A code search engine for software ecosystems

Chris Pfaff, Elena Baninemeh, Siamak Farshidi and Slinger Jansen

Department of Information and Computer Science. Utrecht University, Utrecht, The Netherlands

Abstract

Searching and reusing source code play an increasingly significant role in the daily tasks of software developers. While code repositories, such as GitHub and Stackoverflow, may provide some results, a code search engine is generally considered most helpful when searching for code snippets as they typically crawl data from a wide range of code repositories. Code search engines enable software developers to search for code snippets using search terms. The accuracy of the search results can be increased if the searchers' intent can be modeled and predicted correctly. This study proposes a novel code search engine to model user intents through a dialogue system and then suggests a ranked list of code snippets that can meet user requirements.

Keywords

code search, machine learning, indexing source code, code search engine, ranking code snippets

1. Introduction

Software is becoming increasingly important in our modern world since it solves a wide range of problems that arise in daily life. Software is driven by programming code which is stored in so-called code repositories. These online code repositories contain billions of lines of code and explain the workings of millions of software programs. As repositories and the code they contain grow, extracting the software's inner workings can be increasingly challenging. These workings are captured at the method level, which captures several lines of code working together to address a problem. SearchSECO[1] introduces a worldwide index of the open-source software ecosystem, paving the way for creating a database consisting of all the different software methods used in these repositories. While many code snippet search engines exist, with recent research mainly applying Deep Learning (DL) techniques to the field, they still fail to capture a workable user experience and a valuable ranking of the code snippets that this study aims to implement.

A wide range of different indexing, querying, searching, and ranking approaches have been introduced and actively developed in the literature. However, applying these existing approaches to the SearchSECO global ecosystem database and evaluating different results seems to be an exciting research direction. As many existing systems struggle with connecting the user experience to the search engine itself. They either incorporate advanced searching using regular

B.E.N.E.V.O.L'22

**Corresponding author.

c.j.pfaff@students.uu.nl (C. Pfaff); e.banimeh@uu.nl (E. Baninemeh); s.farshidi@uu.nl (S. Farshidi); s.jansen@uu.nl (S. Jansen)

https://chrispfaff.dev/ (C. Pfaff)

expressions in the query itself or do not holster high precision for the results. Furthermore, the detailed code quality information of code repositories is not readily available, so software engineering researchers often need to put extra effort into evaluating the code qualities and efficiency of the search results.

This research aims to gain insight into how to index, rank, and search code methods (snippets) effectively and efficiently. Designing and creating a search engine based on user intent and existing source code models presents a ranked list of software snippets. These snippets can be hashed and checked against the existing SearchSECO database to extract authorship, licensing, vulnerabilities, etc. The scientific contribution consists of a systematic literature study on indexing, ranking, and searching through code, together with an implementation using existing techniques from the field to show the validity of a proposed solution. This work will enable software engineers to use a user-friendly solution while also opening up the field for future research on this topic and providing a basis for user interaction with the SearchSECO platform.

This study introduces a code search engine for indexing, querying, searching, and ranking code snippets of the worldwide open source software ecosystem. Figure 3 shows the constituent components of the proposed code search engine. Through bundling existing research and trying our system against the existing SearchSECO hash database, we aim to realize an extension of existing work done in the field of source code exploration. This will mainly be accomplished by improving user search intent with AI and dialogue system technologies.

2. Research approach

In this section, we elaborate on the research methods and relate them to individual research questions to which they apply.

2.1. Research questions

The main research question in this study is as follows: *(MRQ) What are the typical search engine components for indexing, querying, searching, and ranking code snippets?* We formulated the following research questions to address the MRQ: **(**RQ_1**)** *Which (metadata) code features can be used for indexing code snippets?* **(**RQ_2**)** *Which indexing techniques can be used for storing code snippets in a code base?* These first two research questions are relevant because they form the basis for extracting usable and rankable code snippets from several different source code repositories. The system will have no extensive knowledge base without a proper view of existing indexing techniques. The next subquestion handles the querying aspect of the final engine. The third research question helps us to understand how to query the indexed code snippets: **(**RQ_3**)** *Which search query techniques are available in the literature?* To create a useful user entry point, the state-of-the-art search queries for code have to be studied. Diverging away from indexing and searching, the ranking of code snippets should also be investigated. The fourth research question handles this subject. **(**RQ_4**)** *Which ranking algorithms are available in the literature for sorting code snippet results?* Once the code snippets have been indexed and the different searching techniques are defined, these two parts have to be connected using a state-of-the-art ranking algorithm. Finally, the last research question provides the basis for evaluating the selected search engine components based on the design decisions according to

