Code Search: A Survey of Techniques for Finding Code

LUCA DI GRAZIA and MICHAEL PRADEL, Department of Computer Science, University of Stuttgart, Germany

The immense amounts of source code provide ample challenges and opportunities during software development. To handle the size of code bases, developers commonly search for code, e.g., when trying to find where a particular feature is implemented or when looking for code examples to reuse. To support developers in finding relevant code, various code search engines have been proposed. This article surveys 30 years of research on code search, giving a comprehensive overview of challenges and techniques that address them. We discuss the kinds of queries that code search engines support, how to preprocess and expand queries, different techniques for indexing and retrieving code, and ways to rank and prune search results. Moreover, we describe empirical studies of code search in practice. Based on the discussion of prior work, we conclude the article with an outline of challenges and opportunities to be addressed in the future.

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

Additional Key Words and Phrases: Code search, code retrieval, API, learning, survey

ACM Reference format:
Luca Di Grazia and Michael Pradel. 2023. Code Search: A Survey of Techniques for Finding Code. *ACM Comput. Surv.* 55, 11, Article 220 (February 2023), 31 pages.
https://doi.org/10.1145/3565971

1 INTRODUCTION

Many kinds of information are stored in digital systems, which offer convenient access, large storage capacities, and the ability to process information automatically. To enable people to quickly find digitally stored information, research on information retrieval has led to powerful search engines. Today, commercial search engines are used by billions of people every day to retrieve various kinds of information [112], such as textual information, images, or videos.

As software is becoming increasingly important in various aspects of our lives, a particular kind of information is being produced in incredibly large amounts: source code. A single, complex software project, such as the Linux kernel or modern browsers, easily comprises multiple millions of lines of source code. At the popular open source project platform GitHub, more than 60 million new projects have been created in 2020 alone [32]. The sheer amount of existing source code leads to a situation where most code to be written by a developer either has already been written elsewhere, or at least, is similar to some code that has already been written [30, 47, 102].

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and DeMoCo projects.
Authors' address: L. Di Grazia and M. Pradel, Department of Computer Science, University of Stuttgart, Universitätsstr. 38, 70569 Stuttgart, Germany; emails: luca.di-grazia@iste.uni-stuttgart.de, michael@binaervarianz.de.

0360-0300/2023/02-ART220 $15.00
https://doi.org/10.1145/3565971

To benefit from existing source code and to efficiently navigate complex code bases, software developers often search for code [109]. For example, a developer may search through a code base she is working on to find where some functionality is implemented, to understand what a particular piece of code is doing, or to find other code locations that need to be changed while fixing a bug. Beyond the code base a developer is working on, developers also commonly search through other projects within an organization or through open source projects. For example, a developer may look for examples of how to implement a specific functionality, search for usage examples of an **application programming interface (API)**, or simply cross-check newly written code against similar existing code. We call these and related activities *code search*. To support developers during code search, *code search engines* automatically retrieve code examples relevant to a given query from one or more code bases.

At a high level, the challenges for building a successful code search engine are similar to those in general information retrieval: provide a convenient querying interface, produce results that match the given query, and do so efficiently. Beyond these high-level similarities, code search comes with interesting additional opportunities and challenges. As programming languages have a formally defined syntax, one can unambiguously parse source code, and then analyze and compare it based on its structural properties [2]. Moreover, source code also has well-defined runtime semantics, as given by the specification of the programming language, e.g., for Java [33], C++ [1], or JavaScript [26]. That is, in contrast to natural language text and other kinds of information targeted by search engines, the meaning of a piece of source code is, at least in principle, well defined. In practice, the code in a large code corpus often is written in a diverse set of programming languages, building on various frameworks and libraries, and using different coding styles and conventions [45]. As a result, code search engines must strike a balance between precisely analyzing code in a specific language and supporting a wide range of languages [113]. Finally, the language in which a query is formulated may not be the same as the language the search results are written in. For example, many code search engines accept natural language queries or behavioral specifications of the code to retrieve, which requires some form of mapping between such queries and code [90].

Motivated by the need to search through the huge amounts of available source code and by the challenges and opportunities it implies, code search has received significant attention by researchers and practitioners. The progress made in the field is good news for developers, as they can benefit from increasingly sophisticated code search engines. At the same time, the impressive amount of existing work makes it difficult for new researchers and interested non-experts to understand the state of the art and how to improve upon it. This article summarizes existing work on code search and describes how different approaches relate to each other. By providing a comprehensive survey of 30 years of work on code search, we hope to provide an overview of this thriving research field. Based on our discussion of existing work, we also point out open challenges and opportunities for future research.

Figure 1 shows the number of papers we discuss per year of publication, illustrating the increasing relevance of the topic. Our survey primarily targets full research papers, i.e., more than six pages, from top-ranked conferences and journals.[1] In addition, we include other publications, e.g., in workshop proceedings, papers on arXiv, and technical reports, as well as publications at lower-ranked venues, if and only if they are recent (less than two years), have had a significant impact (more than 10 citations), or provide a very strong match with the topic of this survey. We use three different platforms to search for papers: Google Scholar,[2] the ACM Digital Library,[3] and

[1]Specifically, venues ranked A* or A in the CORE ranking: http://portal.core.edu.au.
[2]https://scholar.google.com/.
[3]https://dl.acm.org/.

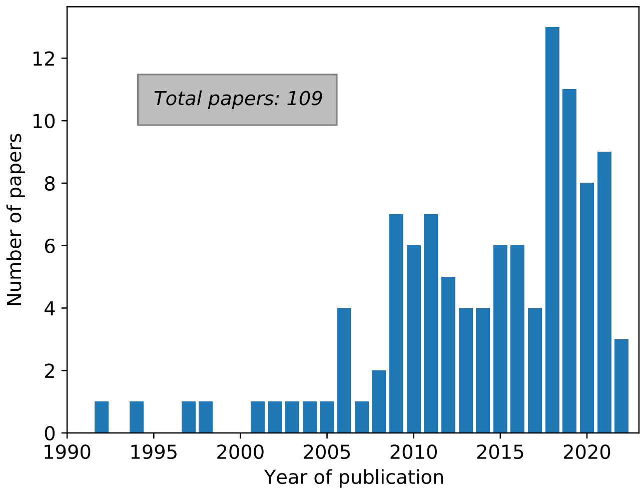


Fig. 1. Papers on code search discussed in this article.

DBLP.[4] To find an initial set of papers, we search with queries "code search" and "code retrieval." Afterwards, we iteratively refined the set of considered papers by following citations, both backward and forward, until reaching a fixed point.

There are several research fields related to code search that are out of the scope of this article. In particular, we do not discuss in detail work on general software repository mining, e.g., to extract patterns or programming rules [50], searching for entire applications, e.g., in an app store [35, 82], and query-based synthesis of new code examples [39]. Moreover, we do not cover in detail work on code clone detection [107] and code completion [14, 105], as those are related but different problems. Code clone detection aims at finding pieces of code that are semantically, and perhaps even syntactically, equivalent to each other, whereas code search aims at finding code that offers more details than a given query. Code completion can be seen as a restricted variant of code search, where the code a developer has already written serves as a query to find the next few tokens or even lines to insert. An important difference is that code search tries to retrieve existing code as-is, whereas code completion synthesizes potentially new code fragments.

Figure 2 outlines the components a typical code search engine is built from and, at the same time, gives an overview of the topics covered in this article. Most code search engines have an offline part, which indexes a code corpus or trains a machine learning model on a code corpus, and an online part, which takes a user-provided query and retrieves code examples that match the query.

- Section 2 presents different kinds of queries accepted by code search engines, including natural language, code snippets, formal specifications, test cases, and queries written in specifically designed querying languages.
- Section 3 describes how code search engines preprocess and expand a given query, e.g., by generalizing terms in a natural language query or by lifting a given code snippet to a richer representation.
- Section 4 presents the core component of a code search engine, which indexes code examples or trains a machine learning model, and then retrieves examples that match a query. We discuss and compare several approaches based on how they represent the code and what kind of retrieval technique they use.

[4]dblp.uni-trier.de/.

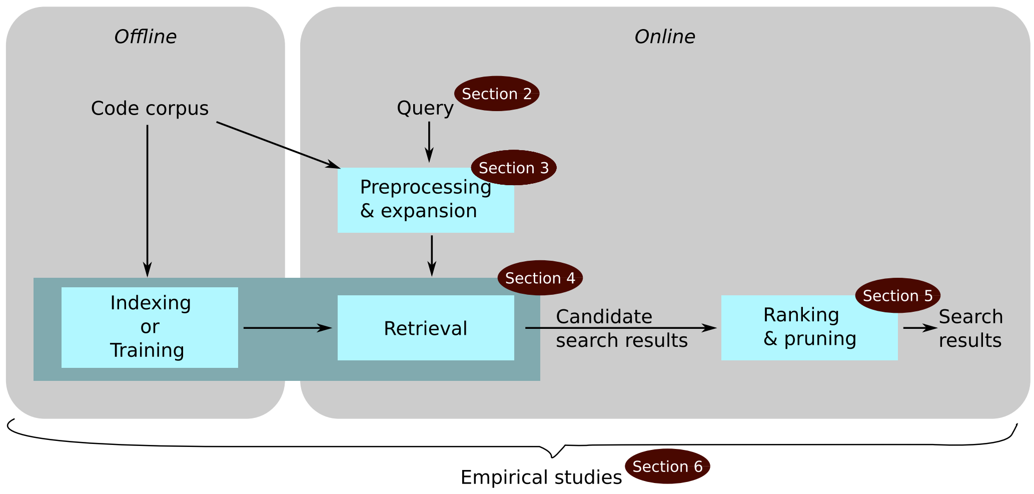


Fig. 2. Overview of the topics covered in this article.

- Section 5 presents different techniques for ranking and pruning search results before presenting them to the user, e.g., based on similarity scores between code examples and the query, or based on clustering similar search results.
- Section 6 discusses empirical studies of developers and how they interact with code search engines, which connects the research described in the other sections to adoption in practice.
- Section 7 outlines several open challenges and research directions for future work.

Prior work surveys code search techniques from different perspectives than this article. Garcia et al. [31] summarize code search-related tools presented until 2006, with a focus on tools aimed at software reuse. Another survey [23] is about techniques for locating where in a project a particular feature or functionality is implemented. While being a problem related to code search, feature location focuses on searching through a single software project, instead of large code corpora, and on the specific use case of locating a feature, instead of the wider range of use cases covered by code search. A short paper by Khalifa [53] discusses existing techniques for code search, focusing on information retrieval-based and deep learning-based approaches, but it covers only five papers. Finally, another survey of code search techniques [69] focuses on general publication trends, application scenarios where code search is used, and how search engines are evaluated. In contrast, this article focuses more on the technical core of code search engines, including different querying languages, pre-processing of queries, ranking and pruning of results, and also empirical studies of code search in practice.

2 QUERIES FOR SEARCHING CODE

The starting point of every search is a query. We define a query as an explicit expression of the intent of the user of a code search engine. This intent can be expressed in various ways, and different code search engines support different kinds of queries. The designers of a code search engine typically aim at several goal when deciding what kinds of queries to support:

- *Ease.* A query should be easy to formulate, enabling users to use the code search engine without extensive training. If formulating an effective query is too difficult, then users may get discouraged from using the code search engine.

