseline A (BM25)	0.58	0.69	0.61	0.40
Baseline B (Embeddings)	0.75	0.88	0.80	0.28
Hybrid	0.91	1.00	0.92	0.94

© Boiko V. O., 2025
DOI 10.15588/1607-3274-2025-3-6
OPEN ACCESS

6 DISCUSSION

The proposed hybrid code retrieval method demonstrates practical applicability for source code search across large-scale and heterogeneous codebases. In comparison to prior research in code search and semantic retrieval [4–10], this approach eliminates the need for training custom models by leveraging general-purpose, pre-trained language models. Unlike domain-specific models, which often require fine-tuning on large, curated datasets, the use of GPT-based APIs allows the system to remain flexible and adaptable to a broad range of programming styles and query types without retraining.

While custom-trained models may exhibit strong performance in narrow domains, they often suffer from limited generalization when applied to unfamiliar codebases or other programming languages. The hybrid method presented in this work benefits from the broad domain coverage and general language understanding embedded in OpenAI's GPT models, enabling it to interpret developer queries more naturally and perform code summarization in a context-aware manner.

An advantage of the system lies in its layered architecture, which combines the precision of keyword-based retrieval with the semantic depth of vector-based embedding search. The addition of GPT-based post-processing further enhances the system's ability to localize relevant code fragments within files, aligning search outputs with user intent. The results of the experiments confirm that the hybrid method achieves higher precision and recall compared to standalone search methods, especially for abstract or semantically rich queries.

However, the system also inherits limitations from its reliance on external APIs. The OpenAI GPT models, while highly capable, are constrained by token-based limits and subscription-dependent rate quotas. These constraints may impact the scalability of the method in high-throughput or real-time search scenarios. To mitigate this, the system includes a chunking and recursive summarization strategy to handle large files, ensuring full coverage of the codebase even when input sizes exceed model capacity.

Moreover, while the current pipeline performs well in general software engineering contexts, future improvements may involve domain-adaptive summarization or the incorporation of static analysis techniques (e.g., AST matching or control flow analysis) to further enrich search quality. Another promising direction involves integrating the hybrid method into development environments allowing for contextual, in-line code discovery and reuse during software maintenance or refactoring tasks.

CONCLUSIONS

The hybrid code search method was developed and implemented as a solution that combines keyword-based retrieval, vector-based semantic search, and LLM-driven summarization and refinement. The system was tested on real-world codebases and evaluated using standard search effectiveness metrics to validate its practical applicability.

The scientific novelty of the obtained results lies in the integration of multiple retrieval modalities into a unified pipeline, enhanced by recursive summarization and line-level reasoning via GPT model. The proposed method introduces a structured, context-aware approach to code retrieval, enabling semantic alignment between developer queries and relevant code segments across a large-scale codebase.

The practical significance of the obtained results is reflected in the method's ability to automate code retrieval tasks without relying on rigid structures or manually crafted rules. This flexibility allows developers to search using natural language and receive highly relevant results at both the file and method levels. The modular architecture facilitates integration into software engineering workflows, development environments, and documentation systems.

Prospects for further research include exploring optimization strategies to reduce dependency on API rate limits and improve runtime performance in large-scale deployments. Additional directions may involve the use of static code analysis techniques, domain-adaptive summarization models, and the expansion of hybrid retrieval methods into other software engineering domains, including automated documentation, test generation, and intelligent code navigation tools.

