12-2022

Opportunities and challenges in code search tools

Chao LIU
Zhejiang University

Xin XIA
Huawei

David LO
Singapore Management University, davidlo@smu.edu.sg

Cuiying GAO
Harbin Institute of Technology

Xiaohu YANG
Zhejiang University

See next page for additional authors

Citation

LIU, Chao; XIA, Xin; LO, David; GAO, Cuiying; YANG, Xiaohu; and GRUNDY, John. Opportunities and challenges in code search tools. (2022). *ACM Computing Surveys*. 54, (9), 1-35.
Available at: https://ink.library.smu.edu.sg/sis_research/6926

Author

Chao LIU, Xin XIA, David LO, Cuiying GAO, Xiaohu YANG, and John GRUNDY

Published in ACM Computing Survey, 54, 9, Article 196 (December 2022)
DOI:10.1145/3480027

Opportunities and Challenges in Code Search Tools

CHAO LIU, Zhejiang University, China
XIN XIA[*], Monash University, Australia
DAVID LO, Singapore Management University, Singapore
CUIYUN GAO, Harbin Institute of Technology (Shenzhen), China
XIAOHU YANG, Zhejiang University, China
JOHN GRUNDY, Monash University, Australia

Code search is a core software engineering task. Effective code search tools can help developers substantially improve their software development efficiency and effectiveness. In recent years, many code search studies have leveraged different techniques, such as deep learning and information retrieval approaches, to retrieve expected code from a large-scale codebase. However, there is a lack of a comprehensive comparative summary of existing code search approaches. To understand the research trends in existing code search studies, we systematically reviewed 81 relevant studies. We investigated the publication trends of code search studies, analyzed key components, such as codebase, query, and modeling technique used to build code search tools, and classified existing tools into focusing on supporting seven different search tasks. Based on our findings, we identified a set of outstanding challenges in existing studies and a research roadmap for future code search research.

CCS Concepts: • **Software and its engineering** → **Search-based software engineering**.

Additional Key Words and Phrases: code search, code retrieval, modeling

ACM Reference Format:
Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John Grundy. 2020. Opportunities and Challenges in Code Search Tools. *ACM Comput. Surv.* 1, 1, Article 1 (January 2020), 35 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

In modern software development, code search is one of the most frequent activities [89, 110, 117, 136]. Studies have shown that more than 90% of developers' search efforts aim at finding code to reuse [7]. This is because developers favor searching for existing or similar high-quality code to mitigate their learning burdens and enhance their software development productivity and quality [17, 18, 35].

"Code search" refers to retrieval of relevant code snippets from a code base, according to the intent of a developer that they have expressed as a search query [31, 35, 59, 69, 88, 108, 116, 121]. With the advent of large code repositories and sophisticated search capabilities, code search is not only a key software development activity but also supports many other promising software engineering tasks. For example, code search tools

[*] Corresponding Author: Xin Xia.

Authors' addresses: Chao Liu, liuchaoo@zju.edu.cn, Zhejiang University, China; Xin Xia, Xia@monash.edu, Monash University, Australia; David Lo, davidlo@smu.edu.sg, Singapore Management University, Singapore; Cuiyun Gao, gaocuiyun@hit.edu.cn, Harbin Institute of Technology (Shenzhen), China; Xiaohu Yang, yangxh@zju.edu.cn, Zhejiang University, China; John Grundy, John.Grundy@monash.edu, Monash University, Australia.

are helpful for bug/defect localization [4, 61, 94, 119, 125, 126, 129], program repair [2, 80], and code synthesis [98, 99], among others.

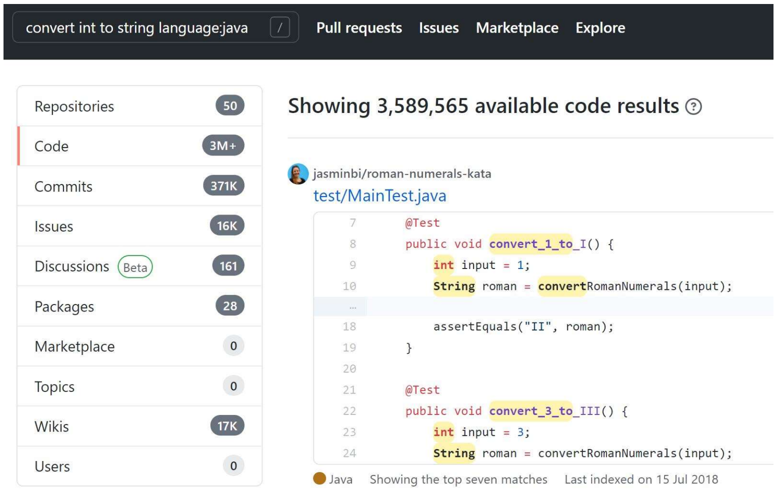
Despite the existence of numerous code search studies, to the best of our knowledge there has been no systematic study to summarize the key approaches and characteristics of existing code search tool research. Such a systematic review would help practitioners and researchers understand the current state-of-the-art code search tools and inspire their future studies. To perform a systematic review of this domain, we identified 81 relevant studies from three widely used electronic databases, including ACM Digital Library, IEEEXplore, and ISI Web of Science. We investigated the following research questions (RQs) in this study:

- **RQ1.** *What are the emerging publication trends for code search studies?* The reviewed 81 studies show that the popularity of the code search topic has increased substantially in recent years with a peak in 2019. 60% of these studies were published in conference proceedings instead of journals. 83% of studies contributed to this domain by proposing new tools rather than performing empirical/case studies of existing tools.
- **RQ2.** *What are the most important factors that contribute to existing code search tools?* From the 67 different code search tools reported, we found that deep learning (DL) based approaches are the most popular modeling techniques over the past two years. Code search tools can be classified into seven categories – text-based code search, I/O example code search, API-based code search, code clone search, binary code search, UI search, and programming video search. Only 12 studies shared an accessible replication package link in their papers or provided their tool source code in GitHub.
- **RQ3.** *How do studies evaluate code search tools?* To evaluate a code search tool, most of the studies built codebases with method-level source code written in Java. they then performed code searches with free-form queries such as text and API names. Most studies manually checked the relevancy between a query and the returned code, and evaluated the tool performance in terms of popular ranking metrics, such as Precision and MRR (Mean Reciprocal Rank).

Based on these findings and the threats discussed in all reviewed code search studies, we observed a number of challenges in existing code search tools. Generally, the codebase scale used is limited with code written in only one programming language. The quantity of search queries is also limited and cannot cover developers' various search scenario in practical usage. The state-of-the-art tools based on learning models such as deep learning still cannot solve the code search problem very well. One major reason is that learning models are optimized with low quality and quantity of training data. The effectiveness of most code search tools is verified by use of manual evaluation that suffers from subjective bias. When measuring the tool performance, the search time and tool scalability are rarely assessed. Other important performance aspects, such as code diversity and conciseness, are not considered. These challenges provide some opportunities for further research studies:

- **Benchmarks:** Developing a standard benchmark with large-scale code base written in multiple programming languages, various representative queries, and an automated evaluation method.
- **Learning Models:** Improving the learning models with better quality of training data, code representation method, and loss functions for model optimization.
- **Model Fusion:** Fusing different types of models – such as deep learning based model, traditional IR-based model, and heuristic model – to balance their advantages and disadvantages for further improvements.
- **Cross-Language Searches:** Building a multi-language code search tool to mitigate the costly deployment of a tool for code in different programming languages.
- **Search Tasks:** Supporting new kinds of code search tasks, such as searching UI (User Interface) code and the code used in programming videos.

The main contributions of this study include:



Showing 3,589,565 available code results

jasminbi/roman-numerals-kata
test/MainTest.java

```
    @Test
    public void convert_1_to_I() {
        int input = 1;
        String roman = convertRomanNumerals(input);
…
        assertEquals("II", roman);
    }

    @Test
    public void convert_3_to_III() {
        int input = 3;
        String roman = convertRomanNumerals(input);
```

Java   Showing the top seven matches   Last indexed on 15 Jul 2018

Fig. 1. Searching Java code from GitHub with the text-based query "convert int to string".

- A novel systematic review on 81 code search studies published until July 31, 2020 as a starting point for future research on code search.
- Analysis of the fundamental components – codebase, query, and model – in 67 different code search tools to help researchers understand their characteristics.
- Classification of code search tools into seven categories and analyzing the relationships between tools for each category as a basis for further comparisons and benchmarks.
- Analysis of the outstanding opportunities and challenges in code search studies based on our findings to inspire further research in this area.

The remainder of this paper is organized as follows. Section 2 briefly introduces the usage of code search tools. Section 3 presents our study methodology that we follow, and Sections 4-6 summarize the key research questions and their answers investigated in this study. Section 7 discusses the challenges for the road ahead on code search studies and presents the potential research opportunities for future work. Section 8 shows the potential threats that may affect the validity of this review. Finally, Section 9 provides a summary of this study.

2 BACKGROUND

The objective of code search tools is to retrieve relevant code from a large-scale codebase according to the intent of developers' search query. For example, GitHub search[1] is one type of tool widely used for searching for source code snippets from a large-scale codebase with millions of open source repositories. Fig. 1 shows an example of using the tool. After typing in the search query "convert int to string" with programming language choice "language:java", the GitHub search returns more than three million lines of potentially relevant code. However, the performance of this tool is not satisfactory, where the first returned code is not the expected code and it is

[1] https://github.com/search

time-consuming to check the relevancy of each code one by one. To improve such code search task performance, researchers have built many new tools. Some leverage DL techniques to improve the search accuracy, and some use clustering of returned code to reduce developers' efforts for code inspection.

The usual workflow when using such a code search tool involves the following six key components:

Codebase. In the code search task, the codebase defines the target search space, whose characteristics strongly affect the tool performance [110, 116, 136]. Different code search studies and their tools build their codebases in different ways. For example, the codebase may be constructed by a set of source/compiled code written in different programming languages (e.g., Java, Python, and C/C++). The codebase scale also varies substantially and the code may be collected from various sources, such as GitHub, FDroid, and Stack Overflow. Section 6.1 presents how developers build codebases from different perspectives.

Query. Code search tools take developers' queries as input. These queries reflect the developers' requirements during a specific software development task [19, 37, 58]. Existing code search tools can support queries in different forms and this determines how developers use the code search tools. For example, free-form text written in natural language is the most common query, which is widely used for general search engines [72, 115, 142], such as GitHub search. Some code search tools support a more structured code-based query to find similar code in their codebase [53, 82, 100]. A detailed analysis of query types is presented in Section 6.2.

Model. A code search tool should support the features of query and codebase. In general, researchers build code search tools by using three types of modeling techniques. The first is a traditional model that performs code search according to a relevancy algorithm (e.g., TF-IDF [135]) between query and candidate code [83, 90, 95]. However, traditional models usually support only text-based queries. The second is a heuristic model that leverages the code analysis technique to capture the syntactic and semantic features in code and ranks code with customized matching approaches [63, 104, 149]. The final one is use of a learning model that learns the relationship between code and query by using a large-scale dataset. Most learning models embed query and code into a shared vector space so that their similarity can be measured by the cosine similarity. DL techniques have been widely used for building code search tools in recent years. This is because a DL model shows no limit on the types of query and codebase. It learns features from large-scale data and this can substantially mitigate the difficulty in code understanding and representation as in the previous two model types [37, 141, 142].

Auxiliary Technique. Although a learning model is promising compared with the traditional and heuristic models, we cannot say a learning model is superior to the other two. This is because code search models are usually associated with auxiliary techniques, such as query reformulation [83, 150], code clustering [81], and learning from user feedback [70]. Using appropriate auxiliary techniques for a specific code search model can also improve their performance substantially [74].

Evaluation Method. To evaluate the validity of a code search tool, the relevancy of the searched code list and a query should be assessed. Manual identification is the most prevalent method. This is because the ground-truth is difficult to measure [37, 74, 115]. However, manual identification cannot scale to large numbers of queries. Therefore, researchers have also investigated other ways to mitigate manual efforts. Section 6.3 compares the tool evaluation methods that have been used.

Performance. Code search tool performance is based on its identified relevancy between query and the returned code. In most studies, code search tools care about the position of the correctly searched code in the result list. For example, MRR (mean reciprocal rank [37]) and NDCG (normalized discounted cumulative gain [22]) are two commonly used metrics. These metrics assume that developers prefer to find the "best" recommended code near the top of a result list [115, 141]. For example, supposing a tool returns three code with the expected one in the second and third places, the MRR metric measures the performance by the reciprocal rank of the first relevant code (i.e., 1/2). However, some developers want to search for more relevant code so that the ranking position can be ignored. Therefore, some code search studies evaluate the tool performance by using classification metrics, such as Precision, Recall, and F1-score (a harmonic average of Precision and Recall) [20, 63]. For the above example with three returned code, the search precision equals to the number of relevant code divided by the total count (i.e., 2/3).

3 METHODOLOGY

To perform a systematic review of code search tools, we followed the guidelines provided by Kitchenham and Charters [54] and Petersen et al. [97].

3.1 Research Questions

We wanted to identify, summarize, classify, and analyze the empirical evidence concerning different code search studies published to date. To achieve this goal, we investigate three research questions (RQs):

- **RQ1.** *What are the emerging publication trends for code search studies?* The goal of this RQ is to investigate the publication trends in terms of the publication year, publication venue, and contribution type (e.g., new tool and empirical study) of code search studies.
- **RQ2.** *What are the most important factors that contribute to existing code search tools?* This RQ investigates which modeling and auxiliary techniques have been used in different code search tools; how we can best classify these different tools; and how often do they provide accessible replication packages.
- **RQ3.** *How do studies evaluate code search tools?* This RQ aims to analyze four fundamental aspects for tool evaluation: codebase, query, evaluation method, and performance measures.