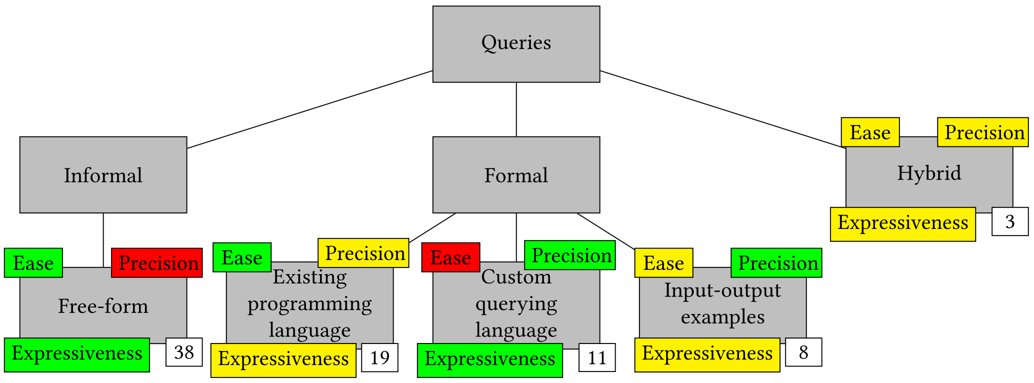


Fig. 3. Taxonomy of code search queries and number of approaches accepting each kind of query.

- *Expressiveness.* Users should be able to formulate whatever intent they have when searching for code. If a user is unable to express a particular intent, then the search engine cannot find the desired results.
- *Precision.* The queries should allow specifying the intent as unambiguously as possible. If the queries are imprecise, then the search is likely to yield irrelevant results.

These goals are non-trivial to reconcile, and different code search techniques balance this trade-off in different ways. Figure 3 shows a taxonomy of the kinds of queries supported by existing approaches. Broadly, we can distinguish between informal queries, formal queries, and combinations of the two. The numbers associated with the leaf nodes of the taxonomy indicate how many papers support each kind of query. The figure also shows how well different approaches achieve the three goals from above. The color of the boxes containing "Ease," "Precision," and "Expressiveness" indicate the support for these goals, where green means strong support, yellow means medium support, and red means little support. The remainder of this section discussed the different kinds of queries in more detail, following the structure lined out in the taxonomy.

2.1 Free-form Queries

Free-form queries are an informal way of specifying the intent of a code search. Such a query may describe in natural language the functionality of the searched code, e.g., "read file line by line." Free-form queries may also contain programming language elements, e.g., when searching for identifier names of a specific API, such as "FileReader close."

Free-form queries are the most commonly used kind of query in the approaches we survey [3, 8, 16–18, 20, 22, 24, 25, 36, 37, 41, 42, 66–68, 70–73, 76, 79, 83, 84, 90, 91, 101, 103, 108, 111, 113, 118, 124, 128, 130, 131, 134, 136, 137] They are attractive as users can easily formulate a query, similar to using a general-purpose web search engine, with a high level of expressiveness. On the downside, free-form queries risk being imprecise. One reason is that natural language terms are ambiguous. For example, the term "float" may refer to either a data type or to a verb. Another reason is that the vocabulary in a query may not match the vocabulary used in a code base. For example, a search term "array" may refer to a data structure that syntactically occurs as two square brackets in Java or Python [128].

Because free-form queries are extremely versatile, different code search engines compare them against different kinds of data. One set of approaches compares free-form queries against natural language text associated with code, e.g., API documentation [17], commit messages [18], or words in the a project's metadata [83]. Another set of approaches compares queries against the source code, e.g., by matching the query against signatures of fields and methods [41, 42] or against all identifiers in the code [73, 79, 113].

The informal nature of free-form queries may make it difficult to accurately match a query against a code snippet, e.g., because of a vocabulary mismatch between the two. For example, plain English queries, such as “match regular expression” or “read text file” [101], may not match the terms used in the corresponding API methods. A popular way to mitigate this mismatch is to project natural language words and source code identifiers into a common vector space [90] via learned word embeddings, such as Word2Vec [85]. Another way to address the limitations of free-form queries is to preprocess and expand queries, which we discuss further in Section 3.

To avoid the ambiguity of free-form queries and because source code is anyway written in a formal language, many code search engines support some kind of formal queries, which we discuss in the following. The commonality of these queries is that they are written in a language with a formally specified syntax and sometimes also formally defined semantics.

Summary: Free-form queries are easy to type and highly expressive, but they can be ambiguous and less precise than other, more formal kinds of queries.

2.2 Queries Based on Existing Programming Languages

As a first kind of formal query, we start by discussing queries based on existing programming languages. A query here is a snippet of code, possibly using some additional syntax not available in the underlying programming language. Because developers already know the programming language they are using, such queries are easy to formulate. The expressiveness and precision of code queries varies depending on the intent of the user and the specific search engine.

Queries based on existing programming languages roughly fall into three categories:

(1) *Plain code.* The most simple kind of code query are snippets of code as defined by the syntax of the underlying programming language [9, 29, 55, 61, 62, 75, 81, 89, 126, 138, 139]. For example, the following query provides a partial implementation, for which the user seeks ways to extend it [9]:

```
try {
  File file = File.createTempFile("foo", "bar");
} catch (IOException e) { }
```

(2) *Code with holes.* Instead of letting the search engine figure out where to extend a given code snippet, some search engines support queries that explicitly define one or more holes in the given code [86, 88]. For example, this query specifies that the user looks for how to complete the body of the given method [88]:

```
public void actionClose(JButton a, JFrame f) {
  __CODE_SEARCH__;
}
```

(3) *Code with pattern matching symbols.* A very precise way of describing the code to search is a query in an extension of the underlying programming language that adds patterns matching symbols. For example, such queries may define where an expression, here denoted with #, or a statement, here denoted with @, is missing [94, 95]:

```
if (# = #) @;
```

Such a query provides an abstract template for the code to search, and the search engine tries to retrieve some or all code snippets that the template can be extended into.

A recurring challenge for search engines that accept queries written in (variants of) existing programming languages is the problem of parsing incomplete code snippets [55, 86, 133]. An off-the-shelf grammar of the programming language may not be able to parse a query, because the query does not encompass a complete source code file or because the code is incomplete. One way to address this problem [55] is to heuristically fix a given code fragment, e.g., by surrounding it with additional code.

A popular kind of application of search engines that accept partial code snippets is as a source code recommendation tool. To ensure that the recommended code matches the current context a developer is working in, e.g., the current file and project, some approaches consider the code around the actual query as context available to the search engine. For example, Holmes and Murphy [43] and Brandt et al. [11] propose to integrate code search directly into the **integrated development environment (IDE)**. Other approaches [88, 126, 138] spontaneously search and display example code snippets while the developer is editing a program. The underlying idea of these approaches is that the user should not spend time on formulating the query, but simply uses the already typed code. Finally, general code completion systems also predict code based on the existing code context while a developer is writing code. For example, GitHub's Copilot[5] suggests multiple lines of code using a large-scale generative neural language model [19]. In contrast to code search, code completion synthesizes suitable code, regardless of whether exactly this code has already been written anywhere, whereas code search retrieves existing code as-is.

Instead of queries in a high-level programming language, some code search engines accept binary code as a query. For example, an approach by David and Yahav [21] accepts a function in its compiled, binary form as a query and then searches for similar functions in a corpus of binaries. Another approach accepts an entire binary as the query, trying to find other binaries that may be compiled from the same or similar source code [54]. Binary-level code search has several applications in security, e.g., to check for occurrences of known vulnerable code, and in copyright enforcement, e.g., to find code copied without permission.

Summary: Program language queries are easy to type, because users do not have to learn a new language, but the expressiveness and precision of code queries varies depending on the intent of the user and the specific search engine.

2.3 Custom Querying Languages

A common alternative to queries based on an existing programming programming language are custom querying languages. They provide a high degree of expressiveness and precision, at the expense of reduced ease of use, because users need to learn the querying language.

2.3.1 Logic-based Querying Languages. The most prevalent kind of custom querying languages is first-order logic predicates that describe properties of the code to search. For example, Janzen and Volder [48] extend the logic programming language TyRuBa[6] to support queries such as the following, which searches for a package with a class called "HelloWorld,"

```
package(?P, class, ?C), class(?C, name, HelloWorld)
```

In a similar vein, Hajiyev et al. [40] describe a code querying technique based on Datalog queries. Datalog is a logic-based language that, given a set of elements and relationships between the elements, answers queries. The approach considers program elements, e.g., classes and methods, and

[5] https://copilot.github.com/.
[6] http://tyruba.sourceforge.net/.

several relationships between them, e.g., the fact that a class inherits from another class, or that a class has a method. A user can query a code base by formulating logic-based queries over these elements and relationships, such as asking for all methods in a class called "A," where "A" inherits from a class called "B." Other approaches support logical queries over identifiers and structural relationships between them [116, 129, 132].

Several languages allow for predicates beyond describing program elements and their relationships. One example is to also support meta-level properties, such as how many imports a file has. For example, the query language by Martie et al. [78] allows for queries such as

```
import count > 5 AND extends class FooBar
```

The Alice search engine [120] supports a kind of semantic predicates, e.g., to search for code that calls the `readNextFile` method in a loop and handles an exception of a type `FileNotFoundException`.

2.3.2 Significant Extensions of Existing Programming Languages. Instead of logic-based querying languages, several search engines accept queries in custom languages that significantly extend an existing programming language. Similarly to the kinds of queries discussed in Section 2.2, such queries contain fragments of an existing programming language. One such language is by Inoue et al. [47] who support different kinds of wildcard tokens that match any single token, any token sequence, or any token sequence discarding paired brackets, respectively. In addition, their queries may use popular regular expression operators for choice, repetition, and grouping to enhance the expressiveness. For example, the following query will search for nested if-else clauses:

```
$( if $$ else $) $+
```

Another significant extension of an existing programming language is the "semantic patch language" of Coccinelle [58]. It allows for describing a patch, as produced by the popular *diff* tool, augmented with metavariables that match a specific piece of code and with a wildcard operator. A query hence describes a set of rules that the old and the new code must match, which then used to search for specific code changes in the version history of a project [57].

2.3.3 Other Custom Languages. A custom querying language by Premtoon et al. [100] describes code in a way that can be mapped to a program expression graph, which describes computations via operator nodes and dataflow edges [127]. In contrast to the above approaches, their queries are not specific to a single programming language, but can be used to search through projects in multiple languages.

Summary: Custom querying language queries can offer high expressiveness and precision, but are (at least initially) less easy to type, because users have to learn the custom language first.

2.4 Input–Output Examples as Queries

All kinds of queries discussed so far focus on the source code itself, but neglect an important property of code that distinguishes it from other kinds of data supported by search engines, such as text: executability. To exploit this property, some search engines support queries that are behavioral specifications and that characterize examples of the code behavior. Such queries typically come in the form of one or more input–output examples of the code to search.

The pioneering work by Podgurski and Pierce [97] is the first to propose input–output examples as queries, and other approaches adopt and improve this idea [49, 106, 121–123]. For example, these search engines enable users to search for code that given the input "susie@mail.com" produces "susie" [121]. Beyond supporting developers who search for specific kinds of code, another application of input–output-based code search is to find code fragments that can be used in automated program repair [51].