RQ_1, RQ_2, RQ_3, and RQ_4 in building a search engine that incorporates indexing, ranking, and searching to create an end-user experience. **(RQ_5)** *What combination of techniques provides the best code snippet search experience?*

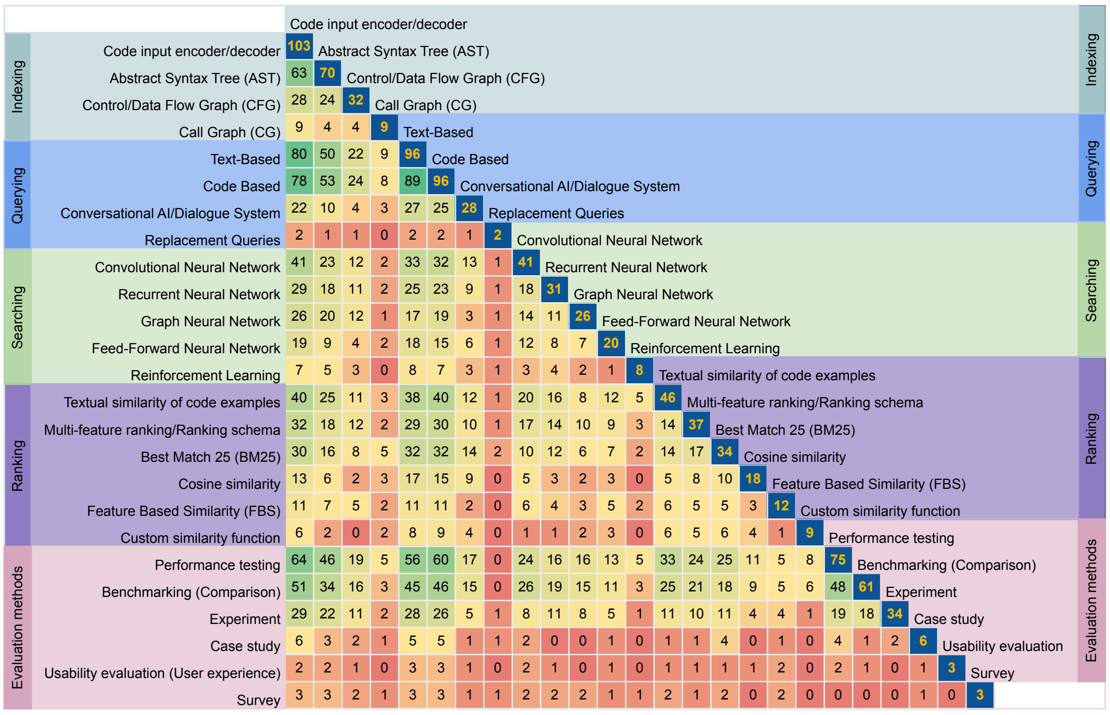
2.2. Research methods

Research methods are classified based on their data collection techniques (interview, observation, literature, etc.), inference techniques (taxonomy, protocol analysis, statistics, etc.), research purpose (evaluation, exploration, description, etc.), units of analysis (individuals, groups, process, etc.), and so forth [2]. Multiple research methods can be combined to achieve a fuller picture and a more in-depth understanding of the studied phenomenon by connecting complementary findings that conclude from the use of methods from the different methodological traditions of qualitative and quantitative investigation [3]. This study will employ systematic literature review (SLR) and design science to answer the research questions.

We employed an SLR based on the guidelines suggested by [4] answer the first four research questions. First, we started with a set of primary studies to understand the initial concepts and to extract potential keywords that can be used to retrieve more relevant studies. Next, we created a search term for querying several digital libraries (such as ACM, Scopus, and Google Scholar). Afterward, we employed a set of inclusion and exclusion criteria and quality assessments to reduce the number of low-quality publications. Finally, 190 publications have been selected for data collection and answering the research questions. These 190 papers were then screened for their content and, based on their relevancy to the subject of code snippet search, filtered for the next phase of the SLR. This left 168 papers for the component analysis. From these 168 papers, the components were extracted to create a components table that depicts the different components and their relevancy in the literature study. As some selected papers did not contain sufficient components or described a single method for indexing, they were taken out of the final component analysis. This resulted in 132 papers as a baseline for the component analysis. A final abstraction from the different component tables can be seen in Figure 2. This table shows the combined frequency of the applied techniques in the literature. Each color depicts a different category of subjects within the SLR: Indexing, Querying, Searching, Ranking, and the evaluation methods proposed in each paper. The colors depict the frequency at which the different concepts are combined in the papers. Text-based and code-based querying are combined frequently and, therefore, have a green color. On the contrary, Replacement queries are mentioned only two times in total and zero times in combination with text-based. Therefore it is red. The blue diagonal depicts the number of times a concept is used within a paper.