REFERENCES

1. Kumar Vivek, Chinmay Bhatt, Varsha Namdeo A framework for document plagiarism detection using Rabin Karp method, *International Journal of Innovative Research in Technology and Management,* 2021, Vol. 5, pp. 17–30.
2. Zhang Ashley Ge, Chen Yan, Oney Steve RunEx: Augmenting Regular-Expression Code Search with Runtime Values, *Proceedings of the 2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2023, pp. 145–155. DOI: 10.1109/VL-HCC57772.2023.00024
3. Karnalim Oscar, Simon Syntax Trees and Information Retrieval to Improve Code Similarity Detection, *Proceedings of the Twenty-Second Australasian Computing Education Conference (ACE 2020)*, 2020, pp. 48–55. DOI: 10.1145/3373165.3373171
4. Liu Chao, Xia Xin, Lo David, Liu Zhiwei, Hassan Ahmed E., Li Shanping CodeMatcher: Searching Code Based on Sequential Semantics of Important Query Words. Ithaca, arXiv, 2020, 36 p. (Preprint / arXiv; 2005.14373). DOI: 10.1145/3465403
5. Kong Xianglong, Chen Hongyu, Yu Ming, Zhang Lixiang Boosting Code Search with Structural Code Annotation, *Electronics*, 2022, Vol. 11, No. 19, P. 3053. DOI: 10.3390/electronics11193053
6. Gotmare Khilesh Deepak, Li Junnan, Joty Shafiq, Hoi Steven C. H. Cascaded Fast and Slow Models for Efficient Semantic Code Search. Ithaca: arXiv, 2021, 12 p. (Preprint / arXiv; 2110.07811). DOI: 10.48550/arXiv.2110.07811
7. Jain Sarthak, Dora Aditya, Sam Ka Seng, Singh Prabhat LLM Agents Improve Semantic Code Search. Ithaca, arXiv, 2024, 12 p. (Preprint / arXiv; 2408.11058). DOI: 10.48550/arXiv.2408.11058
8. Khan M. A. M. Development of a code search engine using natural language processing technique: Graduate thesis.

© Boiko V. O., 2025
DOI 10.15588/1607-3274-2025-3-6

IUT, Department of Computer Science and Engineering, 2023, 65 p.
9. Deng Zhongyang, Xu Ling, Liu Chao, Huangfu Luwen, Yan Meng Code semantic enrichment for deep code search, *Journal of Systems and Software*, 2024, Vol. 207, P. 111856. DOI: 10.1016/j.jss.2023.111856
10. Chen Junkai, Hu Xing, Li Zhenhao, Gao Cuiyun, Xia Xin, Lo David Code Search Is All You Need? Improving Code Suggestions with Code Search, *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024), Lisbon, Portugal, April 14–20, 2024,* 2024, Article No. 73, pp. 1–13. DOI: 10.1145/3597503.3639085
11. Nate Suraj, Patil Om, Medar Shreenidhi, Deshmukh Jyoti A Survey on Transformer-based Models in Code Summarization, *International Research Journal on Advanced Engineering Hub (IRJAEH),* 2025, Vol. 3, pp. 740–745. DOI: 10.47392/IRJAEH.2025.0103
12. Parmar Mihir, Deilamsalehy Hanieh, Dernoncourt Franck, Yoon Seunghyun, Rossi Ryan A., Bui Trung Towards Enhancing Coherence in Extractive Summarization: Dataset and Experiments with LLMs, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 19810–19820. DOI: 10.18653/v1/2024.emnlp-main.1106
13. Korade Nilesh Bhikaji, Salunke Mahendra B., Bhosle Amol, Kumbharkar Prashant Babarao, Asalkar Gayatri, Khedkar Rutuja G. Strengthening Sentence Similarity Identification Through OpenAI Embeddings and Deep Learning, *International Journal of Advanced Computer Science and Applications,* 2024, Vol. 15, No. 4, pp. 821–829. DOI: 10.14569/IJACSA.2024.0150485
14. OpenAI. New and improved embedding model [Electronic resource], *OpenAI,* Mode of access: https://openai.com/index/new-and-improved-embedding-model (date of access: 09.04.2025). – Title from screen.
15. Patil Rajvardhan, Boit Sorio, Gudivada Venkat N., Nandigam Jagadeesh A Survey of Text Representation and Embedding Techniques in NLP, *IEEE Access,* 2023, Vol. 11, pp. 36120–36146. DOI: 10.1109/ACCESS.2023.3266377
16. Jiang Xue, Wang Weiren, Tian Shaohan, Wang Hao, Lookman Turab, Su Yanjing Applications of natural language processing and large language models in materials discovery, *npj Computational Materials,* 2025, Vol. 11. DOI: 10.1038/s41524-025-01554-0
17. Qdrant. Qdrant Vector Database: High-performance vector similarity search [Electronic resource], *Qdrant Documentation*. Mode of access: https://qdrant.tech/qdrant-vector-database (date of access: 09.04.2025). – Title from screen.
18. Elastic. Elasticsearch: The Official Distributed Search & Analytics Engine [Electronic resource], *Elastic.* Mode of access: https://www.elastic.co/elasticsearch (date of access: 09.04.2025). Title from screen.
19. Hoyt Charles Tapley, Berrendorf Max, Galkin Mikhail, Tresp Volker, Gyori Benjamin M. A Unified Framework for Rank-based Evaluation Metrics for Link Prediction in Knowledge Graphs. Ithaca: arXiv, 2022, 18 p. (Preprint / arXiv; 2203.07544). DOI: 10.48550/arXiv.2203.07544

Received 12.04.2025.
Accepted 21.06.2025.