Through analysis of these RQs we also identify limitations, gaps and future research recommendations from these studies. We use these to formulate our research roadmap for future code search studies.

3.2 Search Strategy

We identified a set of search terms in code search studies that were already known to us. We refined these search terms by checking the titles and abstracts of the relevant papers, combined them with logical "OR", and formed the search string: *"code search" OR "code retrieval"*. We used the search string to perform an automated search on three widely used electronic databases including ACM Digital Library, IEEExplore, and ISI Web of Science. The search was performed on the title, abstract, and keywords of the papers. We conducted our search on July 31, 2020, and identified the studies published up until that date. As shown in Table 1, we retrieved 1,117 relevant studies with the automatic search from these three electronic databases. After discarding the duplicated studies, we obtained 692 code search studies.

Table 1. Selection of code search studies.

Process	**#Studies**
ACM Digital Library	165
IEEE Xplore	322
Web of Science	630
Automatic search from three electronic databases.	1117
Removing duplicated studies.	692
Excluding primary studies based on title and abstract.	135
Excluding primary studies based on full text - final left.	81

3.3 Study Selection

Once we retrieved the candidate studies relevant to the code search study, we performed a relevance assessment according to the following inclusion and exclusion criteria:

✓ *The paper must be written in English.*

✓ *The paper must involve at least one tool addressing the code search task.*

✓ *The paper must be a peer-reviewed full research paper published in a conference proceeding or a journal.*

✗ *Keynote records and grey literature are excluded.*

✗ *Conference studies with extended journal versions are discarded.*

✗ *The studies that propose new code search tools but did not evaluate their performance are excluded.*

✗ *The studies that apply existing code search tools for other software engineering tasks (e.g., bug localization and program repair) are ruled out.*

The inclusion and exclusion criteria were piloted by the first and forth authors starting with the assessment of 30 randomly selected primary studies. The reliability of the inclusion/exclusion decisions was measured using pairwise inter-rater reliability with Cohen's Kappa statistic [23]. The agreement rate in the pilot study was "moderate" (0.59). The pilot study helped us to develop a collective understanding of the inclusion/exclusion criteria. Then, an assessment was performed for the full list of the identified studies. The agreement rate in the full assessment was "substantial" (0.73). Disagreements were resolved after open discussions between first and fourth authors. For any case that they did not reach a consensus, the third author was consulted as a tie-breaker. Specifically, we took two weeks to finish the study selection process. As shown in Table 1, we identified 135 code search studies by inspecting the title and abstract of the retrieved studies. After checking the full text of the remaining studies, we finally obtained 81 relevant code search studies.

3.4 Data Extraction

To answer the three research questions above, we read the 81 papers carefully and extracted the required data as summarized in Table 2. Our data collection mainly focused on four kinds of information: publication information, study background, tool details, and experimental setup. To suppress the effect of subjective bias, the data collection was performed by the first and fourth authors, and verified by two senior PhD students who are not co-authors of this study and majored in computer science.

Table 2. Extracted data for research questions.

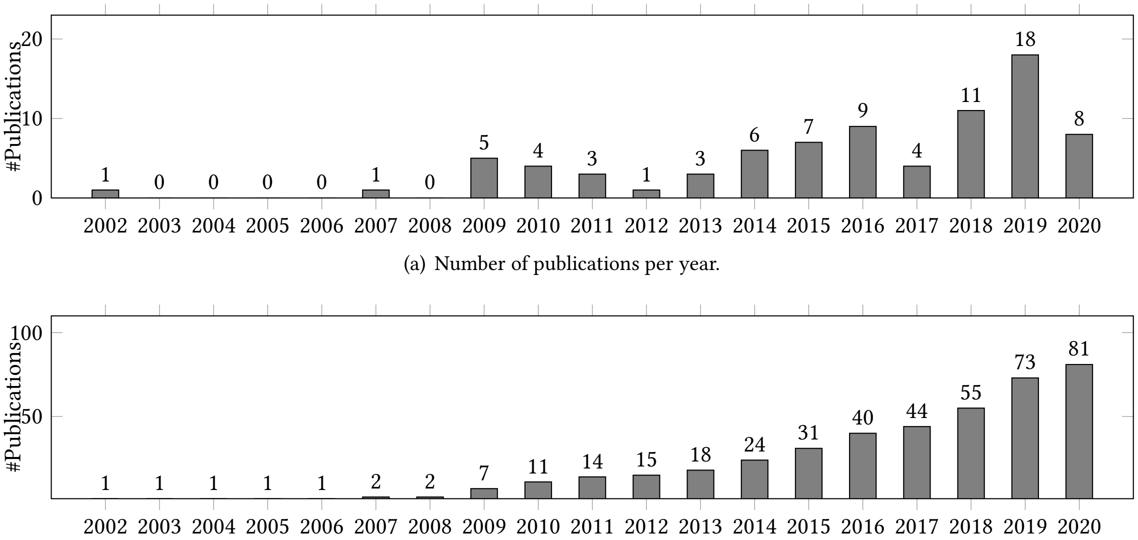
RQ	Description	Extracted Study Data
RQ1	Publication Trend	Publication year, publication venue, publication type (i.e., new tool, empirical study, and case study).
RQ2	Modeling Techniques	Model (type, major technique), auxiliary technique, tool descriptions (background, motivation, application scenario, baseline tools), Replication package link.
RQ3	Evaluation Components	Codebase (type, granularity, language, scale, source), query (type, scale, source), evaluation method, performance measure.

4 RQ1: WHAT ARE THE EMERGING PUBLICATION TRENDS FOR CODE SEARCH STUDIES?

We analyze the publication information from the 81 code search studies retrieved from Section 3, and discuss the key emerging publication trends.

4.1 Publication Trend

Fig. 2(a) shows the number of studies that were published each year. We can see that the first code search study we found was published in 2002. The popularity of code search studies has gradually been increasing from 2009, and the publication peak occurs in 2019 with 22.2% of the total numbers (and not all 2020 papers have yet appeared of course). Fig. 2(b) illustrates the cumulative counts of numbers shown in Fig. 2(a). To test the trend (i.e., increasing, decreasing, or neither) of the cumulative publication number, we performed a Cox Stuart trend test [24] at a 5% significance level. The statistical result shows a substantially increasing trend with p-value<0.01, which implies the growing popularity of the code search study in the last 18 years.



(a) Number of publications per year.

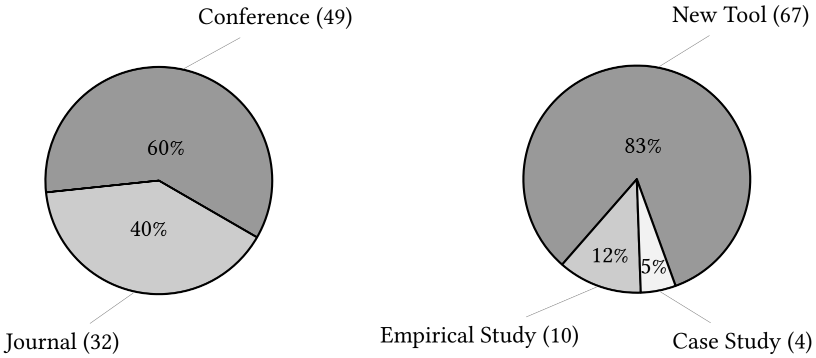
(b) Cumulative number of publications per year, which shows an increasing trend (p-value<0.01) tested by the Cox Stuart trend test [24] at a 5% significance level.

Fig. 2. Publication trend in years.

4.2 Publication Venues and Contribution Types

The 81 reviewed studies were published in various conference proceedings and journals. Fig. 3(a) shows that 60% of them were published in conference proceedings. Fig. 3(b) illustrates how studies contributed to the code search task. We can notice that 83% of the studies proposed new tools, while 12% of the studies performed empirical studies to analyze the historic data of existing code search tools [6, 7, 26, 34, 35, 86, 101, 102, 136, 139]. The remaining 5% studies performed case studies in real world, especially for enterprise usage, to investigate developers' experience and expectations of code search tools [25, 110, 116, 120].

Table 3 lists the top publication venues with at least two code search studies. These venues include a total of 56 studies, 69.1% of the total reviewed studies. These publication venues publish various kinds of code search studies: studies that propose new tools (46), empirical studies (9), case study (1). We can also observe that among these 17 venues, the top-5 popular conferences these works were published are MSR, ICSE, ASE, FSE, and EMSE; meanwhile, the top-5 journals are TSE, TOSEM, SPE, ASEJ, and TSC.



(a) Publication venue types.

(b) Contribution types.

Fig. 3. Publication Venues and Contribution Types

Table 3. Top publication venues with at least two code search studies (NT: new tool; ES: empirical study; CS: case study; TS: Total Studies).

Short Name	**Full Name**	**NT**	**ES**	**CS**	**TS**
MSR	International Conference on Mining Software Repositories	3	4	0	7
ICSE	International Conference on Software Engineering	6	0	0	6
ASE	International Conference Automated Software Engineering	6	0	0	6
FSE	Symposium on the Foundation of Software Engineering	4	1	0	5
TSE	Transactions on Software Engineering	4	0	0	4
EMSE	International Symposium on Empirical Software Engineering and Measurement	1	2	0	3
TOSEM	Transactions on Software Engineering and Methodology	2	0	1	3
SANER	International Conference on Software Analysis, Evolution, and Reengineering	2	1	0	3
SPE	Software-Practice & Experience	3	0	0	3
ICPC	International Conference on Program Comprehension	1	1	0	2
ASEJ	Automated Software Engineering	2	0	0	2
TSC	Transactions on Service Computing	2	0	0	2
JSS	Journal of Systems and Software	2	0	0	2
APSEC	Asia-Pacific Software Engineering Conference	2	0	0	2
WWW	The World Wide Web Conference	2	0	0	2
MLPL	International Workshop on Machine Learning and Programming Languages	2	0	0	2
Access	IEEE Access	2	0	0	2
-	**Total**	**46**	**9**	**1**	**56**

Summary of answers to RQ1:

- *Code search started to be considered in the software engineering research literature in 2002, and its popularity continues to increase with a current peak so far in 2019.*
- *60% of the studies are published in conferences (rather than journals).*
- *67 of the studies propose new code search tools.*

5 RQ2: WHAT ARE THE MOST IMPORTANT FACTORS THAT CONTRIBUTE TO EXISTING CODE SEARCH TOOLS?

We first present an analysis of the modeling and auxiliary techniques used by the reviewed code search tools in Section 5.1 and 5.2 respectively. Section 5.3 then presents a classification of the tools into seven categories and describes how these tools work in general. Section 5.4 investigates how often code search studies provide accessible replication packages.

5.1 Models

For a given query and a codebase, the objective of a code search model is to correctly measure the semantic relevancy between the query and candidate code snippets in the codebase, and retrieve the top-k code according to their relevancy scores. Table 4 shows the main models used in the reviewed code search studies. TF-IDF (Term Frequency-Inverse Document Frequency), BM25 (Best Match 25), and deep learning are the most frequently used modeling techniques in the last three years. To analyze their general features, we classified them into four categories. This includes traditional models (regarding code as text and searching code with IR-based techniques), learning models (leveraging probabilistic model or neural network to learn the relationship between query and code), heuristic models measuring the semantic similarity between query and code by using designed features, and online models (existing online code search tools, e.g., GitHub search).

Table 4. Number of code search models studied in each year from 2002 to 2020 .

Type	Model	02	07	09	10	11	12	13	14	15	16	17	18	19	20	Total
Traditional	TF-IDF	-	-	2	-	-	-	2	-	1	3	1	3	3	2	17
	BM25	-	-	-	-	-	-	-	-	-	-	-	2	5	1	8
	EBM	-	-	-	-	-	-	-	-	1	-	1	-	-	-	2
	SSI	-	-	-	1	-	-	-	-	-	-	-	-	-	-	1
Heuristic	Customized Matching	1	-	1	1	1	-	1	1	2	-	-	3	2	-	13
	Graph Search	-	-	-	-	-	-	-	-	-	2	-	-	-	-	2
Learning	Deep Learning	-	-	-	-	-	-	-	-	-	2	-	1	4	3	10
	fastText	-	-	-	-	-	-	-	-	-	-	-	1	2	-	3
	Probabilistic Model	-	-	-	-	-	-	-	-	1	-	-	-	-	-	1
Online	GitHub Search	-	-	-	-	-	-	-	-	-	1	-	-	2	1	4
	Google Code	-	1	-	1	-	-	-	1	-	-	-	-	-	-	3
	Sourcerer	-	-	-	-	1	-	-	2	-	-	-	-	-	-	3
-	**Total**	1	1	3	3	2	0	3	4	5	8	2	10	18	7	67

Traditional Models. Table 4 shows that 24 code search models leveraged the traditional ranking algorithms TF-IDF [135] and BM25 [105] to measure the relevancy between a query and a candidate code based on the frequency of their shared words [83, 95]. TF-IDF is a simple and effective ranking method. A high TF-IDF score indicates that the query words frequently appeared in the relevant code but rarely occurred in other irrelevant code [10, 85]. BM25 is an improvement of TF-IDF, which restricts the effect of the terms with unexpectedly high frequency under a limited upper bound and balances the term importance for code with different sizes [105].