An extended form of input–output examples are queries in the form of executable test cases [63, 64]. Adapting the test-driven development paradigm, the basic idea is that a developer first implements test cases for some functionality and then searches for existing code that provides the desired functionality. Test cases here serve two purposes: First, they define the behavior of the desired code to be searched. Second, they test the search results for suitability in the local context.

Summary: Using input–output examples as queries allows for precisely specifying the desired behavior, but providing sufficiently many examples to fully express this behavior may require some effort.

2.5 Hybrids of Informal and Formal Queries

A few approaches support not only one kind of query, but hybrid queries that combine multiple of the kinds described above. One example is the work by Reiss [106], which in addition to input–output examples supports free-form queries. For example, a user may search for a method that mentions "roman numeral" and produces "XVII" for the input "17." Another kind of hybrid query combines free-form, natural language terms with references to program elements [133], e.g., "sort playerScores in ascending order," where "playerScores" refers to a variable in the code. Finally, Martie et al. [77] mix free-form queries and logical queries over code properties. For example, a query may ask for code that matches the keywords "http servlet," extends a class `httpservlet`, and has more than three imports.

3 PREPROCESSING AND EXPANSION OF QUERIES

The query provided by a user may not be the best possible query to obtain the results a user expects. One reason is that natural language queries suffer from the inherent imprecision of natural language. Another reason is that the vocabulary used in a query may not match the vocabulary used in a potential search result. For example, a query about "container" is syntactically different from "collection," but both refer to similar concepts. Finally, a user may initially be unsure what exactly she wants to find, which can cause the initial query to be incomplete.

To address the limitations of user-provided queries, approaches for preprocessing and expanding queries have been developed. We discuss these approaches by focusing on three dimensions: (i) the user interface, i.e., if and how a user gets involved in modifying queries, (ii) the information used to modify queries, i.e., what additional source of knowledge an approach consults, and (iii) the actual technique used to modify queries. Table 1 summarizes different approaches along these three dimensions, and we discuss them in detail in the following.

3.1 User Interface of Query Preprocessing and Expansion Approaches

Transparent vs. interactive. The majority of code search engines that perform some form of query preprocessing or expansion do so in a fully transparent way, i.e., the user is not aware of this part of the approach. For example, Sisman and Kak [119] propose to automatically expand queries using similar terms from search results, while others transparently expand queries using dictionaries [66] and synonymous [73]. An exception is the work by Martie et al. [77, 78], where the user interactively reformulates queries based on keyword recommendations made by the search engine. Another interactive approach [72] collects relations between words from the source code,

Table 1. Overview of Approaches for Preprocessing and Expansion of Queries

	Paul and Prakash [95]	Wang et al. [129]	Shepherd et al. [113]	Sisman and Kak [119]	Lv et al. [76]	Lu et al. [73]	Martie et al. [78]	Li et al. [66]	Nie et al. [91]	Martie et al. [77]	Sirres et al. [118]	Rahman and Roy [103]	Lu et al. [72]	Wu and Yang [134]	Li et al. [65]
User interface:															
Transparent to user	✓	✓	✓	✓	✓	✓		✓	✓		✓	✓		✓	✓
Based on (implicit) user feedback				✓			✓			✓					✓
Information used to modify queries:															
Initial search results				✓			✓			✓				✓	
Similarity of search terms and/or identifiers				✓	✓	✓		✓	✓		✓	✓	✓		
NL/code dataset (e.g., Stack Overflow)					✓				✓		✓	✓			
Recurring code changes														✓	
Technique used to modify queries:															
Weigh search terms			✓									✓			
Add or replace search terms				✓	✓		✓	✓	✓		✓	✓	✓	✓	
Lift query to richer representation	✓	✓					✓								

such as that one type name inherits from another, or that a word is part of a compound word, and then removes irrelevant words using an English dictionary. After this process the user can give a feedback about the query and iterate the query refinement until satisfied.

User feedback. To improve the initially given queries, some approaches rely on feedback by the user. Such feedback can be given explicitly, as in the work by Martie et al. [77, 78], where a user can up-rate or down-rate particular search results, which is then used to show more or less search results with similar features. Instead of explicit feedback, Sisman and Kak [119] rely on so-called pseudo relevance feedback given implicitly through the highest-ranked search results retrieved for initially given query. The approach then enriches the initial query with search terms drawn from the initial search results. Another way of using implicit user feedback is by observing what search results a user clicks on, which may provide valuable information on what the user is searching for. The Cosoch approach exploits this feedback in a reinforcement learning-based approach [65]. Their approach tracks across multiple queries that search results a user selects, and tries to maximize the normalized discounted cumulative gain, which measures the quality of a ranked list of search results.

Summary: User interfaces can help preprocessing queries in a transparent way or using feedback from users.

3.2 Information Used to Modify Queries

Initial search results. Effectively modifying a query requires some information in addition to the query itself. Several approaches use the results returned for the initially provided query for this purpose [77, 78, 119]. A downside of relying on initial search results to modify queries is that the search must be performed multiple times before obtaining the final search results, which may negatively affect efficiency.

Similarity of search terms and/or identifiers. A commonality of several approaches for query preprocessing and expansion is to compare words or identifiers in a query with those in a potential search result through some kind of similarity measure. One approach [119] builds on the observation that terms in search results that frequently appear close to terms in the query may also be relevant, and then expands the initial query with those words. Others builds on domain-specific dictionaries [66] or on synonyms [73] found using WordNet [59] to add or replace query terms with related terms. A more recent alternative to curated databases of word similarities are learned word embeddings, e.g., via Word2vec [85], which can help in revising queries [103].

NL/code datasets. A third kind of information used by several approaches to revise queries are datasets of documents that combine natural language and code. For example, Lv et al. [76] use API documentation to identify which API a query is likely to refer to and then expand the query accordingly. Online discussion forums with programming-related questions and answers, e.g., Stack Overflow also have been found to help in revising queries [91, 103, 118]. These approaches search online posts related to a given query, and then extract additional relevant words, software-specific terms, and API identifiers to augment the query. Since the questions and answers cover various application domains and are curated based on feedback by thousands of developers, they provide a valuable dataset to associate natural language words with related programming terms.

Recurring code changes. Motivated by the observation that developers may have to adapt a retrieved code example, e.g., to use the most recent version of an API, Wu and Yang [134] expand queries to proactively consider such potential code adaptations. At first, their approach mines recurring code changes from version histories of open source projects, which provides information such as that a code token `A` is often changed to a code token `B`. Given a user query, they then retrieve matching code examples, and if these examples include a frequently changed token, say `A`, expand the query with the updated token, say `B`. With the expanded query, the search engine hence will retrieve updated versions of the code examples, freeing the developer from adapting the code manually.

Summary: To automatically modify queries, code search engines most commonly use similarities between search terms and identifiers, as well as corpora of natural language and code.

3.3 Techniques Used to Modify Queries

Weigh search terms. The perhaps most straightforward way of augmenting a given search query is to weigh the given search terms. Several code search engines implement this idea [103, 113], with the goal of giving terms that are most relevant for finding suitable results at a higher weight. For example, Rahman and Roy [103] estimate the weights of API class names by applying the page rank algorithm [12] to an API co-occurrence graph.

Add or replace search terms. Another common technique is to add or replace terms in the given search query, e.g., by adding terms that are related or synonymous to those already in the query. Lv et al. [76] propose a query refinement technique specifically aimed at APIs. In a two-step approach, they first identify an API that the query is likely to refer to, and then expand the original query with identifier names related to this API. For example, for an initial query "how to save an image in png format," the approach may identify the API method `System.Drawing.Image.Save` to be likely relevant, and hence, adds the fully qualified method name into the search query.

Lift query to richer representation. To ease matching a query against potential search results, several approaches lift the query into a richer representation. An early example is the SCRUPLE tool by Paul and Prakash [95]. It transforms the query specified by the user with a pattern parser into an extended nondeterministic finite automaton called code pattern automaton. Other approaches lift queries into a graph representation. For example, Wang et al. [129] take a query formulated in a custom querying language and then transforms it into a graph representation that expresses call relations, control flow relations, and data flow relations. As another example, Li et al. [66] transform a natural language query into an "action relationship graph," which expresses sequencing, condition, and callback relationships between parts of the code described in the query. For example, given a query "add class 'checked' to element and fade in the element," their approach would infer that the two parts combined by "and" are supposed to happen in sequence.

Summary: The most popular techniques to modify queries are using weighing, adding, or replacing search terms, as well as lifting queries to a richer representation.

4 INDEXING OR TRAINING, FOLLOWED BY RETRIEVAL OF CODE

The perhaps most important component of a code search engine is about retrieving code examples relevant for a given query. The vast majority of approaches follows a two-step approach inspired by general information retrieval: At first, they either index the data to search through, e.g., by representing features of code examples in a numerical vector, or train a model that learns representations of the data to search through. Then, they retrieve relevant data items based on the pre-computed index or the trained model. To simplify the presentation, we refer to the first phase as "indexing" and mean both indexing in the sense of information retrieval and training a model on the data to search through.

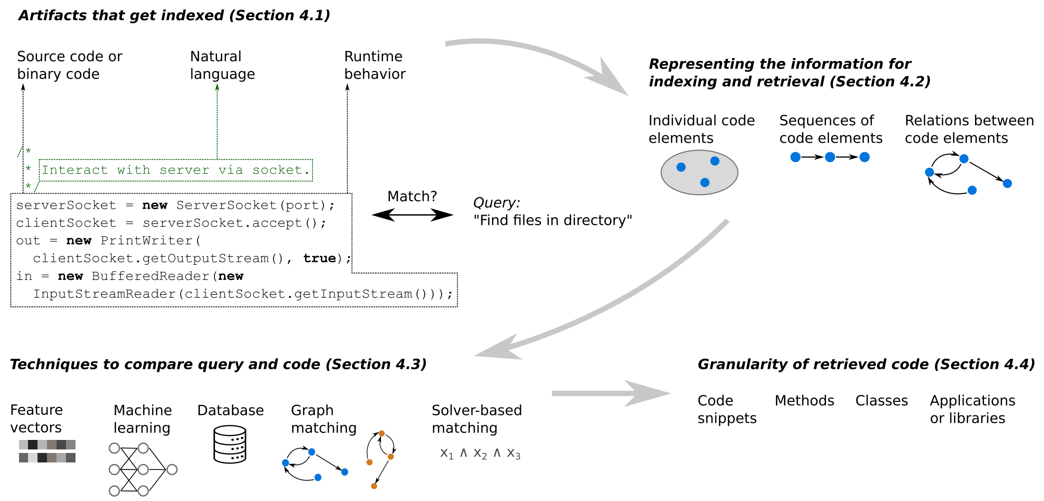
The primary goal of indexing and retrieval is effectiveness, i.e., the ability to find the "right" code examples for a query. To effectively identify these code examples, various ways of representing code and queries to compare them with each other have been proposed. A secondary goal, which is often at odds with achieving effectiveness, is efficiency. As users typically expect code search engines to respond within seconds [109], building an index that is fast to query is crucial. Moreover, as the code corpora to search through are continuously increasing in size, the scalability of both indexing and retrieval is important as well [4].