Based on the collected design decisions and requirements of code search engines, the design cycle [5] will be applied for designing and implementing the proposed code search engine.

A code search engine as a software artifact will be designed and implemented to answer the last research question (RQ_5). Using Design Science as proposed by [5], the main research performed in this study considers the creation of a system that considers the SLR's considerations. As the fifth research question is a design question, we design and create a system to see if our proposed solution has a real-life impact. This will be evaluated based on the SLR results, most likely resulting in expert interviews together with benchmarks of system performance.



		Code input encoder/decoder	Abstract Syntax Tree (AST)	Control/Data Flow Graph (CFG)	Call Graph (CG)	Text-Based	Code Based	Conversational AI/Dialogue System	Replacement Queries	Convolutional Neural Network	Recurrent Neural Network	Graph Neural Network	Feed-Forward Neural Network	Reinforcement Learning	Textual similarity of code examples	Multi-feature ranking/Ranking schema	Best Match 25 (BM25)	Cosine similarity	Feature Based Similarity (FBS)	Custom similarity function	Performance testing	Benchmarking (Comparison)	Experiment	Case study	Usability evaluation	Survey
Indexing	Code input encoder/decoder	103																								
	Abstract Syntax Tree (AST)	63	70																							
	Control/Data Flow Graph (CFG)	28	24	32																						
	Call Graph (CG)	9	4	4	9																					
Querying	Text-Based	80	50	22	9	96																				
	Code Based	78	53	24	8	89	96																			
	Conversational AI/Dialogue System	22	10	4	3	27	25	28																		
	Replacement Queries	2	1	1	0	2	2	1	2																	
Searching	Convolutional Neural Network	41	23	12	2	33	32	13	1	41																
	Recurrent Neural Network	29	18	11	2	25	23	9	1	18	31															
	Graph Neural Network	26	20	12	1	17	19	3	1	14	11	26														
	Feed-Forward Neural Network	19	9	4	2	18	15	6	1	12	8	7	20													
	Reinforcement Learning	7	5	3	0	8	7	3	1	3	4	2	1	8												
Ranking	Textual similarity of code examples	40	25	11	3	38	40	12	1	20	16	8	12	5	46											
	Multi-feature ranking/Ranking schema	32	18	12	2	29	30	10	1	17	14	10	9	3	14	37										
	Best Match 25 (BM25)	30	16	8	5	32	32	14	2	10	12	6	7	2	14	17	34									
	Cosine similarity	13	6	2	3	17	15	9	0	5	3	2	3	0	5	8	10	18								
	Feature Based Similarity (FBS)	11	7	5	2	11	11	2	0	6	4	3	5	2	6	5	5	3	12							
	Custom similarity function	6	2	0	2	8	9	4	0	1	1	2	3	0	6	5	6	4	1	9						
Evaluation methods	Performance testing	64	46	19	5	56	60	17	0	24	16	16	13	5	33	24	25	11	5	8	75					
	Benchmarking (Comparison)	51	34	16	3	45	46	15	0	26	19	15	11	3	25	21	18	9	5	6	48	61				
	Experiment	29	22	11	2	28	26	5	1	8	11	8	5	1	11	10	11	4	4	1	19	18	34			
	Case study	6	3	2	1	5	5	1	1	2	0	0	1	0	1	1	4	0	1	0	4	1	2	6		
	Usability evaluation (User experience)	2	2	1	0	3	3	1	0	2	1	0	1	0	1	1	1	1	2	0	2	1	0	1	3	
	Survey	3	3	2	1	3	3	1	1	2	2	2	1	1	2	1	2	0	2	0	0	0	0	1	0	3

Figure 1: shows SLR results in a table abstraction

2.3. Code search engine requirements: SLR

This section describes the results from the SLR depicted in Figure 2, primarily focusing on code search engines and their potential components to answer the research questions and eliciting the requirements for designing and implementing the code search engine in this study. Based on the SLR, we observed that code encoders/decoders and abstract syntax trees had been mainly employed in the literature for indexing code snippets. Our observations can be considered different design decisions in designing and implementing different search engines. For instance, a search engine may be designed incorporating a code encoder/decoder using AST and neural network techniques to find similar snippets to the user query.