However, the above models only connect query words with the Boolean operator "OR" implicitly and cannot address the "AND" operation. To support code search with complete Boolean queries, two studies [83, 140] in Table 4 extended query with related words and the operator "AND", and leveraged the Extended Boolean Model (EBM) [113] to calculate the relevancy between query and code. We can notice that all the above models match the words in query and code directly. However, a search query can be described using many different words, and the code may also express a requirement completely different from the search intent. In this case, the direct word matching is prone to failure. To overcome this issue, the Structural Semantic Indexing (SSI) is a traditional choice [106]. SSI represents the codebase as a word-code matrix that records the frequency of a term that occurred at each code. The matrix is reduced via Singular Value Decomposition (SVD) to filter out the noise found in a code so that two code which have the same semantics are located close to one another in a multi-dimensional space [8]. However, a major limit is that the query should be one of the codes in the codebase.

Heuristic Models. Traditional models mainly regard code as text. But code is not just text – it is written in highly structured programming languages with specific keywords, syntactic rules, and semantic representations and meanings [37, 145]. To better express code semantics, Table 4 shows that 15 code search studies developed heuristic models based on researchers' domain knowledge to search code in a more intuitive way. Graph search is one representative method, which represents code as a control flow graph [29] or a call graph [71]. The code search is then transformed into a sub-graph matching issue between query and code. However, most heuristic models search code by designing a customized Matching between query and code [9, 137, 138], such as designed similarity score [13, 137], static analysis [103, 124], and dynamic analysis [21, 122].

Learning Models. The semantic gap between query and code is the major challenging issue for traditional and heuristic code search models. To address this challenge, researchers have built learning models that capture the correlation between query and code from large-scale training data. To reduce the relevancy ranking problem to an application of the probability theory, a probabilistic model [5] computes the relevancy score between a query and a code as the probability that the code will be relevant to the query. This reduces the relevancy ranking problem to an application of the probability theory.

Many other learning models leverage techniques to embed query and code into a shared vector space. The code search problem can then be performed by measuring the cosine similarity between vectors. Table 4 shows that three code search studies adopted the fastText [91], a well-known embedding library based on a shallow neural network succeeded in text classification, for query/code embedding. However, we can see that more code search studies preferred to use deep learning techniques for embedding, including CNN (convolutional neural network) [43, 115], LSTM (long short-term memory) [20, 37, 48, 128, 132, 141, 142], GGNN (gated graph neural network) [128], and FFNN (feed-forward neural network) [30].

Online Search Models. Table 4 shows that ten code search studies built models based on online search engines, including GitHub Search[2], Google Code search, and Sourcerer. These studies focus on how to refine the search results returned by these online search engines.

5.2 Auxiliary Technique

Simply building code search tools with techniques listed in Table 4 may not be enough for practical usage scenarios. Therefore, researchers have also utilized auxiliary techniques to improve the search effectiveness and efficiency. Table 5 presents the auxiliary techniques used in different years.

Inverted Index. Time efficiency is of high importance in code search due to the need to search a large-scale codebase. To accelerate the search response time, 22 code search studies indexed code by leveraging the Lucene tool [87]. Lucene is an efficient text-based search engine, which divides indexing information for any given term into blocks, and builds a parallel structure called a skip list [146] to allow queries to efficiently jump over a set of code that does not match a query. One study also applied R*tree instead of Lucene [63], a multi-dimensional structure for vector indexing [12].

[2] https://github.com/search

Table 5. Number of studies used auxiliary techniques in each year from 2002 to 2020.

Technique	**02**	**07**	**09**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**Total**
Inverted Index	-	-	2	1	-	-	3	-	1	3	1	3	6	2	22
Query Reformulation	-	-	-	-	-	-	-	2	2	1	2	3	6	2	18
Clustering	-	-	1	1	1	-	1	1	1	-	-	2	3	-	11
Learning from User Feedback	-	-	-	-	-	-	-	-	-	-	-	-	1	-	1
Total	0	0	2	2	1	0	4	4	4	4	3	8	16	4	52

Query Reformulation. A free-form text query written in natural language is usually short and misses related contexts. Thus, a code search tool likely returns many irrelevant code snippets for its queries without complete and precise semantics. Therefore, many tools have reformulated developers' queries before performing the code search. This includes techniques such as expanding queries with related words from Stack Overflow [118, 150], extending query words with relevant APIs or class names [83, 143], replacing query words with better synonyms in codebase [65, 67].

Clustering. It is time-consuming for developers to inspect each code snippet returned by a tool one by one. Therefore, researchers have tried to reorganize the results list by clustering similar code snippets [38, 55, 58, 81, 82]. In this case, much developer effort can be saved by only checking the representative code examples. If one developer is interested in one representative, they can check the corresponding cluster later. However, Table 5 shows that only 11 of the reviewed code search tools actually improved the code search results by using such a clustering technique. Thus, it is suggested for further studies to consider results clustering improvement as an important tool component.

Learning from User Feedback. After a search tool returns a list of relevant codes, developers usually check each code one by one and inspect the relevant ones. Developers' feedback on search relevancy can help a tool to identify users' real interests and continuously optimize the tool performance. To reach this goal, researchers have leveraged reinforcement learning to capture developers' preferences [70]. However, it is not easy to obtain such user feedback. This may be the reason why researchers have rarely investigated incorporation of feedback learning into code search tools. This is another promising area for further research.

5.3 Classification of Code Search Tasks

To understand how different code search tools work, the reviewed 67 code search tools and classified them into seven categories, as shown in Table 6: *1) Text-based code search* – searches source code shared with the same semantics as developers' text-based search queries; *2) Code clone search* – uses source code as input and finds similar code from a codebase; *3) I/O example code search* – aims to find code that matches a given input/output example; *4) API-based code search* – finds representative API examples from a codebase according to a given API name; *5) Binary code search* – is similar to the code clone search task but focuses on the binary code (i.e., the compiled source code); *6) UI code search* – retrieves UI implementation code that matches developers' manually sketched UI images; *7) Programming video search* – is a special variant of the text-based code search task but searches code in programming videos.

Table 7 shows the number of studies published from 2002 to 2020 for each of these code search task classifications. We can see that text-based code search is the most popular task with a total of 34 proposed tools and it is also the most frequently investigated task in the recent three years. Moreover, UI code search and the programming video search are two emerging tasks, which require more attention from further studies.

Table 6. Classification of code search tasks.

No.	Code Search Task	Query	Codebases	#Studies	Percent
1	Text-Based Code Search	Text	Source Code	34	51%
2	Code Clone Search	Source Code	Source Code	9	14%
3	I/O Example Code Search	Input/Output Example	Source Code	8	12%
4	API-Based Code Search	API	Source Code	7	11%
5	Binary Clone Search	Binary Code	Binary Code	5	7%
6	UI Code Search	UI Sketch	UI Code	3	4%
7	Programming Video Search	Text	Code in Video	1	1%
-	**Total**	-	-	67	100%

Table 7. Number of code search tasks studied in each year ranging from 2002 to 2020.

Task	02	07	09	10	11	12	13	14	15	16	17	18	19	20	Total
Test-Based Code Search	-	-	1	1	-	-	1	1	3	4	2	6	10	6	34
Code Clone Search	1	-	-	1	-	-	-	-	1	2	-	1	3	-	9
I/O example Code Search	-	1	1	-	1	-	-	2	-	1	-	-	1	1	8
API-Based Code Search	-	-	1	1	1	-	1	-	1	-	-	-	2	-	7
Binary Code Search	-	-	-	-	-	-	1	1	-	1	-	1	1	-	5
UI Code Search	-	-	-	-	-	-	-	-	-	-	-	2	1	-	3
Programming Video Search	-	-	-	-	-	-	-	-	-	-	-	-	-	1	1
Total	1	1	3	3	2	0	3	4	5	8	2	10	18	7	67

5.3.1 *Text-Based Code Search.*

The goal of text-based code search is to retrieve source code from a large-scale code corpus that most closely matches the free-form input by developers [10, 19]. Reusing existing code can largely boost developers' coding efficiency and potentially also quality, by reusing high quality code examples. Text-based code search is frequently studied because this task aims to improve the performance of frequently used code search engines in practice, such as GitHub search. Table 8 presents text-based code search tools with their major and auxiliary modeling techniques. This table also shows how different tools are related to each other in terms of their compared baseline tools.

Traditional Models. Sourcerer [72] is one of the simplest tools. It simply regards code as plain text and ranks candidate found code in the codebase by the classic information retrieval approach TF-IDF (term frequency-inverse document frequency) [39]. A higher TF-IDF score means that the query words frequently appeared in the relevant code but rarely occurred in other irrelevant code. To accelerate the search response time, Sourcerer leveraged the Lucene library [87] to build an inverted index for the large-scale codebase. Finally, for a given query, Sourcerer re-ranks the candidate code according to their popularity within the code dependency graph [133].

To improve code search quality, researchers have tried various approaches. Martie et al. [85] provided code searchers with more advanced choices (e.g., package, class, method, parameters, etc.). Jiang et al. [49] refined the result list by building a supervised ranker to predict the relevancy of a candidate code to a query. Li et al. [70], and leveraged reinforcement learning techniques to learn fromn user feedback – whether a code snippet found is relevant to a search query or not – from developers' search sessions. To mitigate developers' inspection efforts, researchers used code clone detection methods [107, 114] to cluster candidate code and present the representative ones for developers [55, 81]. They have also augmented searched code with contextual information in terms of a chain of methods in the code dependency graph [71, 90].

Table 8. Text-based code search tools.

No.	Year	Tool Name	Baselines
1	2020	CARLCS-CNN [115]	DeepCS
2	2020	QESC2 [150]	QECK, CodeHow
3	2020	Ye20 [142]	DeepCS, CoaCor
4	2020	CDRL [43]	DeepCS, QECK, CodeHow
5	2020	CodeMF [42]	QECK
6	2019	CoaCor [141]	DeepCS, CODE-NN
7	2019	Wu19 [134]	CodeHow
8	2019	MMAN [128]	CodeHow, DeepCS
9	2019	NQE [79]	NCS
10	2019	Cosoch [70]	-
11	2019	QESC1 [46]	QECK, CodeHow
12	2019	QREC [44]	QECK, CodeHow
13	2019	UNIF [19]	NCS, DeepCS
14	2019	GKSR [45]	QECK, CodeHow
15	2019	QESR [52]	QECK, CodeHow
16	2018	GitSearch [118]	-
17	2018	NCS [109]	-
18	2018	CodeNuance [81]	CodeExchange
19	2018	QECC [47]	QECK, CodeHow
20	2018	DeepCS [37]	CodeHow, Sourcerer
21	2018	Codepus [64]	-
22	2017	Zhang17 [143]	-
23	2017	SnippetGen [140]	CodeHow
24	2016	QECK [95]	Portfolio, Keivanloo14
25	2016	RACKS [71]	-
26	2016	ROSF [49]	Portfolio, Keivanloo14
27	2016	CODE-NN [48]	-
28	2015	CodeExchange [85]	-
29	2015	CodeHow [83]	Sourcerer
30	2015	Allamanis15 [5]	-
31	2014	Keivanloo14 [55]	-
32	2013	Portfolio [90]	-
33	2010	Bajracharya10 [8]	Sourcerer
34	2009	Sourcerer [72]	-

However, the major challenge is the semantic gap between query and code. Code is not like a search query and code is written in a highly structured programming language with different syntactic rules and semantic representation [42, 83, 150]. To address this issue, researchers have proposed many query processing techniques to align this semantic gap, such as replacing query words with appropriate synonyms that occur in the codebase [134]; and expanding query words with code changes (e.g., pull requests and commits) in the development history [44–47, 52, 52, 150]. It has been observed that APIs are an important factor to complement the missing semantics in queries [8]. Researchers have thus expanded query words with relevant APIs or class names from official API documents [83], codebases [140], or Stack Overflow posts [11, 42, 95, 118, 143].

Learning Models. To incorporate more domain knowledge into code search tools, researchers have built various learning models. Allamanis et al. [5] explained the code search task as a probabilistic model, namely the probability that a code would be retrieved to match the input query. Other proposed tools score query-code pairs by a trained multiplicative model [92]. This shows that the query and code can be jointly modelled, inspiring other new learning models. Iyer et al. [48] proposed CODE-NN that leverages LSTM to build a translation model from query to code. To train the model, CODE-NN collected posts from Stack Overflow, where the question and corresponding code in the post are used as the training data.

To learn a better representation of code and query, Gu et al. [37] proposed the tool DeepCS that represents code by separate components, including method name, API sequence, and word set in the method body. Their tool embeds query and code into vectors so that code search can be performed by measuring the cosine similarity between vectors. DeepCS is trained by the code and corresponding comments. To further improve the performance of DeepCS, Huang et al. [43] incorporated code/query embedding with an attention mechanism. Shuai et al. [115] leveraged a co-attention mechanism to learn the correlation between the embedded query and code. Wan et al. [128] proposed a tool with more code representations, which embeds the code structure by a tree-based LSTM and the call graph of code by a GGNN (Gate-Graph Neural Network). Recently, researchers [141, 142] improved code search with generated code summarization, and then built tools as a reinforcement learning process of code search and code summarization tasks. The generated summarization is important because it is an abstraction of the code and shares the same semantic level as a developer's search query.

However, the above tools are complex, and training them is time-consuming. Therefore, researchers also investigated more lightweight tool approaches at the same time. Sachdev et al. [109] proposed a tool, NCS, that leveraged an unsupervised token-level embedding fastText [16] to transform code and query into vectors. NCS searched candidate code for a query by using the TF-IDF weighting method [39], and finally re-ranked the candidates by comparing their cosine similarity to the query vector. To improve the search effectiveness of NCS, Liu et al. [79] added a query expansion technique to the NCS; Sachdev et al. [109] replaced the unsupervised component TF-IDF in NCS by a neural network with an attention mechanism. This can be trained by the code-comment pairs as DeepCS [37].