We survey the many different approaches to indexing, training and retrieval in code search engines along four dimensions, as illustrated in Figure 4. Section 4.1 discuss what kind of artifacts a search engine indexes. Section 4.2 describes different ways of representing the extracted information. Section 4.3 presents techniques for comparing queries and code examples with each other. Table 2 summarizes the approaches along these first three dimensions. Finally, Section 4.4 discusses different levels of granularity of the source code retrieved by search engines.

4.1 Artifacts That Get Indexed

When creating an index of code examples to retrieve, code search engines consider different artifacts related to code.

4.1.1 Source Code and Binary Code. The most obvious, and by far most prevalent, artifact to index is the source code itself. Many approaches target a high-level programming language, such as Java [5, 9, 22, 29, 40, 43, 62, 90, 120, 136, 138], JavaScript [61, 66], and C [94, 95]. Some search engines support not only one but multiple languages [16, 45, 111], e.g., Sando [113] (C, C++, and C#), cc-grep [47] (C, C++, Java, and Python), Aroma [75] (Hack, Java, JavaScript, and Python), DGMS [67] (Java and Python), and COSAL [81] (Java and Python). Instead of source code, some approaches focus on compiled code [21, 54], which is useful, e.g., to find functions in binaries that are similar


Artifacts that get indexed (Section 4.1)
Source code or binary code
Natural language
Runtime behavior
Match?
Query: "Find files in directory"
Representing the information for indexing and retrieval (Section 4.2)
Individual code elements
Sequences of code elements
Relations between code elements
Techniques to compare query and code (Section 4.3)
Feature vectors
Machine learning
Database
Graph matching
Solver-based matching
x1 ∧ x2 ∧ x3
Granularity of retrieved code (Section 4.4)
Code snippets
Methods
Classes
Applications or libraries

```
/*
 * Interact with server via socket.
 */
serverSocket = new ServerSocket(port);
clientSocket = serverSocket.accept();
out = new PrintWriter(
  clientSocket.getOutputStream(), true);
in = new BufferedReader(new
  InputStreamReader(clientSocket.getInputStream()));
```

Fig. 4. Overview of techniques for indexing and retrieval.

Table 2. Overview of Approaches based on Information Retrieval Technique Respect to Kind of Indexed Information (1–3 Rows) and Kind of Feature Extracted (4–6) Rows

	IR technique		
	Feature vectors	Machine learning	Other
Indexed artifact:			
Source code	[6, 9, 22, 29, 54, 61, 62, 75, 90, 91, 113, 118, 134, 138]	[45, 111, 119, 120]	[21, 40, 43, 47, 66, 78, 94, 95]
Runtime behavior	[97]	[49, 86, 98, 123]	
Natural language	[8, 15, 20, 55, 83, 126]	[16, 37, 68, 88, 108, 124]	[18, 92]
Representation of indexed code:			
Individual code elements	[22, 54]	[16, 20, 111]	[18, 47]
Sequences of code elements	[8, 55, 126, 134, 134]	[37]	[21]
Relationships between code elements	[9, 61, 62, 75, 76, 91]	[68, 86, 89, 124]	[10, 40, 43, 66, 94, 95]

to known vulnerable functions. These approaches first disassemble a given binary and then index disassembled functions or entire binaries.

4.1.2 *Runtime Behavior of Code.* Instead of only statically analyzing and indexing code, some search engines exploit the fact that source code can be executed by analyzing the runtime behavior of the code to search through. Considering runtime behavior may be useful, e.g., when two snippets of code have similar source code but nevertheless perform different behavior. The first code search engine that considers runtime behavior is by Podgurski and Pierce [97]. Their approach expects the user to provide inputs to the code to find and then searches for suitable code examples by sampling the behavior of candidate code examples. Reiss [106] select candidates using keywords and then they apply different kinds of transformations to have solutions with different behavior. To validate the dynamic behavior of the candidates with respect to the requirements given by the user, they run a set of test suites. Another line of work symbolically executes code to gather constraints that summarize the runtime behavior [49, 123]. Finally, the COSAL search engine [81] compares the behavior of code snippets based on an existing technique for clustering code based on its input–output behavior [80].

4.1.3 *Natural Language Information Associated with Code.*

Beyond the source code and its runtime behavior, another valuable artifact is natural language information associated with code. For example, such information comes in the form of comments, API documentation, commit messages, and discussions in online question-answer forums. Several code search engines leverage this information, in particular approaches that retrieve code based on natural language queries, e.g., after training neural models on pairs of natural language descriptions and source code [16, 124].

One group of approaches leverages regular comments and structured comments that provide API documentation, e.g., by considering comments as keywords to compare a query against [68] or by mapping code and natural language into a joint vector space [37, 108]. Another direction is to consider commit messages in a version control system, based on the assumption that the words in a commit message describe the source code lines affected by the commit [18]. Finally, online discussion forums, such as Stack Overflow,[7] provide a dataset of pairs of code snippets and natural language descriptions, which several code search engines use to associate natural language words and code [15, 20, 91].

Summary: The by far most common kind of artifact that gets indexed are source code and binary code. However, there also are search engines that index traces of runtime behavior and natural language information associated with code.

4.2 Representing the Information for Indexing and Retrieval

After discussing what artifacts different approaches extract information from, we now consider how this information is represented for indexing and retrieval. We identify three groups of approaches, presented in the following with increasing levels of complexity: representations based on individual code elements, on sequences of code elements, and on relations between code elements.

4.2.1 *Individual Code Elements.*

The first group of approaches focuses on individual code elements, e.g., tokens or function calls, ignoring their order and any other kind of relationship they may be in [18, 47]. To index the code examples, these approaches then represent a code snippet as a set of code elements. One example is work that represents a code example as a bag of tokens, and a natural language query as a bag of words [20, 126]. Another approach represents binaries as a set of tokens extracted from disassembled binaries [54]. Finally, Diamantopoulos et al. [22] represent API usages as a set of API calls, replacing each method by its type signature. The main benefit of indexing sets of individual code elements is the conceptual simplicity of the approach, which facilitates instantiating an idea for a particular target language. On the downside, the order of code elements and other kinds of relationships may provide useful information for precisely matching a code example against a query.

4.2.2 *Sequences of Code Elements.*

To preserve ordering information of code elements during the indexing, several approaches extract sequences of code elements from a given code example. Most commonly, these sequences are extracted in an Abstract Syntax Tree (AST) based, static analysis that focuses on particular kinds of nodes. For example, Gu et al. [37] represent API usages by extracting sequences of API calls from an AST. Another example is FaCoY [55], which represents a code example as a sequence of tokens extracted from an AST, where each token comes with a token type, e.g., method call or string literal. To represent incomplete code examples, they insert empty statements to complete the snippet. David and Yahav [21] instead use control flow graphs to

[7] https://stackoverflow.com/.

represent information for the binary source code. Sun et al. [124] represents code as a sequence of low-level instructions, which the approach obtains by compiling and then disassembling the code. Finally, deep learning-based code search approaches often tokenize source code using a sub-word tokenizer, such as the WordPiece [135] tokenizer used, e.g., by Salza et al. [111].

4.2.3 Relations between Code Elements. Going beyond individual code elements and sequences thereof, many approaches extract a richer set of relations between code elements. The most common approach is to focus on entities, typically code elements, such as classes, methods, and statements, and relations between them, such as one class inheriting from another class, one method calling another method, and a method containing a statement [9]. Popular examples of this approach include CodeQuest [40] and Sourcerer [68], which extract code elements and their relations through an AST-based analysis. Sourcerer also serves as the basis for other code search approaches, e.g., by Bajracharya et al. [8] and Lv et al. [76].

Sirres et al. [118] extract structural code entities of Java source files, collecting the relationship of imports, classes, methods, and variables. A more recent example is Aroma [75], which parses code into a simplified parse tree and then extracts different kinds of features based on the tokens in the code, parent–child relationships, sibling relationships, and variable usages, focusing. In a similar vein, Ling et al. [67] represent code as a graph that includes structural parent-child relationships, next-token relationships, and definition-use information. Li et al. [66] extract from ASTs three kinds of relationships between code elements: sequencing methods calls, callback between methods, and methods as conditions of if statements. Holmes and Murphy [43] use heuristics to collect relationships of methods inheritance, methods calls, and methods usage. Finally, Paul et al. [94, 95] use non-deterministic finite automata, called code pattern automata, to represent relationships between code elements.

Instead of a relatively lightweight static extraction of information to index, some search engines rely on more sophisticated static analysis. For example, Mishne et al. [86] propose a static type state analysis that extracts temporal specifications in the form of deterministic finite-state automata that capture sequences of API method calls. Another example is work by Premtoon et al. [100], which represent code examples as data flow graphs.

Summary: To index source code examples, code search engines typically represent the code as sets of individual code elements, sequences of code elements, or as relationships between code elements.

4.3 Techniques to Compare Queries and Code

After extracting the information from source code, execution behavior, and natural language information associated with the code, most search engines index the extracted information to then quickly respond to queries based on the pre-computed index. The following discusses different approaches for comparing queries and code, which we group into techniques based on feature vectors computed without machine learning (Section 4.3.1), machine learning-based techniques (Section 4.3.2), database-based techniques (Section 4.3.3), graph-based matching (Section 4.3.4), and solver-based matching (Section 4.3.5).

4.3.1 Indexing and Retrieval Based on Algorithmically Extracted Feature Vectors. Several techniques are based on feature vectors and distances between these vectors. In this sub-section we discuss approaches that compute feature vectors algorithmically, i.e., without any machine learning model. Their general idea is to represent both the code examples and the query as feature vectors, and to then retrieve code examples with a vector similar to that of the query. Because performing a pairwise comparison of the query vector with each code vector is inefficient, the approaches compute an index into the feature space that allows them to efficiently retrieve a ranked list of vectors similar to a given vector.

There are different ways of mapping information about code examples and queries into feature vectors. One approach is Boolean vectors [110] that express whether some feature, e.g., a particular type of AST node, are present [75]. Another common approach is to map a set of tokens or words into a **term frequency-inverse document frequency (TF-IDF)** vector, which expresses not only whether a feature is present, but also how important its presence is in comparison with other features [22, 119, 126, 134].

A popular implementation of feature-based indexing and retrieval is the Lucene library.[8] Originally designed for text search, Lucene is used in various code search engines [6, 8, 54, 55, 78, 98, 118]. It combines the Boolean model, which removes candidate vectors that do not provide the required features, and the vector space model, which computes a distance between the remaining candidate vectors and the query vector. The feature vectors are based on a custom term-frequency formula.[9] Moreover, Nguyen et al. [90] use a revised **Revised Vector Space Model (rVSM)**. The rVSM splits each token in separate words and computes the weight for each word using TF-IDF.

Instead of building upon an existing indexing and retrieval component, some search engines implement their own indexing and retrieval technique. For example, Lee et al. [62] use R*trees [10], which recursively partition the code examples into a tree structure that can then be used to efficiently find the nearest neighbors of a query. Luan et al. [75] identify those code examples that have the most overlap with the query vector by representing the feature set as a sparse vector and by then computing the overlap between queries and code examples via matrix multiplication. Another approach [9] matches a code query against code examples based on feature vectors for different AST subtrees of the code examples, pruning the large number of combinations to compare by considering only subtrees with the same parent node type.