Indexing - Several methods exist for indexing the code methods/snippets into a database. The challenge in this field is truly capturing the semantic meaning of a code snippet. As two code snippets could semantically solve the same problem, their syntax or way of solving the problem could be different. Recent code search engines increasingly use machine learning to encode and transform code snippets so they can be analyzed and indexed efficiently. The SLR found that many recent papers use an Encoder/Decoder system based on semantics and Abstract Syntax Tree (AST) to capture a code snippet in vector space. Most of these papers apply a Convolutional Neural Network (CNN) or a Recurrent Neural Network (RNN) as the neural network architecture for storing the code snippets in vector space.

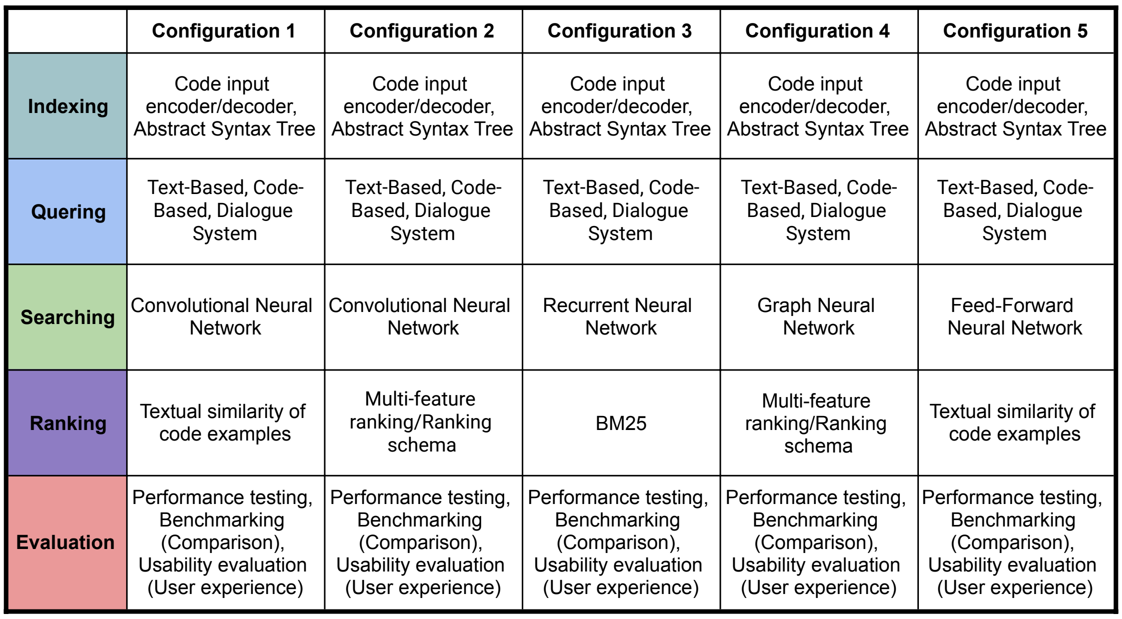
Ranking - For ranking the code snippets, most recent studies used a multi-feature ranking scheme, taking into consideration the different features of a code snippet and then ranking the code snippets based on the relevance to the input query string. The ranking was often based on either the cosine similarity of the encoded query and the encoded code snippets in vector space or a custom similarity function was used to compare input strings to code snippets. This provided encoded code snippets with a ranking score, allowing the search engine to decode them to be visible to the user.

Searching - Almost all selected publications in the SLR used NLP to analyze a natural language string as an input query. Some papers extended this method with code-to-code search. These compared the original code snippet to a dataset/corpus of other code snippets to find syntactic/semantic similarities between code snippets. This can be used for plagiarism detection and authorship questions. We found a gap in research on alternatives for searching through code snippets. Instead of using a plain string or code search box, we propose using query reformulation by a conversational AI/dialogue system. We think that by asking the user the right questions and interacting on the query side, the query can be optimized to find its best code snippet fit in vector space. As similarity analysis is exact and even the most minor adjustments to input query could drastically change the encoded vector created by any vector algorithm, we think optimizing the query may be vital to increase the accuracy of the search engine.

Querying - The existing studies seem to miss considering how essential a user query is in finding a correct ranking. Most research centers on using new NLP techniques or implementing state-of-the-art neural networks to increase code encoding performance. However, when the input query is insufficient and translated into vector space, the results would not satisfy searchers' expectations, so the accuracy of the search engine would not be high. In almost a quarter of cases, the query was the problem in giving the correct answer when testing Neural Code Search NCS ([6]). Code-to-code search engines similar to FaCoY ([7]) have adopted query alternation as a core concept of their workings. However, in general, code search is hardly ever used as a natural language query. (Less than half of the studies in SLR use Replacement Queries). We believe that through a dialogue system, a user could change their query to be more specific, increasing the query's quality and precision in vector space. When the vector encoded query is defined more sharply, the similarity functions have a bigger chance of ranking similar code snippets in higher order.