УДК 004.93, 004.8

МЕТОД ПАРАЛЕЛЬНОГО ГІБРИДНОГО ПОШУКУ ДЛЯ ВЕЛИКИХ РЕПОЗИТОРІЇВ КОДУ

Бойко В. О. – асистент кафедри інженерії програмного забезпечення Хмельницького національного університету, Хмельницький, Україна.

АНОТАЦІЯ

Актуальність. Сучасні програмні системи містять великі кодові бази, що робить пошук коду критично важливим завданням для розробників програмного забезпечення. Традиційні методи пошуку коду спираються на співставлення за ключовими словами або структурний аналіз, але часто не здатні відобразити семантичний зміст запитів користувачів або мають проблеми з неструктурованим та непослідовно задокументованим кодом. Останнім часом семантичний векторний пошук і великі мовні моделі (LLM) показали перспективи в покращенні розуміння коду. Проблема полягає в розробці масштабованого, точного та гібридного методу пошуку коду, здатного знаходити відповідні фрагменти коду на основі як текстових запитів, так і семантичного контексту, при цьому підтримуючи паралельну обробку та пошуку на основі метаданих.

Мета роботи – розробка гібридного методу семантичного пошуку коду шляхом комбінування фільтрації за ключовими словами та пошуку на основі вбудованих представлень, доповненого сумаризацією та семантичними тегами, згенерованими за допомогою LLM для підвищення точності та ефективності пошуку відповідних елементів коду у великих кодових репозиторіях.

Метод. Для досягнення мети дослідження розроблено метод пошуку з двома шляхами з пост-обробкою, де пошук за текстовими ключовими словами та пошук на основі вбудовуваних семантичних представлень виконуються паралельно. Блоки коду попередньо обробляються за допомогою GPT-4o моделі для генерування сумаризації та семантичних тегів.

Результати. Метод реалізовано та перевірено на кодовій базі .NET, що продемонструвало покращену точність при знаходженні семантично релевантних методів. Комбінація паралельних шляхів пошуку та метаданих, згенерованих LLM, покращила якість результатів. Для підвищення релевантності було застосовано LLM-постобробку яка виконується над найбільш релевантними результатами, що дозволяє точніше локалізувати потрібні рядки коду в межах знайдених фрагментів. Інші результати можуть бути оброблені на вимогу користувача.

Висновки. Експериментальні результати підтвердили працездатність та практичну застосовність запропонованої гібридної системи пошуку коду. Модульна архітектура системи підтримує робочі процеси розробників в реальному часі, а її розширюваність дозволяє впроваджувати майбутні покращення через активне навчання та зворотний зв'язок від користувачів. Подальші дослідження можуть бути спрямовані на оптимізацію стратегій вибору вбудованих представлень, інтеграцію автоматичного переформатування запитів та масштабування у багатомовних кодових середовищах.

КЛЮЧОВІ СЛОВА: гібридний пошук коду, векторний пошук, семантичні вбудовування, сумаризація коду, метадані, згенеровані LLM, косинусна схожість, текстова релевантність, пошук класів та методів, індексування на основі класів, інженерія програмного забезпечення.