4.3.2 Learning-based Retrieval. Neural software analysis [99] is becoming increasingly popular, and neural information retrieval [87] offers an attractive alternative to more traditional techniques. Most work takes an end-to-end neural learning approach, where a model learns to embed both queries and code examples into a joint vector space. Given this embedding, code search reduces to finding those code examples that are the nearest neighbors of a given query. We discuss approaches following this overall pattern in the following, focusing at first on natural language-to-code search and then on code-to-code search.

Learning-based natural language-to-code search. Gu et al. [37] pioneered with the first neural, end-to-end, natural language-to-code search engine. Their model embeds the code of methods using three submodels that apply recurrent neural networks to the name of the method, the API sequences in the method, and all tokens in the method body, respectively. Likewise, the model embeds the words in the query using another recurrent neural network. All embedding models are trained jointly to reduce the distance of matching code-query pairs while keeping unrelated pairs apart. In a similar way, Sun et al. [124] embed a code example and a natural language description into a joint vector space. They improve upon earlier work by translating the code into a natural languagelike representation based on transformation rules. Chen and Zhou [20] use two jointly trained auto encoders to map code and text into a vector space, respectively. Cambronero et al. [15] compare different ways to implement neural code search, including unsupervised [108] and supervised approaches and different neural models [37, 46]. Because a single model may not

[8]https://lucene.apache.org/.
[9]https://lucene.apache.org/core/3_5_0/scoring.html.

capture all aspects of a code example, Du et al. [24] propose an ensemble model that combines three neural code encoders, which focus on the structure of code, its variables, and its API usages, respectively.

To foster further comparisons, the CodeSearchNet challenge [45] offers a dataset of 2 million pairs of code and natural language queries, along with several neural baseline models and ElasticSearch.[10] Improvements on learning vector representations of code further improve the effectiveness of learning-based code search. For example, learn from multiple code representations [36], apply attention-based neural networks [136], or learn from a graph representation of code and queries via a graph neural network [67].

Learning-based code-to-code search. To find code based on an incomplete code example, several learning-based approaches have been proposed. One approach expands an incomplete code snippet using an LSTM-based language model and then searches for similar code snippets via a scalable clone detection technique [139]. An improved version of the approach [138] uses a library-sensitive language model for expanding the given code snippet. Another approach for retrieving code given an incomplete code snippet learns a model that predicts the probability that a complete code example fits the given snippet [88]. The model is based on various kinds of contextual information, e.g., the types, API calls, and code comments found around the given code snippet.

Search based on pre-trained models. Recent approaches use large pre-trained language models [27, 38], such as BERT [52], for code search. For example, Salza et al. [111] pre-train a BERT model on multiple programming languages and then they fine-tune the model using two encoders: one for natural language queries and another for code snippets. Chai et al. [16] show the value of transfer learning for code search by pre-training CodeBERT [27] on Java and Python, applying a meta-learning approach called Model-Agnostic Meta-Learning [28] to adapt the neural model to the target language, and finally fine-tuning the model with a dataset from the target language. Instead of an end-to-end neural search that maps entire code examples and queries into a joint vector space, one can also use pre-trained embeddings of individual words and tokens. For example, Ling et al. [67] use GloVe [96] and Zhou et al. [139] use pre-trained FastText embeddings.[11] Sachdev et al. [108] propose an approach that maps individual code tokens into vectors, then computes a TF-IDF-weighted average of them, and finally uses the resulting vector for a nearest neighbor-based search in the vector space.

4.3.3 Database-based Indexing and Retrieval. Given the success of databases for storing and retrieving information, several code search approaches build upon general-purpose databases. David et al. [21] describe a code search engine for binaries that stores short execution traces ("tracelets") in the NoSQL database MongoDB. Given a function as a query, the approach then retrieves other functions by querying the database for matching tracelets. Another database-based approach is by Hajiyev et al. [40], who build upon a relational database. Their approach stores facts extracted from a program, such as return relationships, method calls, and read and write fields, and then formulates search queries as database queries. In contrast to the similarity-based retrieval techniques discussed above, databases retrieve code examples that precisely match a query.

4.3.4 Graph-based Indexing and Retrieval. Given a graph representation of queries and code, another common approach is to retrieve code via graph-based matching. Li et al. [66] abstract both code snippets and a natural language query into graphs that represent different API method calls and their relationships. Then, they address the retrieval problem as a search for similar graphs. The

[10]https://www.elastic.co/elasticsearch/.
[11]https://fasttext.cc.

Yogo search engine [100] represents a given query code example and all code examples to search through as dataflow graphs. To match queries with code examples, the approach then applies a set of rewrite rules to check if the rewritten graphs match.

Instead of matching graphs, another direction is to use a graph representation of code to compute a similarity score. Mcmillan et al. [83]'s Portfolio technique first computes the pairwise similarity of a query and a set of functions, and then propagates the similarity score using the spreading activation algorithm through a pre-computed call graph. In an orthogonal step, the approach also computes the importance of every function by applying the page rank algorithm to the call graph. Finally, the two scores are combined to retrieve relevant functions. SCRUPLE [94, 95] uses a finite automata-based comparison of a code query and code examples. After turning both into a finite automata, a code pattern automaton interpreter compares two pieces of code and reports a match if the automaton reaches the final state.

4.3.5 Solver-based Matching. Code search engines that represent the behavior of code in the form of constraints often use Satisfiability Modulo Theories (SMT) solvers to match queries against code examples [49, 123]. The indexing phase in this case consists of a static analysis that extracts constraints describing input–output relationships. Then, the retrieval phase checks with an SMT solver whether the constraints of a code example satisfy the input–output examples that a user provides as the query. The idea was first proposed by Stolee et al. [123] and later refined and generalized by Jiang et al. [49].

Summary: The most used approaches for indexing and retrieval are feature vector-based retrieval and, more recently, deep learning-based models. The first approach needs less data and represents query and source code both as interpretable feature vectors. The second approach needs more data for training a model, e.g., to embed both queries and code source into a joint vector space.

4.4 Granularity of Retrieved Source Code

Different code search engines retrieve code at different levels of granularity. We categorize the existing approaches into four kinds of granularity. First, many search engines retrieve *code snippets*, which may range from a single line of code to multiple consecutive lines that implements a specific task. Second, other search engines focus on the *method*-level, i.e., these approaches retrieve entire methods. Third, users can also search at the *class*-level, where code search engines return entire classes. Finally, there also are search engines that operate at the *application or library*-level, which we do not cover in full detail here. Table 3 summarizes the approaches and the granularity they use. The same approach may appear in multiple rows [68, 92] if it supports multiple kinds of granularity.

The design decision of the granularity level to target is very important for a code search engine, because it affects what a user can search for. For example, snippets of code-level can be useful to search for code that provides an example of how to use an API [8]. The disadvantage of retrieving code snippets is that they may be incomplete and thus hard to directly reuse. Searching at the method-level can be useful for finding full methods that already solve a specific task [98], which a user may directly reuse. Finally, class-level and application or library-level approaches are useful to find entire components to reuse. Due to the more coarse-grained granularity, the number of suitable results may be limited though.

5 RANKING AND PRUNING OF SEARCH RESULTS

After retrieving code examples that likely match a query, many code search engines rank and prune the results before showing them to the user. This step is critical to enable users to quickly see the

Table 3. Granularity of Source Code Extracted by Code Search Approaches

Granularity	Approaches
Snippet of code	[8, 9, 16–18, 20–22, 29, 37, 43, 55, 55, 66, 68, 72, 75, 75, 86, 88, 91, 92, 94, 95, 100, 108, 111, 118, 121, 124, 128, 133, 137, 139]
Method	[45, 49, 63, 68, 73, 76, 83, 89, 90, 92, 97, 98, 106, 120, 123, 134]
Class	[68, 92, 106]
Application or library	[3, 5, 68, 92]

most relevant matches. In the following, we discuss and compare different ranking (Section 5.1) and pruning (Section 5.2) approaches.

5.1 Ranking of Search Results

5.1.1 Standard Distance Measures. The by far most common ranking approach is to rely on a distance measure implicitly provided by the retrieval component of a code search engine (Section 4). In this approach, the query and each code example are first represented as feature vectors, then a standard distance measure gives the distance between a query vector and a code vector, and finally code examples with a smaller distance to the query are ranked higher. For example, this ranking approach can be implemented using cosine similarity [15, 18, 20, 67, 108, 111, 124, 126], Hamming distance [8], and Euclidean distance [9, 61, 62].

5.1.2 Custom Ranking Techniques. In addition or as an alternative to standard distance measures, several search engines rely on custom ranking techniques. David and Yahav [21] propose a variation of string edit distance to compute the similarity between two sequences of assembly instructions. The basic idea is to treat each instruction as a letter and to use a table that provides a heuristic distance between assembly instructions. Another approach [66] ranks code examples using two scores that are based on the number of tokens that match the given natural language description and the length of a code snippet, respectively. Sachdev et al. [108] augment the rank obtained via cosine similarity with custom rules, such as the number of query tokens present in the candidate, to re-rank the list of results. Another example is from Lu et al. [73]. They compute a representative set of words for each method and then rank results via a normalized intersection of these words. COSAL [81] combines multiple custom ranking techniques, which compare two pieces of code based on their token similarity, structural similarity, and behavioral similarity, respectively.

Some ranking approaches look beyond the given query by also considering the code a developer is editing while making a query. For example, when building a query vector, Takuya and Masuhara [126] give more weight to occurrences of tokens near the cursor position, to find programs that contain similar fragments to the code around the cursor position. In a similar vein, Wightman et al. [133] uses features of the programmer's source code to rank and filter prospective snippet results, including variable types and names, the cursor position within the abstract syntax tree, and code dependencies. A higher rank here means that a code example uses more of the existing variables, and so on, and hence will require fewer modifications.

Some more recent ranking approaches are based on machine learning models. For example, the Lancer approach [138] fine-tunes a pre-trained BERT model[12] to predict whether a code example matches the given, incomplete method, and then ranks code examples based on the predicted

[12]https://github.com/huggingface/pytorch-pretrained-BERT.

score. Ye et al. [137] compute the similarity score using two parameters retrieved with a code summarization model and a code generation model, based on a dual learning technique.

Summary: To rank search results, engines often use standard algorithms, such as cosine similarity and Euclidean distance, or they implement custom variations of these techniques.

5.2 Pruning of Search Results

Orthogonal to ranking, several search engines also prune search results that are unlikely to be of interest to the user. The most straightforward pruning technique is to discard results based on similarity threshold. For example, some approaches discard all candidates with a similarity lower than some threshold [17, 49], while others show only the top N results in the output [55, 72, 103]. Another way of pruning search results is to merge similar code examples, assuming that a user likely wants to see only one of them. For example, Mishne et al. [86] merge similar method call paths relevant to the query to remove redundancy in the final results. Aroma [75] uses a greedy algorithm based on parse tree comparison to find and remove redundant code snippets, followed by re-ranking the pruned search results.

Summary: Filtering by a threshold and merging similar results are the most popular techniques for pruning code search results.

6 EMPIRICAL STUDIES OF CODE SEARCH

The wide adoption of code search in practice raises various interesting questions about the way developers search for code. This section discusses empirical studies related to how, when, and why developers search for code and what tools they use for this purpose. We include all such empirical studies that we are aware of and that fit the selection criteria given in Section 1. We start by describing the experimental setups used in these studies (Section 6.1) and then present some of their main results (Section 6.2). Table 4 gives an overview of the discussed studies, including the topics they address, the methodologies they use, and the amount of code searched through by the studied developers.