Search engine configurations - The different concepts depicted in (Figure 2) can be combined in different configurations to analyze the usefulness of applying them together. Based on the table, a few frequent starting techniques are chosen based on which these initial configurations come to be. The five different starting configurations along with their respective reasoning, are as follows:

All configurations use a code input encoder/decoder in combination with an Abstract Syntax Tree to index the code snippets. This combination captures both the semantics of code, which are stored in encoding and the syntax, which is stored in the AST. This combination of methods is used in many papers and is the most frequent combination of indexing by a considerable margin. Furthermore, all configurations combine code-based, text-based, and dialogue systems for querying. Code-based and text-based querying are frequently used together, and we believe that by applying a dialogue system, the algorithm will understand the user intent better. All five configurations also use the same evaluation methods, as regardless of the techniques used, most papers evaluated their presented algorithm using performance testing and benchmarking. We intend to apply these two evaluation models, regardless of configuration and might add a



	Configuration 1	Configuration 2	Configuration 3	Configuration 4	Configuration 5
Indexing	Code input encoder/decoder, Abstract Syntax Tree	Code input encoder/decoder, Abstract Syntax Tree	Code input encoder/decoder, Abstract Syntax Tree	Code input encoder/decoder, Abstract Syntax Tree	Code input encoder/decoder, Abstract Syntax Tree
Quering	Text-Based, Code-Based, Dialogue System	Text-Based, Code-Based, Dialogue System	Text-Based, Code-Based, Dialogue System	Text-Based, Code-Based, Dialogue System	Text-Based, Code-Based, Dialogue System
Searching	Convolutional Neural Network	Convolutional Neural Network	Recurrent Neural Network	Graph Neural Network	Feed-Forward Neural Network
Ranking	Textual similarity of code examples	Multi-feature ranking/Ranking schema	BM25	Multi-feature ranking/Ranking schema	Textual similarity of code examples
Evaluation	Performance testing, Benchmarking (Comparison), Usability evaluation (User experience)	Performance testing, Benchmarking (Comparison), Usability evaluation (User experience)	Performance testing, Benchmarking (Comparison), Usability evaluation (User experience)	Performance testing, Benchmarking (Comparison), Usability evaluation (User experience)	Performance testing, Benchmarking (Comparison), Usability evaluation (User experience)

Figure 2: shows possible configurations using table components

usability evaluation to evaluate the user experience with the dialogue system. As the three times, a usability evaluation was performed, it was always done in combination with the dialogue system, as seen in Figure 2. For each possible configuration, only the searching and ranking methods change based on frequency alone. These cannot be classified as being a good or a bad combination. Hence the following section describes the choices for the different combinations.
Configuration 1 - The variable components for this configuration are the Convolutional Neural Network(CNN) and the Textual similarity of code examples. These are used 20 times together, more than any other ranking technique the CNN is paired with. Furthermore, Wang et al. [8] improved upon the state-of-the-art in semantic code search by combining encoded code snippets and convolutional neural networks with layer-wise attention. This enables embedding both code and query into a unified semantic vector space so the textual similarity may be applied. This approach has then been improved upon by others like Fang et al. [9] to increase the overall performance of the configuration. However, using a CNN and expanding it with different attention mechanisms may not be enough when representing code snippets and their combination of syntax and semantics.
Configuration 2 - The variable components for this configuration are the CNN and the Multi-feature ranking/Ranking schema. These are used 17 times together, which is a little less when compared to the textual similarity, but regardless the properties of a ranking schema may improve performance when bundled with a CNN. The ranking schema could complement the different patterns applied to the CNN, so the ranking and neural network may be a better fit. The main disadvantage of using a CNN is that patterns must be found or applied in the training data. If it is not available in the training data, then the CNN adds little value, and other attention mechanisms may be used [9].
Configuration 3 - The variable components for this configuration are a Recurrent Neural