5.3.2 Code Clone Search. A code clone search tool takes a piece of code as a query and returns a list of similar codes [58, 82, 100]. It differs from code clone detection [107, 114] because it is query-centric, retrieves only clones that are associate with the query, instead of looking for a complete set of clone pairs in the codebase as the clone detection, and cares about the tool scalability [100]. Table 13 shows the reviewed studies related to the code clone search task.

Early tools regarded code clone search as a token-by-token matching issue. CCFinder [53] identifies whether the partial token sequences of a candidate code snippet contains the target query code. SourcererCC [112] calculated code similarity based on the overlap degree between the tokens of two codes. When the degree value is lower than a pre-defined coefficient, SourcererCC returns the code in a codebase as a clone. However, code does not just consist of tokens but with particular structures. Thus, further tools considered this code structural information. Lee et al. [63] transformed a source code snippet into a control flow graph (CFG). The graph is represented by a characteristic vector via the locality sensitive hashing (LSH) algorithm [36, 50], where a node in CFG is composed by its subgraphs. Code clone search is then regarded as a subgraph matching problem. Balachandran [9] improved the LSH representation method by incorporating the more code structural feature, namely the abstract syntax trees (ASTs).

In recent years, researchers have built tools based on learning models. DLC [132] leveraged a deep learning technique to embed binary code into vectors and learned their lexical relationship. Code clone search can then be performed by measuring the cosine similarity between code vectors. Additionally, to learn the higher semantic difference between code, TBCAA [20] applies the tree-based convolution neural network to capture the structural feature, namely the AST (abstract syntax tree) of code. Researchers have also sought other ways to improve code clone search. Siamese [100] incorporates a multi-representation, corresponding to four clone types, to represent indexed corpus of code, improves the query quality by leveraging the knowledge of token frequency in the codebase, and finally re-ranks the searched candidate code based on the TF-IDF weighting method. FaCoY

[58] extended the query with related code in Stack Overflow, and searches similar code fragments against the code index built from the source code of software projects. Aroma [82] pruned and clustered candidate code, and intersects the snippets in each cluster to carve out a maximal code snippet. This snippet is common to all the snippets in the cluster and which contains the query snippet. The set of intersected code snippets are then returned as recommended code snippets.

Table 9. Code clone search tools.

No.	Year	Tool Name	Baselines
1	2019	TBCAA [20]	Siamese, SourcererCC, CCFinder, DLC
2	2019	Siamese [100]	FaCoy, SourcererCC, CCFinder
3	2019	Aroma [82]	SourcererCC
4	2018	FaCoY [58]	SourcererCC, CCFinder
5	2016	DLC [132]	-
6	2016	SourcererCC [112]	CCFinder
7	2015	Balachandran15 [9]	-
8	2010	Lee10 [63]	-
9	2002	CCFinder [53]	-

5.3.3 I/O Example Code Search. For I/O example code search, a query contains a set of examples specifying the desired input/output (I/O) behaviors of target code [21]. The given I/O examples reflect incomplete functional specifications that can be collected from development requirements [21] or test cases [66]. An I/O example code search tool aims to find the code methods that match a specified I/O example. How the code behaves is unimportant [21]. Table 11 shows I/O example code search tools included in this systematic review. We can see that these tools have mainly refined the results of existing online search engines, such as GitHub search, Google Code, and Sourcerer.

To find a method that matches the expected I/O examples, early tools [103, 111, 124] employed graph-based code mining algorithms to mine paths that start with the input example and end with the output example. However, it was observed although some methods did not satisfy the I/O requirement, their partial code meets the expectation. Therefore, researchers [66, 123] leveraged slicing techniques that locate the output example from a method and extract related code snippets backwards. Such a technique excludes the methods that cannot trace the input example. To improve search performance, researchers also leveraged query processing techniques to optimize the initial results of online search engines, namely expanding query with appropriate synonyms [65, 67].

Dynamic analysis techniques have also been adopted for I/O example code search. Stolee et al. [122] proposed a tool called Satsy. It is based on a symbolic execution approach and works in two phases. During an offline encoding phase, Satsy encodes the semantics of code in codebase into logical formulas concerning their input/output variables. During an online search, Satsy binds concrete values from I/O examples to compatible variables in each formula to construct a constraint, and checks the satisfiability of the constraint using a solver Z3 [28]. Although Satsy has been applied to search for Java code in previous studies [122], its usefulness in daily code search activities is limited, as it handles only loop-free code snippets manipulating data of char, int, boolean, and String types. To extend the usefulness of Satsy, Chen et al. [21] proposed a tool Quebio. Different from Satsy, its symbolic encoding phase supports more language features like the invocation to library APIs, which enables Quebio to handle more data types (e.g., array, List, Set, and Map) during the search. This new feature enables Quebio to be used in a wider range of scenarios.

Table 10. I/O example code search tools.

No.	Year	Tool Name	Baselines
1	2020	Quebio [21]	Satsy
2	2019	TIRSnippet [123]	PARSEWeb
3	2016	Satsy [122]	-
4	2014	Lemos14 [67]	QE$_{wct}$
5	2014	QE$_{wct}$ [65]	CodeGenie
6	2011	CodeGenie [66]	-
7	2009	S6 [103]	-
8	2007	PARSEWeb [124]	-

5.3.4 API-Based Code Search. Developers write code using various APIs, but only a limited portion is explained with code examples, where only around 2% of APIs in JDK 5 (27k in total) provide examples [130, 149]. Therefore, developers have to type the expected API in existing search engines, such as Google. However, the search engine often returns numerous results and most of them do not meet developers' expectations [38, 144]. Thus, many tools have been proposed to mitigate developers' inspection efforts by clustering and ranking candidate code [57, 84, 130].

One early example is MAPO [149] that searches a list of code relevant to the target API and clusters the code according to their API call sequences using a classical hierarchical clustering technique [40]. To help developers find expected code quickly, MAPO also generates code call patterns (i.e., a sequence of API calls) for describing each cluster. As the major challenge is how to cluster and rank the searched code, researchers have proposed a number of solutions. EXoaDocs [57] transforms the searched code into a vector space according to their AST structure, clusters them by using a hierarchical clustering algorithm (the centroid of a cluster is regarded as the representative code), and ranks the code based on three factors. These are representativeness – the reciprocal of the similarity to the representative code of the corresponding cluster; conciseness – the reciprocal of code length; and correctness – the degree the code is related to the target API. PropER-Doc [84] clusters candidate code based on their interacted API types, and ranks the candidates based on three designed metrics: significance, how the API in code related to the query; density, the portion of code lines that refers to the query; cohesiveness, the aggregation level of the query described within the code. UPMiner [130] clusters code based on the similarity of the API sequences and groups the frequent closed sequences into a code pattern for a cluster using the tool BIDE [131]. The code pattern that covers more possible APIs and contains fewer redundant lines is ranked higher. MUSE [62] discards irrelevant lines of code in the codebase, clusters the simplified code by a code clone detection method [41], and ranks the representative code in clusters based on their reusability, understandability, and popularity.

ADECK [144] collects candidate code examples from the community question and answer (Q&A) forum Stack Overflow. It represents the searched post as a tuple consisted of the post title and the best-answered code scored by users. Then ADECK clusters the tuples based their semantic similarities by leveraging the APCluster method [32]. KodeKernel [38] represents a source code as an object usage graph, instead of method invocation sequences or feature vectors. It clusters code by embedding them into a continuous space using a graph kernel. KodeKernel then selects a representative code from each cluster based on two designed ranking metrics: centrality, the average distance from one code to another in the cluster; and specificity, the code contains less rarely appeared lines.

5.3.5 Binary Code Search. When deploying source code on different operating systems with various compilers and optimization methods, source code is usually transformed into different binary codes. For a given binary code, how to search for the same binary code but compiled in other forms becomes the objective of the binary code

Table 11. API-based code search tools.

No.	Year	Tool Name	Baselines
1	2019	KodeKernel [38]	eXoaDocs, MUSE
2	2019	ADECK [144]	eXoaDocs
3	2015	MUSE [62]	-
4	2013	UPMiner [130]	MAPO
5	2011	PropER-Doc [84]	-
6	2010	eXoaDocs [57]	-
7	2009	MAPO [149]	-

search task. Assembly code (i.e., the human-readable binary code) is commonly used to build the codebase [30, 56]. This is important for plagiarism detection, malware detection, and software vulnerability auditing [27, 56].

To address the binary code search task, Rendezvous [56] compares the descriptive statistics between code tokens in terms of the mnemonic n-grams, mnemonic n-perms, control flow sub-graph, and data constants. To ensure tool scalability, Rendezvous builds indexing for a codebase to reduce the scope of search space according to the given query. However, the low-level compiler transformations can strongly affect the performance of Rendezvous. To tackle this issue, Tracy [27] decomposes code into tracelets – the continuous, short, partial traces of an execution – and compares these tracelets based on a Jaccard containment similarity [3] in the face of low-level compiler transformations. Kam1n0 [29] represents the control flow graph (CFG) of binary code as a LSH (locality sensitive hashing) scheme [60], where a node in CFG is formed as a combination of its subgraph. Binary code search is then transformed into a subgraph search problem. It is challenging to align the semantics between two binary codes. To overcome this challenge, BingGo-E [138] selectively inlines a binary code with relevant libraries and user-defined codes to complete the semantics in code. Asm2Vec [30] leverages a deep learning technique to jointly learn the semantic relationships between binary code. The learned code representation can largely mitigate the manual incorporation of the complex prior domain knowledge.

Table 12. Binary clone search tools.

No.	Year	Tool Name	Baselines
1	2019	Asm2Vec [30]	Rendezvous
2	2018	BingGo-E [138]	Tracy
3	2016	Kam1n0 [29]	Rendezvous, Tracy
4	2014	Tracy [27]	Rendezvous
5	2013	Rendezvous [56]	-

5.3.6 UI Code Search. When developing software user interfaces (UIs) developers commonly draft UI sketches and implement corresponding UI code with related APIs. This often takes enormous efforts. UI code search tools take UI sketches as a query and search for UI code snippets that match the requirement of the UI sketch from a codebase. Table 13 shows three UI code search tools we reviewed in this study. Reiss et al. [104] converted the image of a developer's UI sketch into a scalable vector graphic (SVG) diagram, and reduced the search space by using an existing text-based search engine S^6 [51] with related keywords. After a series of transformations for the candidate code, this tool returns the ones that can be compiled and run. Behrang et al. [13] proposed a similar tool GUIFetch but measures the query-code relevancy with a comprehensive metric based on the screen similarity in terms of the screen type, size, and position, and the screen transition similarity. To improve query representation, Xie et al. [137] adapted the pix2code [14] tool to automatically extract the code structure from a UI sketch. Pix2code uses a DL model that can capture the UI components types and their hierarchical relationships, which was trained with manually labeled UI sketches. To sort candidate UI code, Xie et al. [137] measured the layout distance between query and candidate code based on the Levenshtein distance [68].

Table 13. UI code search tools.

No.	Year	Tool Name	Baselines
1	2019	Xie19 [137]	Reiss18
2	2018	GUIFetch [13]	-
3	2018	Reiss18 [104]	-

5.3.7 Programming Video Search. Programming video code snippet search is a new code search task recently investigated by Bao et al. [10]. This task aims to search for relevant code snippets in programming videos (e.g., from YouTube) using text-based queries. The major challenge is how to capture the code in videos and transform it into text. Code search can then be implemented by using text-based code search tools described earlier in Section 5.3.1. To capture the relevant code frame in programming videos, Bao et al. [10] proposed a tool psc2code. It removes noisy frames by a CNN (Convolutional Neural Network) based image classification, extracts source code by calling a professional ORC (Optical Character Recognition) tool, and performs code search by using the TF-IDF algorithm.

5.4 Replication Packages

We wondered how often the reviewed 67 code search tools share replication packages in their papers. We searched and inspected all the links in each paper. If a replication package link is available, we checked the link accessibility and whether the replication package contains source code. If we found no relevant link in a paper, we also searched its replication package in GitHub with the paper title. The pie chart in Fig. 4 shows that only 18% of the reviewed studies provide accessible replication packages. Among the other studies, 14 studies provide inaccessible links in the paper or accessible links without source code. We found no description of replication packages for 41 studies. To facilitate future code search study, we provide the usable replication packages in Table 14.

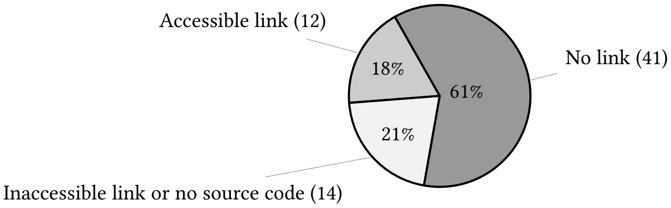


Fig. 4. Replication package.

Table 14. Replication package links.