6.1 Setups of Empirical Studies

While practically all empirical studies address the broad questions of how, when, and why developers search for code, they use different setups and methodologies to address this question. Early studies [56, 117] are mostly about what activities developers spend their time on and what tools they use, including tools used for code search. In contrast, more recent studies [7, 93, 102, 109, 114, 115] focus specifically on code search tools and what activities they are used for.

We see three kinds of methodologies, and sometimes combinations of them: questionnaires answered by developers [56, 114, 117], analyses of logs of search engines [7, 93, 102, 109, 115, 117], and observing developers, e.g., by shadowing them [117] or by recording their screens [56]. The first two kinds of studies are typically based on data gathered from tens [56, 114, 115] to hundreds [109] of developers. In contrast, log analysis often covers much larger datasets, ranging between tens of thousands [93] and 10 million [7] logged activities.

The studies also vary by the amount of code that is searched through by the studied developers. Reflecting the general trends in code search engines, early studies are about searching through a single project [56, 114, 117], whereas later studies are about searching through multiple projects, either within a larger organization [93, 109] or the open source ecosystem [7, 102, 115].

Table 4. Overview of Empirical Studies on Code Search

	Singer et al. [117]	Sim et al. [114]	Ko et al. [56]	Sim et al. [115]	Panchenko et al. [93]	Bajracharya and Lopes [7]	Sadowski et al. [109]	Rahman et al. [102]
Topic of study:								
Usage of development tools	✓		✓					
Usage of search tools		✓		✓	✓	✓	✓	✓
Activities of developers	✓	✓	✓				✓	
Methodology:								
Questionnaire	✓	✓	✓				✓	
Log analysis	✓			✓	✓	✓	✓	✓
Observing developers	✓		✓					
Searched code:								
Single project	✓	✓	✓					
Multiple projects within organization	✓				✓		✓	
Many open source projects				✓		✓		✓

6.2 Results of Studies and Their Implications

A recurring finding in studies is that code search is among the most common activities developers spend their time on. Early studies report that *grep*, *find*, and its variants are used on a regular basis [114, 117]. For example, measurements of tool invocations by Singer et al. [117] shows that *grep* and its variants are the second-most frequently used developer tools, right after the compiler. The observational study by Ko et al. [56] also reports code search to be a common activity. However, their definition of "searching for code" only partially matches ours, because we assume that there is an explicitly formulated query, whereas they also mean reading code to find a specific code location. A study at Google based on a specialized code search engine shows that the average developer is involved in five search sessions per day, with a total of 12 daily queries [109]. Overall, these findings highlight the importance of code search in software development, motivating researchers and practitioners to work on code search techniques.

Several studies investigate the goals that developers have when searching for code. The three most commonly reported goals are finding example code to reuse, e.g., when trying to understanding how to use an API (between 15% [114] and 34% [109] of all searches), program understanding (between 14% [114] and 29% [109] of all searches), and understanding and fixing a bug (between 10% [109] and 20% [114] of all searches). Beyond these three goals, a long tail of other goals is reported, such as understanding the impact of a planned code change, finding locations relevant for a code clean-up, understanding the coding style used within an organization, and identifying the person responsible for a particular piece of code. A perhaps surprising finding is that developers also often use code search as a quick way to navigate through code they are already familiar with [109].

Being a central element of every search, queries and their properties have received some attention in studies. A study of the Koders code search engine finds most queries to be short, with 79% of users providing only a single search term [7]. In contrast, other studies report longer queries, e.g., an average of 4.2 terms per query in a study across five search engines [115], and of 4.7 terms for code-related queries given to Google's general-purpose search engine [102]. Beyond the size of queries, several studies investigate what terms are used in queries. Bajracharya and Lopes [7] find that both code queries and natural language queries are common. Comparing code-related queries with general-purpose web search queries, Rahman et al. [102] find that code queries use a smaller vocabulary. Another interesting finding related to queries is that they are frequently reformulated within a search session [7, 109, 115], even more often than general web search queries [102].

Finally, some studies analyze and compare how effective code search engines are at providing useful search results. One study reports that between 25% and 60% of all queries are effective, depending on the kind of query, where "effective" means that the search results cause the user to download a relevant piece of code [7]. Another study compares specialized code search engines with general-purpose web search engines. It finds that the former are more effective when searching for entire subsystems, e.g., a library to use, whereas the latter are more effective for finding individual blocks of code [115]. The same study also reports that it is easier to find referenceexamples than components that are reusable as-is.

Summary: Empirical studies of developers show that they commonly perform code search to reach various goals, including code understanding, finding code to reuse, and quickly navigating to code the developer already knows.

7 OPEN CHALLENGES AND RESEARCH DIRECTIONS

7.1 Support for Additional Usage Scenarios

Each code search engine focuses on one or more usage scenarios, such as finding examples of how to use a specific API or finding again some code that a developer has previously worked on. In addition to the currently supported usage scenarios, we envision future work to support other search-related developer tasks. For example, developers may want to search not only through a static snapshot of code, but also search for specific kinds of changes in the version histories of projects. Searching for changes could help developers, e.g., to understand how a particular API usage typically evolves, to find examples of code changes similar to a change a developer is currently working on, or to find code changes that have introduced bugs. Lawall et al. [57] and Di Grazia et al. [34] propose promising first steps into this direction. Another example of a currently unsupported usage scenario is cross-language search. In this scenario, a user formulates a code query in one programming language to find related code written in another programming language. Such cross-language search could help developers transfer their knowledge across languages, e.g., when a developer knows how to implement a particular functionality in one but not in another language.

7.2 Cross-fertilization with Code Completion and Clone Detection

Code search relates to other problems that have received significant attention by researchers and that offer opportunities for cross-fertilization. One such problem is code completion, i.e., the problem of suggesting suitable code snippets while the developer is writing code in an IDE. Recent large-scale language models used for code completion offer a functionality similar to code search. For example, a typical usage scenario of GitHub's Copilot tool[13] and the underlying Codex model [19] takes a short natural language description of a desired functionality and maps it to a code snippet offering that functionality. This usage scenario is closely related to code search engines that receive free-form queries (Section 2.1). It remains an open challenge to apply successful techniques from

[13]https://github.com/features/copilot.

code search in code completion, and vice versa. Another problem that is strongly related to code search is clone detection [107]. Similar to code search engines that accept programming language queries (Section 2.2), clone detectors try to find code that is similar to a given code example. A key difference is that clone detection tries to find multiple code examples that implement the same functionality (possibly with syntactic and semantic differences that do not affect the overall behavior), whereas code search tries to retrieve code that offers more functionality than the given query. Despite these different goals, there is potential for cross-fertilization of the two related fields, e.g., by adapting effective representations of code (Section 4) or mechanisms for pruning search results (Section 5.2).

7.3 Learning-based Code Search

Given the tremendous progress in machine learning, adopting the newest models to code search is likely to offer new opportunities to code search. In particular, we identify three open challenges. First, future work could benefit from the increasingly effective code representation models proposed in the neural software analysis field [99] to compute a vector representation of code and of queries, which can then be used to identify code examples similar to a given query. Some instances of this idea have already been presented [37], but as code representation models keep increasing, adopting new models is likely to also improve code search. Second, future work could design models that not only retrieve code, but also generalizing examples seen during training into new code that fits a query. Open challenges here include to formulate code search as a zero-shot learning problem [13] and to adapt models that combine question answering with retrieval [60].

7.4 Deployment and Adoption in Practice

Code search is an area of strong interest by academic researchers, tool builders in industry, and practitioners. Despite the already impressive use of code search by developers, we see several open challenges related to its deployment and adoption in practice. On the one hand, there are challenges faced by people who are running and maintaining a code search engine. For example, the problem of how to incrementally re-index a code corpus when the code is evolving has not yet received significant attention by researchers. A naive approach is to continuously re-index the entire corpus, which is likely to unnecessarily repeat significant computational effort.

On the other hand, there are challenges faced by users of code search engines. While various techniques have been proposed for the core components of code search, its user interface is receiving less attention, with some noteworthy exceptions, such as some of the query expansion techniques discussed in Section 3. An interesting line of future work could be to automatically formulate clarification questions, e.g., in natural language, which could allow a user to prune the search space with a single click. Another promising direction is to support users in defining code queries (Section 2) by adding automatic code completion features known from IDEs into the interface of a search engine. Finally, future work could design "query-less" search engines that suggest suitable code snippets while a developer is writing code, without the need to explicitly formulate a query. First steps toward that goal have been taken, e.g., by Brandt et al. [11] and Takuya and Masuhara [126].

7.5 Common Datasets and Benchmarks

An effective way to foster further progress in the research field is to offer reusable datasets for evaluating and comparing code search engines. Ideally, such a dataset should be realistic, large scale, and cover multiple programming languages. Several benchmark datasets have been proposed and are use by parts of the existing work. One kind of benchmarks consists of groups of semantically equivalent implementations, e.g., BigCloneBench [125], Google Code Jam,[14] and AtCoder.[15] Such benchmarks are particularly useful to evaluate code-to-code search engines (Section 2.2), as one implementation in a group can be used as a query, while the other implementations are expected to show up among the results. Benchmarks that come with executable test cases, e.g., those derived from coding competitions, are also useful to evaluate approaches based on dynamic analysis (Section 4.1.2).

Another kind of benchmarks offers pairs of natural language queries and code. For example, the CodeSearchNet challenge offers such a dataset, which has been automatically gathered and covers Go, Java, JavaScript, PHP, Python, and Ruby code [45]. In a similar vein, CodeXGLUE offers query-code pairs for Python and Java [74]. The Search4Code dataset provides code-related queries extracted from Bing search queries via a weakly supervised discriminative model [104]. Instead of relying on automated extraction of datasets, CoSQA is a benchmark of pairs of natural language queries and code examples that have been manually annotated [44].

We anticipate future work to build even more than today upon these datasets, either as a benchmark to evaluate a novel code search engine, or as a training dataset to learn from. There also are opportunities for creating datasets and benchmarks that go beyond those available today. For example, the community would benefit from a dataset that not only includes queries and search results, but also information on how developers act on search results, e.g., by selecting lower-ranked results or by copying and adapting code examples. An interesting challenge for benchmarks used to evaluate learning-based code search is how to ensure that a model does not see the benchmark during learning. As large-scale, pre-trained models [19, 27, 38], which often are trained on a large fraction of all publicly available source code, are becoming increasingly popular, the chances that a publicly available benchmark is coincidentally used during training increases.

8 CONCLUDING REMARKS

This article provides a comprehensive overview of 30 years of research on code search. Given the huge amounts of existing code, searching for specific code examples is a common activity during software development. To support developers during this activity, various techniques for finding relevant code have been proposed, with an increase of interest during recent years. We discuss what kinds of queries code search engines support, and give an overview of the main components used to retrieve suitable code examples. In particular, the article discusses techniques to pre-process and expand queries, approaches toward indexing and retrieving code, and ways of pruning and ranking search results. Our article enables readers to obtain an overview of the field, or to fill in gaps of their knowledge of the state of the art. Based on our survey of past work, we conclude that code search has evolved into a mature research field, with solid results that have already made an impact on real-world software development. Despite all advances, many open challenges remain to be addressed in the future, and we hope our article will provide a useful starting point for addressing them.