Network (RNN) and the Best Match 25 (BM25) ranking algorithm. These were chosen based on the frequency of which searching algorithm best suited BM25. With 12 papers mentioning them, RNN is more frequently used than the other searching algorithms. Additionally, Ling et al. [10] showed that using an RNN in their AdaCS outperforms state-of-the-art deep code search methods on new codebases. This configuration may perform better when training data is scarce in the specific code domain. However, when training RNN architectures, it is essential to consider long training time and enabling parallel training [11].
Configuration 4 - The variable components for this configuration are the Graph Neural Network (GNN) and the Multi-feature ranking/Ranking schema. These are used ten times together, which makes the Ranking schema the best choice when using a GNN. Moreover, Wang et al. [12] combine AST with semantic code search in a graph-type approach. They call this FA-AST and apply two different types of GNN to measure code pair similarity. While CNN and RNN have their uses in code search, structuring the code as a graph and applying GNN, more information can be structured and used [12] [13]. Wang et al. [12] do state that their approach may not perform as well on complex real-life datasets, as the reliability of the GNN relies on a solid dataset.
Configuration 5 - The variable components for this configuration are the Feed-Forward Neural Network (FFNN) and the Textual similarity of code examples. These are used 12 times together, making their combination more frequent than other ranking techniques with the FNN. The advantage of using an FFNN is that it simplifies the flow of the neural network. Fujiware et al. [14] proposed a method using FNN based on learning and searching steps. Furthermore, while this approach has a more straightforward structure than RNN, the researchers mention using RNN in the future, depicting the relative simplicity of using FFNN.

2.4. Code search engine components: Design science

In this section, we elaborate on the constituent components of the proposed code search engine and its workflow (Figure 3). Note, the data collection and indexing phase has been adopted from our earlier work on dataset indexing pipeline [15].
Code repositories contain metadata and data regarding software projects, such as source code, documentation, notes, commits, and web pages. A code repository can be public or private and be offered by Git platforms, such as GitHub and Gitlab.
Web crawling / Web APIs is the process of a spider bot that systematically browses code repositories and extracts data and metadata of software projects. It retrieves such contents in structured formats (e.g., JSON or key-values). It is essential to highlight that Git platforms might offer a set of Web APIs that make the data extraction process much more straightforward.
Metadata extraction is the process of retrieving any embedded metadata present in a document. It is responsible for extracting metadata features such as classes and properties inside an RDF document or textual contents of potential features mentioned on landing pages of code repositories. The metadata of the retrieved documents will be extracted based on domain experts' rules.
Language model employs various methods like Bag of Words or pre-trained algorithms like BERT to specify the probability of a given sequence of words occurring in a textual document. It analyzes bodies of documents to convert qualitative information into quantitive information.

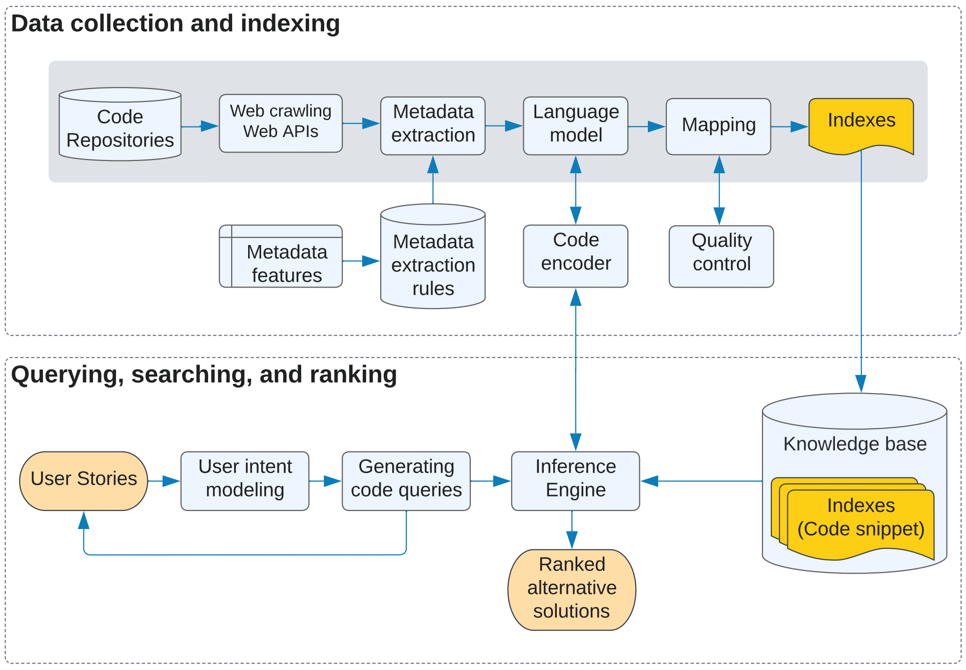


Figure 3: shows the constituent components of the proposed code search engine.