Task	Tool	URL
Text-Based Code Search	CARLCS-CNN CoaCor DeepCS CODE-NN	https://github.com/cqu-isse/CARLCS-CNN https://github.com/LittleYUYU/CoaCor https://github.com/guxd/deep-code-search https://github.com/sriniiyer/codenn
I/O example Code Search	MUSE	https://github.com/lmorenoc/icse15-muse-appendix
API-Based Code Search	-	-
Code Clone Search	Siamese Aroma FaCoY	https://github.com/UCL-CREST/Siamese https://github.com/facebookresearch/aroma-paper-artifacts https://github.com/FalconLK/facoy
Binary Clone Search	Kam1n0 Tracy	https://github.com/McGill-DMaS/Kam1n0-Community https://github.com/Yanivmd/TRACY
UI Search	Xie19	https://github.com/yingtao-xie/code_retrieval/
Programming Video Search	psc2code	https://github.com/baolingfeng/psc2code

Summary of answers to RQ2:

- *74% of tools searched source code with queries written in natural language (i.e., text, API name, input/output example).*
- *Deep learning is the most popular modeling technique in the last two years.*
- *Inverted Index was frequently used for accelerating code search efficiency; researchers also leveraged other auxiliary techniques (i.e., query reformulation, code clustering, active learning) to improve the search accuracy.*
- *We found only 12 code search studies shared accessible replication package links in their papers or provided source code in GitHub.*

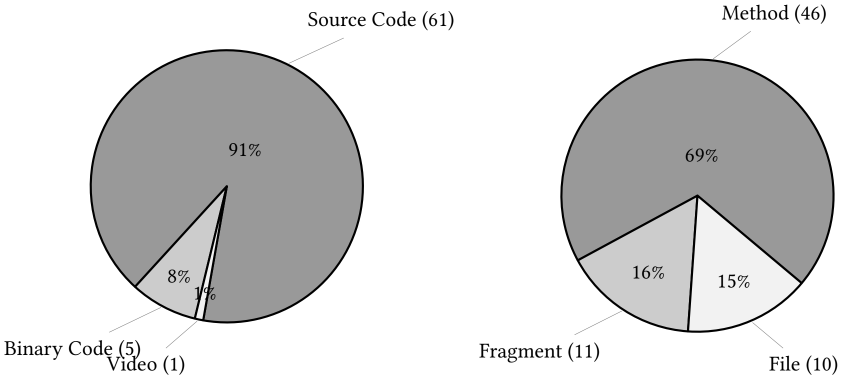
6 RQ3: HOW DO STUDIES EVALUATE CODE SEARCH TOOLS?

We investigated the key approaches used to evaluate the 67 code search tools including aspects of the codebase, query, evaluation method, and performance measures.

6.1 Codebase

Various characteristics of the codebase define the search space of a specific code search type. After inspecting the 67 code search tools, we found three types of codebases. As illustrated in Fig. 5(a), 91% of codebases are built with source code written by developers in high-level programming languages (e.g., Java); 8% of codebases consist of binary code, i.e., the compiled source code; and one study collected a corpus of programming tutorial video with source code as a codebase to help developers search code from videos [10]. Fig. 5(b) shows the distribution of the code granularity: 69% of the tools regard methods as search targets; 16% of the tools focus on recovering code fragments; and 15% of the tools concentrate on retrieving relevant files, e.g. the app UI code [104].

Fig. 6 (a) shows the distribution of programming language in code bases: Java is the most favored language with 52 studies, followed by C#, Assembly, C/C++, SQL, Python, and Javascript. Fig. 6(b) shows the distribution



(a) Codebase types.

(b) Codebase granularity.

Fig. 5. Codebase types and granularity.

of the codebase scale in terms of the number of studied instances (i.e., method, fragment, or file). We can see that 35 of the selected studies used small scale codebases with more than 1k and lower than 1m code items. 22 studies constructed larger scale codebases with more than one million code items. However, the codebase scales of ten studies are unknown because their tools are based on online search engines, such as the GitHub search[3] with millions of open source repositories. Table 15 shows the codebase scale in different code granularities (method, fragment, and file) respectively. We can see that most studies that search code methods have built larger codebases (>10m).

Table 16 summarizes the data sources of different codebases. By classifying the sources into four categories, we can see that the open source community is the biggest source category including GitHub [37], SourceForge [149], Google code [103], Apache [64], FDroid [49], OpenHub [71], and Tigris.org [63]. Among these communities, GitHub is the most popular one related to 23 code search studies. Researchers also collected data from app stores (Google Play and Apple store [137]), enterprise with closed projects (Microsoft [130] and Amazon [71]), and programming forum and videos (Stack Overflow [5, 144] and YouTube [10]). 19% of studies manually selected and downloaded some projects according to their experience [8, 90], while 13% of studies proposed new techniques to better support online search engines (e.g., GitHub search [38] and Google code [84]).

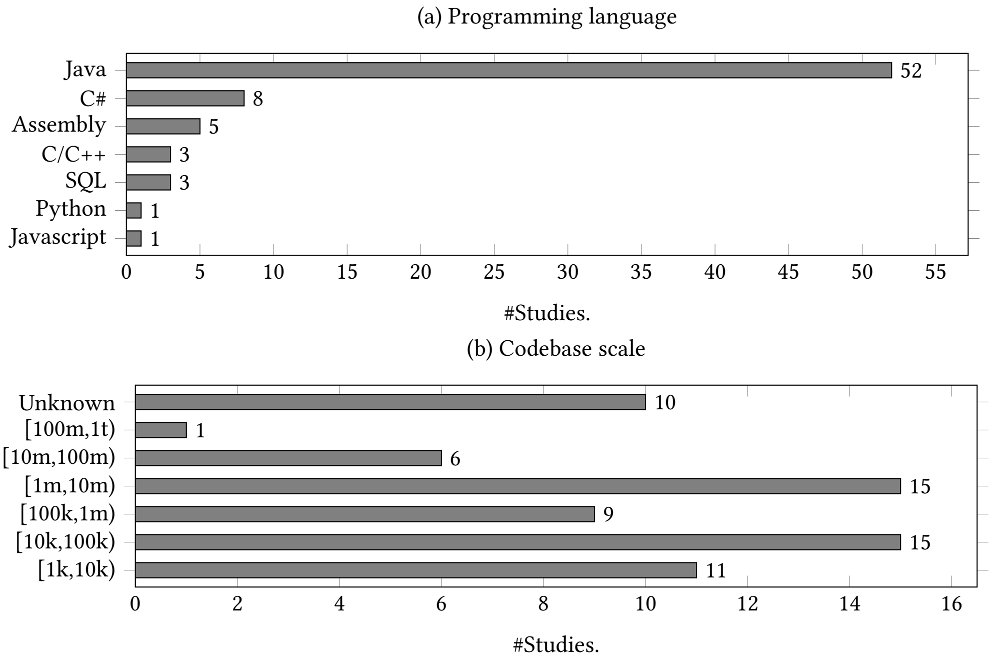
Table 15. Codebase scale in different code granularities (i.e., method, fragment, and file).

Scale	**Unknown**	**[100m,1t)**	**[10m,100m)**	**[1m,10m)**	**[100k,1m)**	**[10k,100k)**	**[1k,10k)**	**Total**
Method	7	1	6	7	6	12	7	46
Fragment	0	0	0	6	2	1	2	11
File	3	0	0	2	1	2	2	10
Total	10	1	6	15	9	15	11	67

6.2 Queries

Code search queries reflect developers' search requirements, and their features determine how a code search tool can support developers' intents. Fig. 7(a) shows that there are seven types of queries: *1) text* – a short string of

[3] https://github.com/search



(a) Programming language

(b) Codebase scale

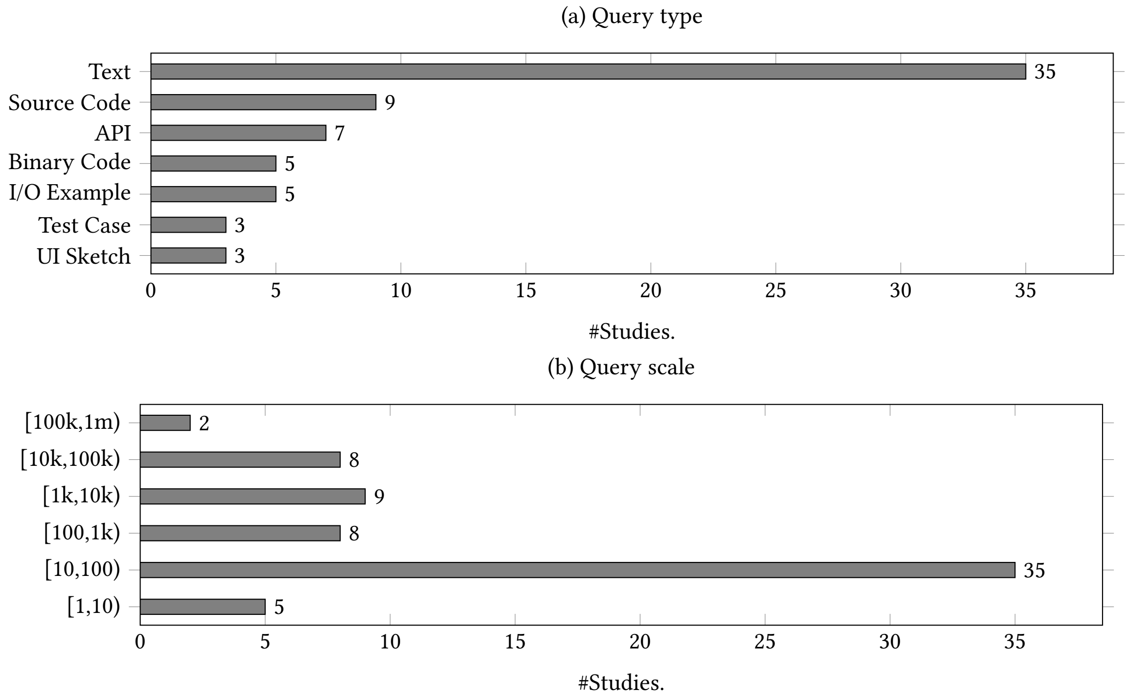
Fig. 6. Codebase language and scale.

Table 16. Codebase source.

Source Type	Source	#Studies	Percent
Public Code Repositories	GitHub	23	29%
	SourceForge	6	8%
	Google Code	4	5%
	Apache	3	4%
	FDroid	2	3%
	OpenHub	1	1%
	Tigris.org	1	1%
App Store	Google Play	1	1%
	Apple Store	1	1%
Internal Code Repositories	Microsoft	1	1%
	Amazon	1	1%
Forum and Video Sharing Platform	Stack Overflow	8	10%
	YouTube	1	1%
Other	Selected Projects	15	19%
	Online Search Engine	10	13%

text written in natural language, e.g., "how to convert inputstream to a string"; *2) source code* – a code method or fragment to retrieve similar code from codebase; *3) API* – an official or third-party API name to search the API usage example; *4) binary code* – an assembly code (i.e., the compiled source code) with similar search task as the query of source code; *5) I/O Example* – the input and output variable types or examples for a code method; *6) Test case* – a piece of testing code; and *7) UI sketch* – an image of UI skeleton drafted by UI designers. Among these query choices, the text query can be supported by 35 code search studies. The query-based search is popular because it is be regarded as a general search engine like the GitHub search.

Fig. 7(b) illustrates the distribution of query scales in the reviewed 67 code search tools. We can see that 35 studies evaluated tool effectiveness with 10-100 queries. Five studies tested tools with no more than ten queries, while 25 studies performed code search with larger scale of inputs of more than 100 queries.


(a) Query type

(b) Query scale

Fig. 7. Query type and scale.

We also analyzed the query sources adopted by different code search studies. As illustrated in Table 17, 39% of queries were collected from question and answer forums including Stack Overflow [19] and Eclipse FAQs [8]; 6% of queries were extracted from software development kits, including JDK [38], Android development kit [144], and .Net framework [130]. The remaining 55% of queries were manually selected from frequently used examples: codebase [66, 124], search logs of practically used code search engine [85, 140], a summary of search on the Internet [90, 104], and automatically generated queries [47, 150].

6.3 Evaluation Methods

The reviewed code search studies have evaluated new tools in four ways, as listed in Table 18. Manual identification is the major evaluation method in 37 related studies. In this case, researchers or invited developers manually inspected the top-*n* result list returned by a code search tool and identified their relevancy to query intent one by one. However, such manual evaluation approaches may suffer from subjective bias. To mitigate this issue, 30 code search studies provided three approaches to automatically identify the relevancy between queries and searched

Table 17. Query source.

Type	Source	#Studies	Percent
Question and Answer Forum	Stack Overflow	25	36%
	Eclipse FAQs	2	3%
Development Kit	Java Development Kit	3	4%
	Android Development Kit	1	1%
	.NET Framework	1	1%
Frequently Used Examples	Codebase	30	43%
	Search Log	4	6%
	Web Search	2	3%
	Automatic Generation	2	3%

results. 19 studies constructed a ground-truth where selected code snippets correspond to each query result in advance. However, obtaining the ground-truth is not easy and is very manually intensive to do. Therefore, two studies [38, 137] asked researchers or developers to manually annotate the ground-truth in the codebase, which requires the codebase scale to be amenable to these limited manual efforts. To mitigate issues when using manual efforts, nine studies designed a measurement to score the query-code relevancy (e.g., leveraging a clone detection method to score the similarity between a search code and an example code [46, 109]) and determined relevancy if the score is larger than a pre-defined threshold. However, choosing the right threshold is difficult and researchers have tried their best to simulate manual identification.

Table 18. Evaluation method.

No.	Method	Description	Count	Percent
1	Manual	Manual identification	37	55%
2	Automated1	Automated identification based on collected query-code pairs	19	28%
3	Automated2	Automated identification based on manually annotated query-code relevancy	2	3%
4	Automated3	Automated identification based on a designed algorithm to judge the query-code relevancy	9	14%
-	**Total**	-	67	100%

6.4 Performance Metrics

To measure the effectiveness of a code search task, studies used various performance metrics as listed in Table 19. Generally, there are two types of accuracy measure approaches – one treating code search as a ranking issue and the other treating code search as a classification issue.