REFERENCES

[1] 1998. *ISO/IEC 14882 International Standard - First Edition 1998-09-01: Programming Languages C++*. ISO.
[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques, & Tools.* Pearson Education India.
[3] S. Akbar and A. Kak. 2019. SCOR: Source code retrieval with semantics and order. In *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR'19).* 1–12.

[14]https://codingcompetitions.withgoogle.com/codejam.
[15]https://atcoder.jp/.

[4] Artem Babenko and Victor Lempitsky. 2016. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2055–2063.

[5] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. 2006. Sourcerer: A search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. 681–682.

[6] Sushil Krishna Bajracharya and Cristina Videira Lopes. 2009. Mining search topics from a code search engine usage log. In *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR 2009) (Co-located with ICSE)*, Michael W. Godfrey and Jim Whitehead (Eds.). IEEE Computer Society, 111–120. https://doi.org/10.1109/MSR.2009.5069489

[7] Sushil Krishna Bajracharya and Cristina Videira Lopes. 2012. Analyzing and mining a code search engine usage log. *Empir. Softw. Eng.* 17, 4–5 (2012), 424–466. https://doi.org/10.1007/s10664-010-9144-6

[8] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. 2010. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*. Association for Computing Machinery, New York, NY, 157–166. https://doi.org/10.1145/1882291.1882316

[9] V. Balachandran. 2015. Query by example in large-scale code repositories. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*. 467–476.

[10] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 322–331.

[11] 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 28th International Conference on Human Factors in Computing Systems (CHI'10)*, Elizabeth D. Mynatt, Don Schoner, Geraldine Fitzpatrick, Scott E. Hudson, W. Keith Edwards, and Tom Rodden (Eds.). ACM, 513–522. https://doi.org/10.1145/1753326.1753402

[12] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.* 30, 1–7 (1998), 107–117.

[13] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems (NeurIPS'20)*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.).

[14] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *Proceedings of the European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE'09)*. ACM, 213–222.

[15] José Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE'19)*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 964–974. https://doi.org/10.1145/3338906.3340458

[16] Yitian Chai, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Cross-domain deep code search with meta learning. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. Association for Computing Machinery, New York, NY, 487–498. https://doi.org/10.1145/3510003.3510125

[17] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. 2009. Sniff: A search engine for java using free-form queries. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 385–400.

[18] Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang, and Amir Michail. 2001. CVSSearch: Searching through source code using CVS comments. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE, 364–373.

[19] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. CoRR abs/2107.03374 (2021). arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374.

[20] Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. Association for Computing Machinery, New York, NY, 826–831. https://doi.org/10.1145/3238147.3240471

[21] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *SIGPLAN Not.* 49, 6 (June 2014), 349–360. https://doi.org/10.1145/2666356.2594343

[22] Themistoklis Diamantopoulos, Georgios Karagiannopoulos, and Andreas L. Symeonidis. 2018. Codecatch: Extracting source code snippets from online sources. In *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE'18)*. Association for Computing Machinery, New York, NY, 21–27. https://doi.org/10.1145/3194104.3194107

[23] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: A taxonomy and survey. *J. Softw.: Evol. Process* 25, 1 (2013), 53–95.

[24] Lun Du, Xiaozhou Shi, Yanlin Wang, Ensheng Shi, Shi Han, and Dongmei Zhang. 2021. Is a single model enough? MuCoS: A multi-model ensemble learning approach for semantic code search. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management (CIKM'21)*, Gianluca Demartini, Guido Zuccon, J. Shane Culpepper, Zi Huang, and Hanghang Tong (Eds.). ACM, 2994–2998. https://doi.org/10.1145/3459637.3482127

[25] Frederico A. Durão, Taciana A. Vanderlei, Eduardo S. Almeida, and Silvio R. de L. Meira. 2008. Applying a semantic layer in a source code search tool. In *Proceedings of the ACM Symposium on Applied Computing (SAC'08)*. Association for Computing Machinery, New York, NY, 1151–1157. https://doi.org/10.1145/1363686.1363952

[26] ECMA. 2011. Standard ECMA-262, ECMAScript Language Specification, 5.1 Edition.

[27] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics (EMNLP 2020)* Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[28] Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 1126–1135.

[29] Y. Fujiwara, N. Yoshida, E. Choi, and K. Inoue. 2019. Code-to-code search based on deep neural network and code mutation. In *Proceedings of the IEEE 13th International Workshop on Software Clones (IWSC'19)*. 1–7.

[30] Yang Fuqing, Mei Hong, and Li Keqin. 1999. Software reuse and software component technology [J]. *Acta Electr. Sin.* 27 (1999), 68–75.

[31] V. C. Garcia, E. S. de Almeida, L. B. Lisboa, A. C. Martins, S. R. L. Meira, D. Lucredio, and R. P. d. M. Fortes. 2006. Toward a code search engine based on the state-of-art and practice. In *Proceedings of the 13th Asia Pacific Software Engineering Conference (APSEC'06)*. 61–70.

[32] GitHub. 2021. The 2020 State of the Octoverse. Retrieved from https://octoverse.github.com/.

[33] James Gosling, William N. Joy, and Guy L. Steele Jr. 1996. *The Java Language Specification*. Addison-Wesley.

[34] Luca Di Grazia, Paul Bredl, and Michael Pradel. 2022. DiffSearch: A scalable and precise search engine for code changes. arXiv:2204.02787. Retrieved from https://arxiv.org/abs/2204.02787.

[35] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad M. Cumby. 2010. A search engine for finding highly relevant applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Volume 1 (ICSE'10)*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 475–484. https://doi.org/10.1145/1806799.1806868

[36] Jian Gu, Zimin Chen, and Martin Monperrus. 2021. Multimodal representation for neural code search. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'21)*. IEEE, 483–494. https://doi.org/10.1109/ICSME52107.2021.00049

[37] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE'18)*.

[38] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *Proceedings of the 9th International Conference on Learning Representations (ICLR'21)*.

[39] Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java expressions from free-form queries. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 416–432. https://doi.org/10.1145/2814270.2814295

[40] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. 2006. *codeQuest:* Scalable source code queries with datalog. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06), Lecture Notes in Computer Science*, Vol. 4067, Dave Thomas (Ed.). Springer, 2–27. https://doi.org/10.1007/11785477_2

[41] E. Hill, L. Pollock, and K. Vijay-Shanker. 2009. Automatically capturing source code context of NL-queries for software maintenance and reuse. In *Proceedings of the IEEE 31st International Conference on Software Engineering*. 232–242.

[42] Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*. IEEE, 524–527.

[43] Reid Holmes and Gail C. Murphy. 2005. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. Association for Computing Machinery, New York, NY, 117–125. https://doi.org/10.1145/1062455.1062491

[44] Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. CoSQA: 20,000+ web queries for code search and question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, (ACL/IJCNLP'21), Volume 1: Long Papers*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, 5690–5700. https://doi.org/10.18653/v1/2021.acl-long.442

[45] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. arXiv:1909.09436. Retrieved from http://arxiv.org/abs/1909.09436.

[46] Hamel Husain and Ho-Hsiang Wu. 2018. How to create natural language semantic search for arbitrary objects with deep learning. *Retrieved November* 5 (2018), 2019.

[47] Katsuro Inoue, Yuya Miyamoto, Daniel M. Germán, and Takashi Ishio. 2020. Code clone matching: A practical and effective approach to find code snippets. arXiv:2003.05615. Retrieved from https://arxiv.org/abs/2003.05615.

[48] Doug Janzen and Kris De Volder. 2003. Navigating and querying code without getting lost. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, William G. Griswold and Mehmet Aksit (Eds.). ACM, 178–187. https://doi.org/10.1145/643603.643622

[49] Renhe Jiang, Zhengzhao Chen, Zejun Zhang, Yu Pei, Minxue Pan, and Tian Zhang. 2018. [Research Paper] semantics-based code search using input/output examples. In *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'18)*. IEEE Computer Society, 92–102. https://doi.org/10.1109/SCAM.2018.00018

[50] Huzefa H. Kagdi, Michael L. Collard, and Jonathan I. Maletic. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Res. Pract.* 19, 2 (2007), 77–131. https://doi.org/10.1002/smr.344

[51] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search (t). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE, 295–306.

[52] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. 4171–4186.

[53] Muhammad Khalifa. 2019. Semantic source code search: A study of the past and a glimpse at the future. arXiv:1908.06738. Retrieved from http://arxiv.org/abs/1908.06738.

[54] W. M. Khoo, A. Mycroft, and R. Anderson. 2013. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*. 329–338.

[55] 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 *Proceedings of the 40th International Conference on Software Engineering*. 946–957.

[56] A. J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.* 32, 12 (2006), 971–987. https://doi.org/10.1109/TSE.2006.116

[57] Julia Lawall, Derek Palinski, Lukas Gnirke, and Gilles Muller. 2017. Fast and precise retrieval of forward and back porting information for Linux device drivers. In *Proceeding of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. 15–26.

[58] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 years of automated evolution in the linux kernel. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'18)*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 601–614.

[59] Claudia Leacock and Martin Chodorow. 1998. Combining local context and WordNet similarity for word sense identification. *WordNet Electr. Lexic. Datab.* 49, 2 (1998), 265–283.

[60] Kenton Lee, Ming-Wei Chang, and Kristina Toutanova. 2019. Latent retrieval for weakly supervised open domain question answering. In *Proceedings of the 57th Conference of the Association for Computational Linguistics (ACL'19)*, Anna Korhonen, David R. Traum, and Lluís Màrquez (Eds.). Volume 1: Long Papers, Association for Computational Linguistics, Florence, Italy, July 28- August 2, 2019, 6086–6096. https://doi.org/10.18653/v1/p19-1612

[61] M. Lee, S. Hwang, and S. Kim. 2011. Integrating code search into the development session. In *Proceedings of the IEEE 27th International Conference on Data Engineering*. 1336–1339.

[62] Mu-Woong Lee, Jong-Won Roh, Seung-won Hwang, and Sunghun Kim. 2010. Instant code clone search. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*. Association for Computing Machinery, New York, NY, 167–176. https://doi.org/10.1145/1882291.1882317

[63] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Paulo Cesar Masiero, and Cristina Videira Lopes. 2011. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Inf. Softw. Technol.* 53, 4 (2011), 294–306. https://doi.org/10.1016/j.infsof.2010.11.009

[64] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, Joel Ossher, Ricardo Santos Morla, Paulo Cesar Masiero, Pierre Baldi, and Cristina Videira Lopes. 2007. CodeGenie: Using test-cases to search and reuse source code. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer (Eds.). ACM, 525–526. https://doi.org/10.1145/1321631.1321726

[65] Wei Li, Shuhan Yan, Beijun Shen, and Yuting Chen. 2019. Reinforcement learning of code search sessions. In *Proceedings of the 26th Asia-Pacific Software Engineering Conference (APSEC'19)*. IEEE, 458–465. https://doi.org/10.1109/APSEC48747.2019.00068

[66] 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 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. Association for Computing Machinery, New York, NY, 690–701. https://doi.org/10.1145/2950290.2950341

[67] Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. 2021. Deep graph matching and searching for semantic code retrieval. *ACM Trans. Knowl. Discov. Data* 15, 5 (2021), 88:1–88:21. https://doi.org/10.1145/3447571

[68] Erik Linstead, Sushil Krishna Bajracharya, Trung Chi Ngo, Paul Rigor, Cristina Videira Lopes, and Pierre Baldi. 2009. Sourcerer: Mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.* 18, 2 (2009), 300–336. https://doi.org/10.1007/s10618-008-0118-x

[69] Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John C. Grundy. 2022. Opportunities and challenges in code search tools. *ACM Comput. Surv.* 54, 9 (2022), 196:1–196:40. https://doi.org/10.1145/3480027

[70] Jason Liu, Seohyun Kim, Vijayaraghavan Murali, Swarat Chaudhuri, and Satish Chandra. 2019. Neural query expansion for code search. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL'19)*. Association for Computing Machinery, New York, NY, 29–37. https://doi.org/10.1145/3315508.3329975

[71] Wenjian Liu, Xin Peng, Zhenchang Xing, Junyi Li, Bing Xie, and Wenyun Zhao. 2018. Supporting exploratory code search with differencing and visualization. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 300–310. https://doi.org/10.1109/SANER.2018.8330218

[72] J. Lu, Y. Wei, X. Sun, B. Li, W. Wen, and C. Zhou. 2018. Interactive query reformulation for source-code search with word relations. *IEEE Access* 6 (2018), 75660–75668.