The language model is employed to understand developer comments and notes to extract more data from source codes. In other words, the language model calculates similarities among metadata features of a particular code snippet and potential contextual information that could be assigned to it. Since contextual information, such as domain keywords, can be seen as vectors, we can use different similarity approaches, such as the cosine similarity or Jaccard index, to calculate the similarity of these vectors [16].
Code encoder is a hash generator that produces unique identifiers of code snippets and can be used to index and retrieve them from a database because it helps accelerate the process; it is much easier to find an item using its shorter hashed vital than its original value.
Mapping refers to the process of adding external contextual information, such as domain keywords, to extracted metadata features based on predefined rules by domain experts and language models' predictions.
Indexing is a data structure technique to efficiently retrieve code repositories on some attributes on which the indexing has been done. Indexing techniques can reduce the processing time of a search query. For instance, inverted indexing categorizes datasets based on collected topics and external contextual information. Then, the final indexes can be ingested in a document data storage (such as ElasticSearch or Apache Solr).
Quality control is an essential phase of the data collection and indexing pipeline as the quality of the mapping will be evaluated based on the number of values that mapped correctly to metadata features. This process can be considered hyperparameter tuning, which is choosing a set of optimal hyperparameters for the similarity approaches, such as cosine similarity and

Jaccard index.
Metadata features are high-level documents that establish a common form of structuring and understanding code snippets and include principles and implementation issues. There are many metadata standards proposed for specific disciplines. For instance, Schema.org is an open metadata standard that indicates how a source code should be organized and how it can be related to other software assets (for instance, documentation of the project).
Metadata extraction rules are a set of human-made rules which domain experts should define to increase the accuracy of metadata extraction and refine potential extracted values that can be assigned to the metadata features. An example of potential rules in the rule base is presented as follows. The example shows a metadata feature called "URL", which is a "unique identifier", and its data type is "codeRepository". The Regular expression in the constraint field ensures a potential value for this metadata feature starts with "www/http:/https:" and it should not have any white space characters. The *metadata extraction rules* component should look for metadata features such as "ISBN", "GTIN", or "UUID" to extract potential values that can be mapped to "URL".

```
"URL": [
  "datatype" : "codeRepository",
  "description": "Link to the repository where the
  un-compiled, human-readable code and related code
  is located  (SVN, GitHub, CodePlex).",
  "constraint": ["(www|http:|https:)+[^\s]+[\w]"],
  "suggested fields": ["ISBN", "GTIN",
  "UUID", "URI","URL","id"]], ...
```

Knowledge base is a database that stores and retrieves all indexes of the collected code snippets.
User stories are informal, general explanations of code snippets written from the software developer's perspective. For instance, "*how can I remove spaces from a string in python*".
User intent modeling or search intent states which goal or intention a searcher has when interacting with the search engine. Sometimes, the user's intention might be beyond what she has requested from the search engine, for instance, in the previous example, and the searcher might be looking for a code snippet that uses regular expressions in python to remove all types of white spaces. So, the search engine should be able to predict the searcher's following possible queries and suggest alternative solutions to increase efficiency and effectiveness in fulfilling users' requirements.
Generating code queries generates a set of code snippets based on user intentions. In other words, it converts user requirements into code queries.
Inference Engine searches the knowledge base according to the generated code queries and then returns a set of ranked feasible code snippets.

3. Discussion and conclusion

In this study, we propose a search engine incorporating different techniques and methodologies from state-of-the-art research on code search. By incorporating the latest algorithms and training data, we aim to train our system to perform as close to the state-of-the-art. Extending this, we aim to incorporate a dialogue system or conversational AI to improve query reformulation, providing the user with a better experience using the code search engine. We hypothesized that the ranking and similarity analysis might improve by applying query reformulation as the query contains more metadata about the code snippet the user desires.

Research in software reuse involves topics such as design for reuse and making reusable components easier to find. For instance, in component-based software engineering (CBSE), reuse is designed, and components are developed and packaged for that purpose. The selection of a component is driven by a set of feature requirements specified in advance [17, 18, 19]. Work on software reuse repositories has included packaging code into libraries for reuse, constructing archives of reusable components, and searching on those repositories [1].

In the literature, we observed that search engines had been studied for decades by other researchers and practitioners, so the first four questions can be addressed through the SLR. Accordingly, a significant part of this study is highly dependent on the quality of the selected papers based on the quality assessments.