For ranking, for a search result list with top-*k* code, a metric measures the accuracy concerning the important location in the list [37, 74, 115]. FRank@k and FFP@k are metrics for an individual query, which are the ranks of the first relevant code and the first false positive code respectively. To measure the comprehensive search accuracy for all queries, MRR@k and SuccessRate@k care about the first correctly search code for a query, where MRR@k is an average of reciprocal ranks of the first correct search for each query; SuccessRate@k counts the percentage of queries that retrieved at least one relevant code. To measure the accuracy concerning multiple relevant codes for a query, researchers adopted Precision@k, MAP@k, and NDCG@k, where Precision@k equals to the percentage of relevant code in the top-k result list; MAP@k considers Precision@k with k ranging from 1

Table 19. Performance Metrics.

Type	Metric	Description	#Studies
Ranking	Precision@k	The percentage of relevant code in the top-k result list for each query.	27
	MRR@k	The average of reciprocal ranks of the results of a set of queries, where the reciprocal rank of a query is the inverse rank of first relevant code in the top-k result list.	19
	NDCG@k	The normalized discounted cumulative gain for the top-k search results.	16
	SuccessRate@k	The proportion of queries that the relevant code could be found in the top-k result lists.	11
	MAP@k	Mean average precision, where the average precision of a query is the mean of precision at each rank.	5
	FRank@k	The rank of the first relevant code in the top-k result list for a given query.	4
	FFP@k	The rank of the first false positive code in the top-k results for a given query.	1
	MSE@k	The mean squared errors between the relevancy ratings of each search code and the ground-truth ratings.	1
	KCC@k	The Kendall's correlation coefficient between the searched code rankings and the ground-truth rankings.	1
	SCC@k	The Spearman's correlation coefficient between the searched code rankings and the ground-truth rankings.	1
	PCC@k	The Pearson correlation coefficient between the searched code rankings and the ground-truth rankings.	1
Classification	Precision	The proportion of code in a result list are truly relevant for a query.	16
	Recall	The proportion of relevant code in codebase that are correctly retrieved for a query.	9
	F1-score	A harmonic mean of Precision (p) and Recall (r), equaling to $2pr/(p+r)$.	4
	AUROC	The area under the receiver operating characteristics curve.	2
	F2-score	A weighted harmonic mean of Precision (p) and Recall (r) with double weights on recall, equaling to $5pr/(4p+r)$.	1
	AUPR	The area under the precision-recall curve.	1
Other	Search Time	The average time duration for each code search query to be processed.	13

to k; NDCG@k is also a position sensitive metric similar to MAP@k but implemented in terms of the discounted cumulative gain.

The previous metrics are based on code-query relevancy status (i.e., relevant or not). In contrast, we found that one study measures code search tool performance based on relevancy scores like MSE@k, the mean squared errors between the relevancy ratings by a tool, and the ratings by researchers or developers. Similarly, the correlation test between two ratings lists can also be used as a metric, including the KCC@k (Kendall's correlation coefficient [1]), SCC@k (Spearman correlation coefficient [93]), and PCC@k (Pearson correlation coefficient [15]).

Some studies regard code search as a classification task. Different from treat it as a ranking task, classification ignores the position of located code snippets for a given query and aims to retrieve as many relevant code snippets as possible [20, 29, 63]. For a query, a code search tool searches all relevant code from the codebase without size limitation. Its performance can then be measured by Precision (the proportion of code in a result list is truly relevant for a query) and Recall (the proportion of relevant code in the codebase that is correctly retrieved for a query).

However, the same tool usually cannot usually achieve both high Precision and high Recall. To solve this issue, studies have also used summary metrics: F1-score, the harmonic mean of Precision and Recall [63, 75, 76]; F2-score, a variant of F1-score that puts more weights on Precision [56]; AUROC, the area under the receiver operating characteristics curve [72]; and AUPR, the area under the precision-recall curve [29]. Despite the accuracy measures, 13 studies also considered the tool efficiency in terms of the code search time [112, 132].

Summary of answers to RQ3:

- *Most reviewed study codebases were built with large-scale method-level source code written in Java, which were collected from public code repositories (e.g., GitHub and FDroid).*
- *Most code search studies have tested their proposed tools with top-n frequently used text-based queries collected from Q&A forums (e.g., Stack Overflow).*
- *Performance of 55% of code search tools were estimated using manual analysis.*
- *Most of the reviewed studies assessed tool performance with ranking metrics (e.g., MRR and NDCG).*

7 CHALLENGES AND OPPORTUNITIES

7.1 Challenges

Challenge 1: Diversity of the Codebase. The characteristics of a codebase determine what a tool can find in the search space. However, most researchers have built their codebases in different ways that may affect the tool performance and usability:

1) Small Scale. The codebase scale in a practical environment (e.g., GitHub and FDroid) is usually large with millions of lines of code. However, many code search studies only tested their tools on a small scale codebase [43, 70, 130, 140]. In this case, findings may not generalize to large codebases [70]. Moreover, for a small-scale codebase, a code search tool may not work just because the codebase contains no code relevant to some search queries.

2) Language Specific. As illustrated in Fig. 6, most codebases only focused on code written in one type of programming language, especially Java. However, developers write and search code in various programming languages (e.g., Python and C#). Although some studies claimed that their tools can be easily extended to other languages, they only tested their tools on a codebase for one specific language [43, 123, 128].

Challenge 2: Limited Queries. To test the tool effectiveness, studies carefully collected queries from Q&A forums, development kits, and frequently used examples as listed in Table 17. The studies tried their best to find appropriate queries to simulate developers' search behaviors in the real world. However, the selected queries are limited in four aspects:

1) Limited Quantity. Fig. 7(b) shows that nearly 60% of studies tested their proposed tools by using no more than 100 queries. Such query scale can only cover a limited number of real-world queries actually used by software developers [42, 43, 118]. The main reason that hinders the query scale is the tool evaluation method. Table 18 shows that 55% of the code search tools were verified by manual evaluation only. Increasing the query scale would substantially increase the burden of user study participants during their labelling efforts.

2) Query Representativeness. Commonly, code search studies selected the top-*n* frequently used queries from Q&A forums, or randomly sample a small set of queries from the real world [42, 43, 118]. However, studies seldom investigated the representativeness of the selected queries. Thus, it would be uncertain and questionable if a code search tool can work for other types of queries. Moreover, the selected queries are usually too general and in reality developers often search for very domain-specific code [130]. Therefore, it is necessary to analyze the distribution and characteristics of queries to verify the tool generalizability.

3) English Query Only. For text-based queries, code search studies have nearly always only investigated the queries written in English. However, developers are scattered all around the world and use different languages, not just English [118]. Therefore, it is beneficial and necessary to make an extension to non-English queries especially for text-based search tools. It is also necessary to investigate how developers with more limited English can use English-based code search tools.

Challenge 3: Model Construction Issues. Learning models are popular and favored in recent years. This is because learning models (e.g., deep learning models) require no substantial manual efforts in incorporating domain knowledge into the traditional and heuristic models. However, existing learning models possess several threats to their model validity:

1) Validity of Parameters. DL models involve many internal parameters, such as batch size, stop condition, learning rate, etc. Code search studies have usually initialized these parameters with default settings and do not verify the effectiveness of this parameter choice [44, 45, 95]. They have also sometimes tuned the parameters without explaining the reasons or validity. Such unverified parameters may threaten the model generalizability [73].

2) Quality and Quantity of Training Data. For text-based code search, DL models were trained with pairs of code and comments. The comment is a replacement for the query. Nevertheless, developers wrote queries in different styles and languages. Noisy comments also likely affect model performance. Thus, the quality of this training data may adversely threaten the trained model effectiveness [115, 128, 141]. Moreover, only a few codes contain comments so that a model trained with commented code may not work for other code [74].

Challenge 4: Evaluation Issues. We observed that tool evaluation is the most widely discussed future work issue in the reviewed code search studies. Based on their discussions and our findings, we attributed this evaluation issue to two aspects:

1) Relevancy Identification. Table 18 shows that only 28% of code search tools were evaluated by automated identification with a carefully curated ground-truth. This is because a ground-truth is difficult to construct for most code search tasks. Although nine code search studies identified the query-code relevancy by designed measurements, it is uncertain if such measurements are actually reasonable. Due to the above difficulties, most code search studies chose to identify relevancy with human efforts. However, their evaluation with manual identification is subject to potential serious bias and human errors [37, 42, 73, 81, 128].

2) Dataset Configuration. For a tool based on a learning model, studies commonly split the codebase into three parts (i.e., training, validating, and testing data) with different ratios (e.g., 8:1:1) [20, 45]. This is a common setting for tool verification and the split can avoid the overfitting issue [37]. However, this setting substantially reduces the scale of the codebase in tool testing. In a real-world code search scenario, the search space usually contains millions of repositories with limited number of training data. The testing data scale is also increasing continuously. Therefore, to better simulate practical search cases, it is suggested to split the codebase with less training data and more testing data.

Challenge 5: Limited Performance Measures. Although researchers used various performance metrics to measure the tool performance, as listed in Table 19, the adopted metrics are not enough to meet the requirements for code search evaluation. We observed that current code search tools lack the following considerations:

1) Tool Efficiency and Scalability. Searching for relevant code from a large-scale codebase is one feature of the code search task. Therefore, developers expect that the used tool performs code search as fast as possible. However, most code search studies did not estimate the tool performance in terms of the tool search time, as illustrated in Table 19. Especially for tools based on learning models, the search time is commonly not acceptable due to the high model complexity [33, 74]. Moreover, as the codebase is frequently updated by developers, it is also necessary to estimate the scalability and reproducibility of the proposed tool on codebases of different scales.

2) Other Important Metrics. To estimate the performance of code search tools, the accuracy (e.g., MRR and NDCG) and efficiency (e.g., search time) metrics are not enough. Some code search studies (e.g., text-based search, API-based search, and exampled search) just return a list of relevant code to a search query, but ignore the diversity of the list of returned code without excluding repeated results [86]. Some returned code snippets are not concise with many lines irrelevant to the actual query intent. This kind of code can not be reused easily by developers in their real-world software development scenarios [144]. Therefore, it is important to consider the readability and reusability of the searched code [62].

Challenge 6: Replication Issues. Tool reproducibility is of high merit in code search research, as other researchers or practitioners require less effort in replicating / extending / comparing to the study. We observed that existing code search tools suffer from the following replication issues:

1) Replication Package. Sharing source code and dataset can greatly help to support the tool replicability and mitigate researchers' replication efforts. However, Fig. 4 shows that only 18% of code search tools provide accessible replication packages [73, 118]. It is recommended that future studies share their contributions publicly [73]. Some researchers may share their code upon email request, but it is highly recommended to provide the accessible replication package links in the papers and to maintain these repositories.

2) Online Search Engine. Table 4 shows that ten code search tools depend on an online search engine (e.g., GitHub search, Google Code search, and Sourcerer). However, the online search engines could be improved through time, so that new studies cannot replicate the experimental results reported by early studies. To facilitate the tool replication, further studies are encouraged to provide the access date of their used online search engines. Besides, if there exists source code or paper for an online engine (e.g., Sourcerer), it is necessary to check their differences. In this way, when researchers found the performance of their re-implemented code search tool is different from the performance reported in a paper, they could understand whether the difference comes from the improvement of the online search engine or their re-implementation errors.

7.2 Opportunities

Opportunity 1: Better Benchmarks. One urgent task for code search research is to build a standard benchmark that different tools can be evaluated with. Such a benchmark requires that: the codebase is industrial-scale with millions of lines of code and multiple programming languages [43, 128]; the tested search queries involve various diverse types of real-world examples covering not just top-*n* frequently used general queries; developing a standard automated tool evaluation method to exclude manual efforts and subjective bias; and excluding repeated code before assessing the tool performance.

Opportunity 2: DL-Based Model with Big Data. It is promising to build DL-based code search models trained with big data to learn the correlations between queries and code. There are some opportunities to improve the DL-based models such as: standardizing code written by different developers with various programming styles and experiences to mitigate training difficulties; improving the quality of the training data; developing better code representation methods to capture the programming semantics [115, 145]; leveraging better loss functions to optimize the DL-based model, e.g., using the ranking loss function [77, 78, 115, 142], and increasing size of the training and testing data sets.

Opportunity 3: Fusion of Different Types of Models. Researchers have developed various code search tools. Although learning models have shown promising advantages over traditional and heuristic models, their disadvantages are still obvious, such as slow code search time, low scalability and need for periodic retraining. However such efficiency issues can be complemented by the other two types of models. Therefore, it is suggested to explore fusing the advantages of different types of code search models into tools [74].

Opportunity 4: Multi-Language Tool. It is difficult to apply an existing tool designed for a specific programming language to searching code written in other programming languages. This is because tools often depend on language-specific features and parsing processes. Moreover, a learning model is likely to cost a long time to retrain data for a new programming language. Therefore, it is recommended to develop a multi-language code search tool. For example, leveraging the multi-task learning techniques [147, 148] to capture the semantics of multi-language in code search, and investigating whether a model trained on one language can be transferred to other language [96, 127]. In this way, code search practitioners do not have to train and deploy code search tools on each programming language one by one.