© Boiko V. O., 2025
DOI 10.15588/1607-3274-2025-3-6

ЛІТЕРАТУРА

1. Kumar V. A framework for document plagiarism detection using Rabin Karp method / Vivek Kumar, Bhatt Chinmay, Namdeo Varsha // International Journal of Innovative Research in Technology and Management. – 2021. – Vol. 5. – P. 17–30.
2. Zhang A. G. RunEx: Augmenting Regular-Expression Code Search with Runtime Values / Ashley Ge Zhang, Yan Chen, Steve Oney // Proceedings of the 2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). – 2023. – P. 145–155. DOI: 10.1109/VL-HCC57772.2023.00024
3. Karnalim O. Syntax Trees and Information Retrieval to Improve Code Similarity Detection / Oscar Karnalim, Simon // Proceedings of the Twenty-Second Australasian Computing Education Conference (ACE 2020). – 2020. – P. 48–55. DOI: 10.1145/3373165.3373171
4. CodeMatcher: Searching Code Based on Sequential Semantics of Important Query Words / [Chao Liu, Xin Xia, David Lo et al.]. – Ithaca : arXiv, 2020. – 36 p. – (Preprint / arXiv; 2005.14373). DOI: 10.1145/3465403
5. Boosting Code Search with Structural Code Annotation / [Xianglong Kong, Hongyu Chen, Ming Yu, Lixiang Zhang] // Electronics. – 2022. – Vol. 11, No. 19. – P. 3053. DOI: 10.3390/electronics11193053
6. Cascaded Fast and Slow Models for Efficient Semantic Code Search / [Khilesh Deepak Gotmare, Junnan Li, Shafiq Joty, Steven C. H. Hoi]. – Ithaca: arXiv, 2021. – 12 p. – (Preprint / arXiv; 2110.07811). DOI: 10.48550/arXiv.2110.07811
7. LLM Agents Improve Semantic Code Search / [Sarthak Jain, Aditya Dora, Ka Seng Sam, Prabhat Singh]. – Ithaca: arXiv, 2024. – 12 p. – (Preprint / arXiv; 2408.11058). DOI: 10.48550/arXiv.2408.11058
8. Khan M. A. M. Development of a code search engine using natural language processing technique: Graduate thesis / Mohammad Abdullah Matin Khan. – IUT, Department of Computer Science and Engineering, 2023. – 65 p.
9. Code semantic enrichment for deep code search / [Zhongyang Deng, Ling Xu, Chao Liu et al.] // Journal of Systems and Software. – 2024. – Vol. 207. – P. 111856. DOI: 10.1016/j.jss.2023.111856
10. Code Search Is All You Need? Improving Code Suggestions with Code Search / [Junkai Chen, Xing Hu, Zhenhao Li et al.] // Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024), Lisbon, Portugal, April 14–20, 2024. – 2024. – Article No. 73. – P. 1–13. DOI: 10.1145/3597503.3639085
11. A Survey on Transformer-based Models in Code Summarization / [Suraj Nate, Om Patil, Shreenidhi Medar, Jyoti Deshmukh] // International Research Journal on Advanced Engineering Hub (IRJAEH). – 2025. – Vol. 3. – P. 740–745. DOI: 10.47392/IRJAEH.2025.0103
12. Towards Enhancing Coherence in Extractive Summarization: Dataset and Experiments with LLMs / [Mihir Parmar, Hanieh Deilamsalehy, Franck Dernoncourt et al.] // Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing. – 2024. – P. 19810–19820. DOI: 10.18653/v1/2024.emnlp-main.1106
13. Strengthening Sentence Similarity Identification Through OpenAI Embeddings and Deep Learning / [Nilesh Bhikaji Korade, Mahendra B. Salunke, Amol Bhosle et al.] // International Journal of Advanced Computer Science and Applications. – 2024. – Vol. 15, No. 4. – P. 821–829. DOI: 10.14569/IJACSA.2024.0150485
14. OpenAI. New and improved embedding model [Electronic resource] // OpenAI. – Mode of access: https://openai.com/index/new-and-improved-embedding-model (date of access: 09.04.2025). – Title from screen.
15. A Survey of Text Representation and Embedding Techniques in NLP / [Rajvardhan Patil, Sorio Boit, Venkat N. Gudivada, Jagadeesh Nandigam] // IEEE Access. – 2023. – Vol. 11. – P. 36120–36146. DOI: 10.1109/ACCESS.2023.3266377
16. Applications of natural language processing and large language models in materials discovery / [Xue Jiang, Weiren Wang, Shaohan Tian et al.] // npj Computational Materials. – 2025. – Vol. 11. DOI: 10.1038/s41524-025-01554-0
17. Qdrant. Qdrant Vector Database: High-performance vector similarity search [Electronic resource] // Qdrant Documentation. – Mode of access: https://qdrant.tech/qdrant-vector-database (date of access: 09.04.2025). – Title from screen.
18. Elastic. Elasticsearch: The Official Distributed Search & Analytics Engine [Electronic resource] // Elastic. – Mode of access: https://www.elastic.co/elasticsearch (date of access: 09.04.2025). – Title from screen.
19. A Unified Framework for Rank-based Evaluation Metrics for Link Prediction in Knowledge Graphs / [Charles Tapley Hoyt, Max Berrendorf, Mikhail Galkin et al.]. – Ithaca: arXiv, 2022. – 18 p. – (Preprint / arXiv; 2203.07544). DOI: 10.48550/arXiv.2203.07544

© Boiko V. O., 2025
DOI 10.15588/1607-3274-2025-3-6