Perkusich et al. [20] stated that (1) the number of studies employing intelligent techniques in software engineering is increasing, (2) reasoning under uncertainty, search-based solutions, and machine learning are the most used intelligent techniques in the field; (3) the risks of applying such intelligent techniques in the software engineering field is considerably high, so (4) more empirical evaluation and validation for such methods are required.

Based on design science research, an evaluation method should be employed to assess the efficiency and effectiveness of the code search engine. While performing the SLR, we observed that various evaluation methods, such as performance testing and user experience (Table **??**), had been typically employed for assessing search engines. So, in this study, we will conduct a usability evaluation (user experience) to evaluate the code search engine. Besides this UX evaluation, the performance of the used machine learning techniques will be benchmarked to the state-of-the-art algorithms described in the literature studies. Codexglue [21] is one example of a machine learning benchmarking dataset that may be used to evaluate the search performance of the final implementation.

References

[1] S. Jansen, S. Farshidi, G. Gousios, T. van der Storm, J. Visser, M. Bruntink, Searchseco: A worldwide index of the open source software ecosystem, in: The 19th Belgium-Netherlands Software Evolution Workshop, BENEVOL 202, CEUR-WS. org, 2020.

[2] J. R. Meredith, A. Raturi, K. Amoako-Gyampah, B. Kaplan, Alternative research paradigms in operations, Journal of operations management 8 (1989) 297–326.

[3] R. B. Johnson, A. J. Onwuegbuzie, Mixed methods research: A research paradigm whose time has come, Educational researcher 33 (2004) 14–26.

[4] B. Kitchenham, S. Charters, Guidelines for performing systematic literature reviews in software engineering, ebse technical report. (2007).

[5] R. J. Wieringa, Design science methodology for information systems and software engineering, Springer, 2014.

[6] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, S. Chandra, Retrieval on source code: a neural

code search, in: Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2018, pp. 31–41.

[7] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, Y. L. Traon, Facoy: a code-to-code search engine, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 946–957.

[8] H. Wang, J. Zhang, Y. Xia, J. Bian, C. Zhang, T.-Y. Liu, Cosea: Convolutional code search with layer-wise attention, arXiv preprint arXiv:2010.09520 (2020).

[9] S. Fang, Y.-S. Tan, T. Zhang, Y. Liu, Self-attention networks for code search, Information and Software Technology 134 (2021) 106542.

[10] C. Ling, Z. Lin, Y. Zou, B. Xie, Adaptive deep code search, in: Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 48–59.

[11] M. White, M. Tufano, C. Vendome, D. Poshyvanyk, Deep learning code fragments for code clone detection, in: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2016, pp. 87–98.

[12] W. Wang, G. Li, B. Ma, X. Xia, Z. Jin, Detecting code clones with graph neural network and flow-augmented abstract syntax tree, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2020, pp. 261–271.

[13] X. Ling, L. Wu, S. Wang, G. Pan, T. Ma, F. Xu, A. X. Liu, C. Wu, S. Ji, Deep graph matching and searching for semantic code retrieval, ACM Transactions on Knowledge Discovery from Data (TKDD) 15 (2021) 1–21.

[14] Y. Fujiwara, N. Yoshida, E. Choi, K. Inoue, Code-to-code search based on deep neural network and code mutation, in: 2019 IEEE 13th International Workshop on Software Clones (IWSC), IEEE, 2019, pp. 1–7.

[15] S. Farshidi, Z. Zhao, An adaptable indexing pipeline for enriching meta information of datasets from heterogeneous repositories, in: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2022, pp. 472–484.

[16] M. Steyvers, T. Griffiths, Probabilistic topic models, in: Handbook of latent semantic analysis, Psychology Press, 2007, pp. 439–460.

[17] S. Farshidi, S. Jansen, M. Deldar, A decision model for programming language ecosystem selection: Seven industry case studies, Information and Software Technology 139 (2021) 106640.

[18] S. Farshidi, Multi-Criteria Decision-Making in Software Production, Ph.D. thesis, Utrecht University, 2020.

[19] S. Farshidi, S. Jansen, A decision support system for pattern-driven software architecture, in: European Conference on Software Architecture, Springer, 2020, pp. 68–81.

[20] M. Perkusich, L. C. e Silva], A. Costa, F. Ramos, R. Saraiva, A. Freire, E. Dilorenzo, E. Dantas, D. Santos, K. Gorgônio, H. Almeida, A. Perkusich, Intelligent software engineering in the context of agile software development: A systematic literature review, Information and Software Technology 119 (2020) 106241.

[21] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al., Codexglue: A machine learning benchmark dataset for code understanding and generation, arXiv preprint arXiv:2102.04664 (2021).