Opportunity 5: New Code Search Tasks. UI code search [13, 104, 137] and programming video search [10] are two emerging code search tasks. Different from the traditional popular code search tasks using text-based code search, they extended the capability of existing code search engines to use UI sketches as queries and videos as sources for useful code snippets. Specifically, the UI code search can provide more relevant help for UI developers, while programming video search provides more detailed tutorials for novice developers. Therefore, it is recommended that researchers pay more attentions for new tools for these new code search tasks.

8 THREATS TO VALIDITY

8.1 Publication Bias

Publication bias indicates the issue of publishing more positive results over negative results [54]. This is because positive results, e.g., a code search tool with statistically significant advantages over baselines, have a much higher chance of getting published. Meanwhile, negative results, e.g., the suspected flawed studies, are likely rejected for publication. Thus, to ensure the publication chance, some studies may report biased, incomplete and incorrect conclusions due to their low quality of experimental design (e.g., using limited or selected testing data). Therefore, the claims in this review supported or rejected by our selected major studies could be biased if the original literature suffers from such publication bias.

8.2 Search Terms

It is always challenging to find all relevant primary studies in any systematic review [54, 97]. To address this issue, we presented a detailed search strategy. The search string was constructed with terms identified by checking titles, abstracts, and keywords from many relevant papers that were already known to the authors. The adopted search string was piloted and the identified studies confirmed the applicability of the search string. These procedures provided high confidence that the majority of the key studies were included in the review.

8.3 Study Selection Bias

The study selection process was carried out in two phases. The first phase excluded studies based on the title and abstract by two independent researchers. A pilot study of the selection process was conducted to place a foundation for a better understanding of the inclusion/exclusion criteria. Potential disagreements were resolved during the pilot study and inclusion/exclusion criteria were refined. Inter-rater reliability was evaluated to mitigate the threat that emerged from the researchers' personal subjective judgment. The agreement between the two researchers was "substantial" for selecting relevant papers from the full set of papers. The selection process was repeated until a full agreement was achieved. When the researchers could not decide on a particular study, a third researcher was consulted. The second phase was based on the full text. Due to this well-established study selection process, it is unlikely that any relevant studies were missed.

8.4 Data Extraction

For data extraction, the studies were divided between two researchers; each researcher extracted the data from the relevant studies and the extracted data were rechecked by the other researcher. Issues in data extraction were discussed after the pilot data extraction and the researchers were able to complete the data extraction process following the refinement of the criteria. Extracted data were then inspected by automated scripts to check the correctness of the extracted values across the paper content, improving the validity of our analysis.

8.5 Data Analysis

We may have mis-compared some studies, mis-understood some of the techniques or evaluation methods reported, missed replication package links or failed to find repositories searching with paper title, or may have mis-understood reported dataset and evaluation metric details. From the extracted data we took due care in each of these areas to properly analyse, represent, classify and summarise the reviewed studies in this paper. However, there may be errors in our classification and analysis that impacts the overall findings of this review.

9 CONCLUSION

In recent years, researchers have proposed many tools to support the very common and important code search task to help boost developers' software development productivity and quality of code produced. To investigate the current state of research on code search, we performed a comprehensive review of 81 code search studies found from searching three electronic databases, ACM Digital Library, IEEEXplore, and ISI Web of Science. After extracting data from and analysing these selected studies, we found that:

- The popularity of code search research is substantially increasing with a peak in 2019, which accounts for more than 50% of the studies published from 2002-2020; 60% of studies were published in conferences; 83% of the studies proposed new code search tools.
- 74% of the reviewed tools focused on the general code search task (inputted with text-based queries, API-based queries, and input/output example) to improve the existing code search engines (e.g., GitHub Search); 21% of the tools aimed to search similar source/binary code from codebase; the remaining 5% of tools intended to search UI code or code in programming videos; in the recent two years, researchers frequently leverage DL techniques to tackle the challenges in early tools; but only 18% of code search tools shared accessible replication package links in their papers or provided source code in GitHub.
- To verify tool validity, method-level Java code collected from open source community (e.g., GitHub and FDroid) is the first choice to build a large-scale codebase; Representative search queries were commonly extracted from Q&A forums, development kit, and frequently used examples; most studies manually identified the relevancy of query and searched code and assessed the tool performance by using the ranking metrics (e.g., MRR and NDCG).

After analyzing the shortcomings of existing code search tools, we recommend further studies to build a better benchmark with large-scale code written in multiple programming languages, including various queries covering not only top frequently used examples but also domain-specific cases, and a standard automated tool evaluation method. DL-based tools require further improvements, such as better code representation, higher quality of training data, and speed improvements. It is recommended to fuse them with other models such as traditional IR-based and heuristic models. It is difficult to deploy an existing tool to search code written in multiple programming languages. Therefore, a multi-language tool is needed in the future. Additionally, new code search tasks such as searching UI code or code in programming videos are also worthy of more attention.

REFERENCES

[1] Hervé Abdi. 2007. The Kendall rank correlation coefficient. *Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA* (2007), 508–510.

[2] Afsoon Afzal, Manish Motwani, Kathryn Stolee, Yuriy Brun, and Claire Le Goues. 2019. SOSRepair: Expressive Semantic Search for Real-World Program Repair. *IEEE Transactions on Software Engineering* (2019).

[3] Parag Agrawal, Arvind Arasu, and Raghav Kaushik. 2010. On indexing error-tolerant set containment. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 927–938.

[4] Shayan Akbar and Avinash Kak. 2019. SCOR: source code retrieval with semantics and order. In *Working Conference on Mining Software Repositories*. IEEE, 1–12.

[5] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International Conference on Machine Learning*. 2123–2132.

[6] Sushil Bajracharya and Cristina Lopes. 2009. Mining search topics from a code search engine usage log. In *Working Conference on Mining Software Repositories*. IEEE, 111–120.

[7] Sushil Krishna Bajracharya and Cristina Videira Lopes. 2012. Analyzing and mining a code search engine usage log. *Empirical Software Engineering* 17, 4-5 (2012), 424–466.

[8] Sushil K Bajracharya, Joel Ossher, and Cristina V Lopes. 2010. Leveraging usage similarity for effective retrieval of examples in code repositories. In *International Symposium on Foundations of Software Engineering*. 157–166.

[9] Vipin Balachandran. 2015. Query by example in large-scale code repositories. In *IEEE International Conference on Software Maintenance and Evolution*. IEEE, 467–476.

[10] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, Minghui Wu, and Xiaohu Yang. 2020. psc2code: Denoising Code Extraction from Programming Screencasts. *ACM Transactions on Software Engineering and Methodology* 29, 3 (2020), 1–38.

[11] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. 2014. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering* 19, 3 (2014), 619–654.

[12] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: an efficient and robust access method for points and rectangles. In *ACM SIGMOD international conference on Management of data*. 322–331.

[13] Farnaz Behrang, Steven P Reiss, and Alessandro Orso. 2018. GUIfetch: supporting app design and development through GUI search. In *International Conference on Mobile Software Engineering and Systems*. 236–246.

[14] Tony Beltramelli. 2018. pix2code: Generating code from a graphical user interface screenshot. In *EICS*. 1–6.

[15] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. In *Noise reduction in speech processing*. Springer, 1–4.

[16] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *TACL* 5 (2017), 135–146.

[17] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 513–522.

[18] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1589–1598.

[19] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *International Symposium on Foundations of Software Engineering*. 964–974.

[20] Long Chen, Wei Ye, and Shikun Zhang. 2019. Capturing source code semantics via tree-based convolution over API-enhanced AST. In *ACM International Conference on Computing Frontiers*. 174–182.

[21] Zhengzhao Chen, Renhe Jiang, Zejun Zhang, Yu Pei, Minxue Pan, Tian Zhang, and Xuandong Li. 2020. Enhancing example-based code search with functional semantics. *Journal of Systems and Software* (2020), 110568.

[22] Charles LA Clarke, Maheedhar Kolla, Gordon V Cormack, Olga Vechtomova, Azin Ashkan, Stefan Büttcher, and Ian MacKinnon. 2008. Novelty and diversity in information retrieval evaluation. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*. 659–666.

[23] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[24] David Roxbee Cox and Alan Stuart. 1955. Some quick sign tests for trend in location and dispersion. *Biometrika* 42, 1/2 (1955), 80–95.

[25] Kostadin Damevski, David Shepherd, and Lori Pollock. 2014. A case study of paired interleaving for evaluating code search techniques. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 54–63.

[26] Kostadin Damevski, David Shepherd, and Lori Pollock. 2016. A field study of how developers locate features in source code. *Empirical Software Engineering* 21, 2 (2016), 724–747.

[27] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Acm Sigplan Notices* 49, 6 (2014), 349–360.

[28] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[29] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2016. Kam1n0: Mapreduce-based assembly clone search for reverse engineering. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 461–470.

[30] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *IEEE Symposium on Security and Privacy*. IEEE, 472–489.

[31] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process* 25, 1 (2013), 53–95.

[32] Brendan J Frey and Delbert Dueck. 2007. Clustering by passing messages between data points. *science* 315, 5814 (2007), 972–976.

[33] Wei Fu and Tim Menzies. 2017. Easy over hard: A case study on deep learning. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 49–60.

[34] Xi Ge, David Shepherd, Kostadin Damevski, and Emerson Murphy-Hill. 2014. How developers use multi-recommendation system in local code search. In *VL/HCC*. IEEE, 69–76.

[35] Mohammad Gharehyazie, Baishakhi Ray, and Vladimir Filkov. 2017. Some from here, some from there: Cross-project code reuse in github. In *Working Conference on Mining Software Repositories*. IEEE, 291–301.

[36] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *VLDB*, Vol. 99. 518–529.

[37] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *International Conference on Software Engineering*. IEEE, 933–944.

[38] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2019. CodeKernel: a graph kernel based approach to the selection of API usage examples. In *International Conference on Automated Software Engineering*. IEEE, 590–601.

[39] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 842–851.

[40] Jiawei Han, Jian Pei, and Micheline Kamber. 2011. *Data mining: concepts and techniques*. Elsevier.

[41] Simon Harris. 2003. Simian-similarity analyser. *HYPERLINK Available from: http://www. harukizaemon. com/simian/index. html* (2003).

[42] Gang Hu, Min Peng, Yihan Zhang, Qianqian Xie, Wang Gao, and Mengting Yuan. 2020. Unsupervised software repositories mining and its application to code search. *Software-Practice & Experience* 50, 3 (2020), 299–322.

[43] Qing Huang, An Qiu, Maosheng Zhong, and Yuan Wang. 2020. A Code-Description Representation Learning Model Based on Attention. In *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 447–455.

[44] Qing Huang and Guoqing Wu. 2019. Enhance code search via reformulating queries with evolving contexts. *Automated Software Engineering* 26, 4 (2019), 705–732.

[45] Qing Huang and Huaiguang Wu. 2019. QE-integrating framework based on Github knowledge and SVM ranking. *Science China. Information Science* 62, 5 (2019), 52102.

[46] Qing Huang, Yang Yang, and Ming Cheng. 2019. Deep learning the semantics of change sequences for query expansion. *Software-Practice & Experience* 49, 11 (2019), 1600–1617.

[47] Qing Huang, Yangrui Yang, Xue Zhan, Hongyan Wan, and Guoqing Wu. 2018. Query expansion based on statistical learning from code changes. *Software-Practice & Experience* 48, 7 (2018), 1333–1351.

[48] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *AMACL*. 2073–2083.

[49] He Jiang, Liming Nie, Zeyi Sun, Zhilei Ren, Weiqiang Kong, Tao Zhang, and Xiapu Luo. 2016. Rosf: Leveraging information retrieval and supervised learning for recommending code snippets. *IEEE Transactions on Services Computing* 12, 1 (2016), 34–46.

[50] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *International Conference on Software Engineering*. IEEE, 96–105.

[51] Renhe Jiang, Zhengzhao Chen, Zejun Zhang, Yu Pei, Minxue Pan, and Tian Zhang. 2018. Semantics-Based Code Search Using Input/Output Examples. In *International Working Conference on Source Code Analysis and Manipulation*. IEEE, 92–102.

[52] Huan Jin and Lei Xiong. 2019. A Query Expansion Method Based on Evolving Source Code. *Wuhan University Journal of Natural Sciences* 24, 5 (2019), 391–399.

[53] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[54] Staffs Keele et al. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report. Technical report, Ver. 2.3 EBSE Technical Report. EBSE.

[55] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *International Conference on Software Engineering*. 664–675.

[56] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. 2013. Rendezvous: A search engine for binary code. In *Working Conference on Mining Software Repositories*. IEEE, 329–338.

[57] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. 2010. Towards an intelligent code search engine.. In *AAAI*.

[58] Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY: a code-to-code search engine. In *International Conference on Software Engineering*. 946–957.

[59] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2019. Features and how to find them: a survey of manual feature location. *Software Engineering for Variability Intensive Systems* (2019), 153–172.

[60] Brian Kulis and Kristen Grauman. 2009. Kernelized locality-sensitive hashing for scalable image search. In *2009 IEEE 12th international conference on computer vision*. IEEE, 2130–2137.

[61] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *International Conference on Program Comprehension*. IEEE, 218–229.

[62] Moreno Laura, Bavota Gabriele, Di Penta Massimiliano, Oliveto Rocco, and Marcus Andrian. 2015. How Can I Use This Method?. In *International Conference on Software Engineering*. ACM.

[63] Mu-Woong Lee, Jong-Won Roh, Seung-won Hwang, and Sunghun Kim. 2010. Instant code clone search. In *International Symposium on Foundations of Software Engineering*. 167–176.

[64] Shin-Jie Lee, Xavier Lin, Wu-Chen Su, and Hsi-Min Chen. 2018. A comment-driven approach to API usage patterns discovery and search. *Journal of Internet Technology* 19, 5 (2018), 1587–1601.

[65] Otávio AL Lemos, Adriano C de Paula, Felipe C Zanichelli, and Cristina V Lopes. 2014. Thesaurus-based automatic query expansion for interface-driven code search. In *Working Conference on Mining Software Repositories*. 212–221.

[66] Otávio Augusto Lazzarini Lemos, Sushil Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Lopes. 2011. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Information and Software Technology* 53, 4 (2011), 294–306.

[67] Otávio Augusto Lazzarini Lemos, Adriano Carvalho de Paula, Gustavo Konishi, Sushil Krishna Bajracharya, Joel Ossher, Cristina Videira Lopes, et al. 2014. Thesaurus-Based Tag Clouds for Test-Driven Code Search. *Journal of Universal Computer Science* 20, 5 (2014), 772–796.

[68] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

[69] Hongwei Li, Zhenchang Xing, Xin Peng, and Wenyun Zhao. 2013. What help do developers seek, when and how?. In *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*. IEEE, 142–151.

[70] Wei Li, Shuhan Yan, Beijun Shen, and Yuting Chen. 2019. Reinforcement Learning of Code Search Sessions. In *Asia-Pacific Software Engineering Conference*. IEEE, 458–465.

[71] Xuan Li, Zerui Wang, Qianxiang Wang, Shoumeng Yan, Tao Xie, and Hong Mei. 2016. Relationship-aware code search for JavaScript frameworks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 690–701.

[72] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery* 18, 2 (2009), 300–336.

[73] Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2020. On the Replicability and Reproducibility of Deep Learning in Software Engineering. *arXiv preprint arXiv:2006.14244* (2020).

[74] Chao Liu, Xin Xia, David Lo, Zhiwei Liu, Ahmed E Hassan, and Shanping Li. 2020. Simplifying Deep-Learning-Based Model for Code Search. *arXiv preprint arXiv:2005.14373* (2020).

[75] Chao Liu, Dan Yang, Xin Xia, Meng Yan, and Xiaohong Zhang. 2018. Cross-project change-proneness prediction. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 64–73.

[76] Chao Liu, Dan Yang, Xin Xia, Meng Yan, and Xiaohong Zhang. 2019. A two-phase transfer learning model for cross-project defect prediction. *Information and Software Technology* 107 (2019), 125–136.

[77] Chao Liu, Dan Yang, Xiaohong Zhang, Haibo Hu, Jed Barson, and Baishakhi Ray. 2018. A recommender system for developer onboarding. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 319–320.

[78] Chao Liu, Dan Yang, Xiaohong Zhang, Baishakhi Ray, and Md Masudur Rahman. 2018. Recommending github projects for developer onboarding. *IEEE Access* 6 (2018), 52082–52094.

[79] Jason Liu, Seohyun Kim, Vijayaraghavan Murali, Swarat Chaudhuri, and Satish Chandra. 2019. Neural query expansion for code search. In *ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 29–37.

[80] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*. 31–42.

[81] Wenjian Liu, Xin Peng, Zhenchang Xing, Junyi Li, Bing Xie, and Wenyun Zhao. 2018. Supporting exploratory code search with differencing and visualization. In *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 300–310.

[82] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: code recommendation via structural code search. *OOPSLA* 3 (2019), 152.

[83] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 260–270.

[84] Lee Wei Mar, Ye-Chi Wu, and Hewijin Christine Jiau. 2011. Recommending proper API code examples for documentation purpose. In *Asia-Pacific Software Engineering Conference*. IEEE, 331–338.

[85] Lee Martie, Thomas D LaToza, and Andre van der Hoek. 2015. Codeexchange: Supporting reformulation of internet-scale code queries in context (T). In *International Conference on Automated Software Engineering*. IEEE, 24–35.

[86] Lee Martie and Andre Van der Hoek. 2015. Sameness: an experiment in code search. In *Working Conference on Mining Software Repositories*. IEEE, 76–87.

[87] Michael McCandless, Erik Hatcher, Otis Gospodnetić, and O Gospodnetić. 2010. *Lucene in action*. Vol. 2. Manning Greenwich.

[88] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *International Conference on Software Engineering*. 111–120.

[89] Collin McMillan, Negar Hariri, Denys Poshyvanyk, Jane Cleland-Huang, and Bamshad Mobasher. 2012. Recommending source code for use in rapid software prototypes. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 848–858.

[90] Collin Mcmillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. 2013. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Transactions on Software Engineering and Methodology* 22, 4 (2013), 1–30.

[91] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[92] Jeff Mitchell and Mirella Lapata. 2010. Composition in distributional models of semantics. *IEEE Computer Society* 34, 8 (2010), 1388–1429.

[93] Leann Myers and Maria J Sirois. 2004. S pearman Correlation Coefficients, Differences between. *Encyclopedia of statistical sciences* (2004).

[94] Brent D Nichols. 2010. Augmented bug localization using past bug information. In *Annual Southeast Regional Conference*. 1–6.

[95] Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. 2016. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing* 9, 5 (2016), 771–783.

[96] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2009), 1345–1359.

[97] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology* 64 (2015), 1–18.

[98] Denys Poshyvanyk and Mark Grechanik. 2009. Creating and evolving software by searching, selecting and synthesizing relevant source code. In *International Conference on Software Engineering-Companion Volume*. IEEE, 283–286.

[99] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis. In *International Conference on Software Engineering*. IEEE, 357–367.

[100] Chaiyong Ragkhitwetsagul and Jens Krinke. 2019. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering* 24, 4 (2019), 2236–2284.

[101] Md Masudur Rahman, Jed Barson, Sydney Paul, Joshua Kayani, Federico Andrés Lois, Sebastián Fernandez Quezada, Christopher Parnin, Kathryn T Stolee, and Baishakhi Ray. 2018. Evaluating how developers use general-purpose web-search for code retrieval. In *Working Conference on Mining Software Repositories*. 465–475.

[102] Sukanya Ratanotayanon, Hye Jung Choi, and Susan Elliott Sim. 2010. My repository runneth over: an empirical study on diversifying data sources to improve feature search. In *International Conference on Program Comprehension*. IEEE, 206–215.

[103] Steven P Reiss. 2009. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 243–253.

[104] Steven P Reiss, Yun Miao, and Qi Xin. 2018. Seeking the user interface. *Automated Software Engineering* 25, 1 (2018), 157–193.

[105] Stephen Robertson and Hugo Zaragoza. 2009. *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc.

[106] Barbara Rosario. 2000. Latent semantic indexing: An overview. *Techn. rep. INFOSYS* 240 (2000), 1–16.

[107] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

[108] Julia Rubin and Marsha Chechik. 2013. A survey of feature location techniques. In *Domain Engineering*. Springer, 29–58.

[109] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 31–41.

[110] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *International Symposium on Foundations of Software Engineering*. 191–201.

[111] Naiyana Sahavechaphan and Kajal Claypool. 2006. XSnippet: mining for sample code. In *ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 413–430.

[112] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling code clone detection to big-code. In *International Conference on Software Engineering*. 1157–1168.

[113] Gerard Salton, Edward A Fox, and Harry Wu. 1983. Extended Boolean information retrieval. *Commun. ACM* 26, 11 (1983), 1022–1036.

[114] Abdullah Sheneamer and Jugal Kalita. 2016. A survey of software clone detection techniques. *International Journal of Computer Applications* 137, 10 (2016), 1–21.

[115] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving Code Search with Co-Attentive Representation Learning. In *International Conference on Program Comprehension*.

[116] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V Lopes. 2011. How well do search engines support code retrieval on the web? *ACM Transactions on Software Engineering and Methodology* 21, 1 (2011), 4.

[117] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 2010. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*. IBM Corp., 174–188.

[118] Raphael Sirres, Tegawendé F Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim, and Yves Le Traon. 2018. Augmenting and structuring user queries to support efficient free-form code search. *Empirical Software Engineering* 23, 5 (2018), 2622–2654.

[119] Bunyamin Sisman and Avinash C Kak. 2013. Assisting code search with automatic query reformulation for bug localization. In *Working Conference on Mining Software Repositories*. IEEE, 309–318.

[120] Jamie Starke, Chris Luce, and Jonathan Sillito. 2009. Searching and skimming: An exploratory study. In *IEEE International Conference on Software Maintenance and Evolution*. IEEE, 157–166.

[121] Kathryn T Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 3 (2014), 26.

[122] Kathryn T Stolee, Sebastian Elbaum, and Matthew B Dwyer. 2016. Code search with input/output queries: Generalizing, ranking, and assessment. *Journal of Systems and Software* 116 (2016), 35–48.

[123] Rui Sun, Hui Liu, and Leping Li. 2019. Slicing Based Code Recommendation for Type Based Instance Retrieval. In *International Conference on Software and Systems Reuse*. Springer, 149–167.

[124] Suresh Thummalapenta and Tao Xie. 2007. Parseweb: a programmer assistant for reusing open source code on the web. In *International Conference on Automated Software Engineering*. 204–213.

[125] Suresh Thummalapenta and Tao Xie. 2009. Alattin: Mining alternative patterns for detecting neglected conditions. In *International Conference on Automated Software Engineering*. IEEE, 283–294.

[126] Suresh Thummalapenta and Tao Xie. 2011. Alattin: mining alternative patterns for defect detection. *Automated Software Engineering* 18, 3-4 (2011), 293–323.

[127] Lisa Torrey and Jude Shavlik. 2010. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, 242–264.

[128] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *International Conference on Automated Software Engineering*. IEEE, 13–25.

[129] Bei Wang, Ling Xu, Meng Yan, Chao Liu, and Ling Liu. 2020. Multi-Dimension Convolutional Neural Network for Bug Localization. *IEEE Transactions on Services Computing* (2020).

[130] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. 2013. Mining succinct and high-coverage API usage patterns from source code. In *Working Conference on Mining Software Repositories*. IEEE, 319–328.

[131] Jianyong Wang and Jiawei Han. 2004. BIDE: Efficient mining of frequent closed sequences. In *Proceedings. 20th international conference on data engineering*. IEEE, 79–90.

[132] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *International Conference on Automated Software Engineering*. IEEE, 87–98.

[133] Norman Wilde, Ross Huitt, and Scott Huitt. 1989. Dependency analysis tools: reusable components for software maintenance. In *Proceedings. Conference on Software Maintenance*. IEEE, 126–131.

[134] Huaiguang Wu and Yang Yang. 2019. Code search based on alteration intent. *IEEE Access* 7 (2019), 56796–56802.

[135] Ho Chung Wu, Robert Wing Pong Luk, Kam Fai Wong, and Kui Lam Kwok. 2008. Interpreting tf-idf term weights as making relevance decisions. *ACM Transactions on Information Systems (TOIS)* 26, 3 (2008), 1–37.

[136] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E Hassan, and Zhenchang Xing. 2017. What do developers search for on the web? *Empirical Software Engineering* 22, 6 (2017), 3149–3185.

[137] Yingtao Xie, Tao Lin, and Hongyan Xu. 2019. User Interface Code Retrieval: A Novel Visual-Representation-Aware Approach. *IEEE Access* 7 (2019), 162756–162767.

[138] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. 2018. Accurate and scalable cross-architecture cross-os binary code search with emulation. *IEEE Transactions on Software Engineering* 45, 11 (2018), 1125–1149.

[139] Shuhan Yan, Hang Yu, Yuting Chen, Beijun Shen, and Lingxiao Jiang. 2020. Are the Code Snippets What We Are Searching for? A Benchmark and an Empirical Study on Code Search with Natural-Language Queries. In *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 344–354.

[140] Yangrui Yang and Qing Huang. 2017. IECS: Intent-enforced code search via extended boolean model. *Journal of Intelligent & Fuzzy Systems* 33, 4 (2017), 2565–2576.

[141] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. CoaCor: code annotation for code retrieval with reinforcement learning. In *The World Wide Web Conference*. 2203–2214.

[142] Wei Ye, Rui Xie, Jinglei Zhang, Tianxiang Hu, Xiaoyin Wang, and Shikun Zhang. 2020. Leveraging Code Generation to Improve Code Retrieval and Summarization via Dual Learning. In *The World Wide Web Conference*. 2309–2319.

[143] Feng Zhang, Haoran Niu, Iman Keivanloo, and Ying Zou. 2017. Expanding queries for code search using semantically related api class-names. *IEEE Transactions on Software Engineering* 44, 11 (2017), 1070–1082.

[144] Jingxuan Zhang, He Jiang, Zhilei Ren, Tao Zhang, and Zhiqiu Huang. 2019. Enriching API Documentation with Code Samples and Usage Scenarios from Crowd Knowledge. *IEEE Transactions on Software Engineering* (2019).

[145] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[146] Jingtian Zhang, Sai Wu, Zeyuan Tan, Gang Chen, Zhushi Cheng, Wei Cao, Yusong Gao, and Xiaojie Feng. 2019. S3: a scalable in-memory skip-list index for key-value store. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2183–2194.

[147] Yu Zhang and Qiang Yang. 2017. A survey on multi-task learning. *arXiv preprint arXiv:1707.08114* (2017).

[148] Yu Zhang and Qiang Yang. 2018. An overview of multi-task learning. *National Science Review* 5, 1 (2018), 30–43.

[149] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming*. Springer, 318–343.

[150] Qun Zou and Changquan Zhang. 2020. Query expansion via learning change sequences. *International Journal of Knowledge-based and Intelligent Engineering Systems* 24, 2 (2020), 95–105.