[73] Meili Lu, Xiaobing Sun, Shaowei Wang, David Lo, and Yucong Duan. 2015. Query expansion via wordnet for effective code search. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER'15)*. IEEE, 545–549.

[74] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *CoRR* abs/2102.04664.

[75] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proc. ACM Program. Lang.* 3 (2019), 152.

[76] 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 *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. 260–270.

[77] Lee Martie, André van der Hoek, and Thomas Kwak. 2017. Understanding the impact of support for iteration on code search. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. Association for Computing Machinery, New York, NY, 774–785. https://doi.org/10.1145/3106237.3106293

[78] Lee Martie, Thomas D. LaToza, and Andre van der Hoek. 2015. Codeexchange: Supporting reformulation of internet-scale code queries in context (t). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE, 24–35.

[79] L. Martie and A. v. d. Hoek. 2015. Sameness: An experiment in code search. In *Proceedings of the IEEE/ACM 12th Working Conference on Mining Software Repositories*. 76–87.

[80] George Mathew, Chris Parnin, and Kathryn T. Stolee. 2020. SLACC: Simion-based language agnostic code clones. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 210–221. https://doi.org/10.1145/3377811.3380407

[81] George Mathew and Kathryn T. Stolee. 2021. Cross-language code search using static and dynamic analyses. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ISEC/FSE'21)*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 205–217. https://doi.org/10.1145/3468264.3468538

[82] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. 2012. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Trans. Softw. Eng.* 38, 5 (2012), 1069–1087.

[83] 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 Trans. Softw. Eng. Methodol.* 22, 4, Article 37 (October 2013), 30 pages. https://doi.org/10.1145/2522920.2522930

[84] Amir Michail. 2002. Browsing and searching source code of applications written using a GUI framework. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*. Association for Computing Machinery, New York, NY, 327–337. https://doi.org/10.1145/581339.581381

[85] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*. 3111–3119.

[86] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. Association for Computing Machinery, New York, NY, 997–1016. https://doi.org/10.1145/2384616.2384689

[87] Bhaskar Mitra, Nick Craswell, et al. 2018. *An Introduction to Neural Information Retrieval*. Now Foundations and Trends.

[88] Rohan Mukherjee, Swarat Chaudhuri, and Chris Jermaine. 2020. Searching a database of source codes using contextualized code search. *Proc. VLDB Endow.* 13, 10 (June 2020), 1765–1778. https://doi.org/10.14778/3401960.3401972

[89] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. 2016. API code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. Association for Computing Machinery, New York, NY, 511–522. https://doi.org/10.1145/2950290.2950333

[90] T. V. Nguyen, A. T. Nguyen, H. D. Phan, T. D. Nguyen, and T. N. Nguyen. 2017. Combining Word2Vec with revised vector space model for better code retrieval. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C'17)*. 183–185.

[91] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li. 2016. Query expansion based on crowd knowledge for code search. *IEEE Trans. Serv. Comput.* 9, 5 (2016), 771–783.

[92] Joel Ossher, Sushil Krishna Bajracharya, Erik Linstead, Pierre Baldi, and Cristina Videira Lopes. 2009. SourcererDB: An aggregated repository of statically analyzed and cross-linked open source Java projects. In *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR'09)*, Michael W. Godfrey and Jim Whitehead (Eds.). IEEE Computer Society, 183–186. https://doi.org/10.1109/MSR.2009.5069501

[93] Oleksandr Panchenko, Hasso Plattner, and Alexander Zeier. 2011. What do developers search for in source code and why. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*. 33–36.

[94] Santanu Paul. 1992. SCRUPLE: A reengineer's tool for source code search. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, Volume 1 (CASCON'92)*. IBM Press, 329–346.

[95] Santanu Paul and Atul Prakash. 1994. A framework for source code search using program patterns. *IEEE Trans. Softw. Eng.* 20, 6 (1994), 463–475.

[96] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP'14)*. 1532–1543.

[97] Andy Podgurski and Lynn Pierce. 1993. Retrieving reusable software by sampling behaviour. *ACM Trans. Softw. Eng. Methodol.* 2, 3 (1993), 286–303. https://doi.org/10.1145/152388.152392

[98] D. Poshyvanyk and M. Grechanik. 2009. Creating and evolving software by searching, selecting and synthesizing relevant source code. In *Proceedings of the 31st International Conference on Software Engineering–Companion Volume*. 283–286.

[99] Michael Pradel and Satish Chandra. 2022. Neural software analysis. *Commun. ACM* 65, 1 (2022), 86–96. https://doi.org/10.1145/3460348

[100] Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic code search via equational reasoning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*.

[101] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: Synthesizing what i mean: Code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16).* Association for Computing Machinery, New York, NY, 357–367. https://doi.org/10.1145/2884781.2884808

[102] Md. Masudur Rahman, Jed Barson, Sydney Paul, Joshua Kayan, Federico Andres Lois, Sebastian Fernandez Quezada, Christopher Parnin, Kathryn T. Stolee, and Baishakhi Ray. 2018. Evaluating how developers use general-purpose web-search for code retrieval. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*, Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.). ACM, 465–475. https://doi.org/10.1145/3196398.3196425

[103] M. M. Rahman and C. Roy. 2018. Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'18).* 473–484.

[104] Nikitha Rao, Chetan Bansal, and Joe Guan. 2021. Search4Code: Code search intent classification using weak supervision. In *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR'21).* IEEE, 575–579. https://doi.org/10.1109/MSR52588.2021.00077

[105] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14).* 44.

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

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

[108] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: A neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages.* ACM, 31–41.

[109] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How developers search for code: A case study. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering.* 191–201.

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

[111] Pasquale Salza, Christoph Schwizer, Jian Gu, and Harald C. Gall. 2022. On the effectiveness of transfer learning for code search. *IEEE Trans. Softw. Eng.* (2022).

[112] Tom Seymour, Dean Frantsvog, Satheesh Kumar, et al. 2011. History of search engines. *Int. J. Manage. Inf. Syst.* 15, 4 (2011), 47–58.

[113] David Shepherd, Kostadin Damevski, Bartosz Ropski, and Thomas Fritz. 2012. Sando: An extensible local code search framework. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'12).* Association for Computing Machinery, New York, NY, Article 15, 2 pages. https://doi.org/10.1145/2393596.2393612

[114] Susan Elliott Sim, Charles L. A. Clarke, and Richard C. Holt. 1998. Archetypal source code searches: A survey of software developers and maintainers. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC'98).* IEEE Computer Society, 180–187. https://doi.org/10.1109/WPC.1998.693351

[115] Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V. Lopes. 2011. How well do search engines support code retrieval on the web? *ACM Trans. Softw. Eng. Methodol.* 21, 1 (2011), 4:1–4:25. https://doi.org/10.1145/2063239.2063243

[116] Renuka Sindhgatta. 2006. Using an information retrieval system to retrieve source code samples. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06).* Association for Computing Machinery, New York, NY, 905–908. https://doi.org/10.1145/1134285.1134448

[117] Janice Singer, Timothy C. Lethbridge, Norman G. Vinson, and Nicolas Anquetil. 1997. An examination of software engineering work practices. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, J. Howard Johnson (Ed.). IBM, 21.

[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. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 945. https://doi.org/10.1145/3180155.3182513

[119] B. Sisman and A. C. Kak. 2013. Assisting code search with automatic Query Reformulation for bug localization. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13).* 309–318.

[120] Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, and Miryung Kim. 2019. Active inductive logic programming for code search. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE/ACM, 292–303. https://doi.org/10.1109/ICSE.2019.00044

[121] Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the search for source code. *ACM Trans. Softw. Eng. Methodol.* 23, 3, Article 26 (June 2014), 45 pages. https://doi.org/10.1145/2581377

[122] Kathryn T. Stolee and Sebastian G. Elbaum. 2012. Toward semantic search via SMT solver. In *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT/FSE'12)*, Will Tracz, Martin P. Robillard, and Tevfik Bultan (Eds.). ACM, 25. https://doi.org/10.1145/2393596.2393625

[123] Kathryn T. Stolee, Sebastian G. Elbaum, and Matthew B. Dwyer. 2016. Code search with input/output queries: Generalizing, ranking, and assessment. *J. Syst. Softw.* 116 (2016), 35–48. https://doi.org/10.1016/j.jss.2015.04.081

[124] Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code search based on context-aware code translation. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. Association for Computing Machinery, New York, NY, 388–400. https://doi.org/10.1145/3510003.3510140

[125] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*. IEEE Computer Society, 476–480. https://doi.org/10.1109/ICSME.2014.77

[126] Watanabe Takuya and Hidehiko Masuhara. 2011. A spontaneous code recommendation tool based on associative search. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE'11)*. Association for Computing Machinery, New York, NY, 17–20. https://doi.org/10.1145/1985429.1985434

[127] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: A new approach to optimization. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 264–276. https://doi.org/10.1145/1480881.1480915

[128] Venkatesh Vinayakarao, Anita Sarma, Rahul Purandare, Shuktika Jain, and Saumya Jain. 2017. ANNE: Improving source code search using entity retrieval approach. In *Proceedings of the 10th ACM International Conference on Web Search and Data Mining (WSDM'17)*, Maarten de Rijke, Milad Shokouhi, Andrew Tomkins, and Min Zhang (Eds.). ACM, 211–220. https://doi.org/10.1145/3018661.3018691

[129] S. Wang, D. Lo, and L. Jiang. 2011. Code search via topic-enriched dependence graph matching. In *Proceedings of the 18th Working Conference on Reverse Engineering*. 119–123.

[130] Shaowei Wang, David Lo, and Lingxiao Jiang. 2014. Active code search: Incorporating user feedback to improve code search relevance. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*. Association for Computing Machinery, New York, NY, 677–682. https://doi.org/10.1145/2642937.2642947

[131] Wenhua Wang, Yuqun Zhang, Zhengran Zeng, and Guandong Xu. 2020. TranSˆ3: A transformer-based framework for unifying code summarization and code search. arXiv:2003.03238. Retrieved from https://arxiv.org/abs/2003.03238.

[132] Xiaoyin Wang, David Lo, Jiefeng Cheng, Lu Zhang, Hong Mei, and Jeffrey Xu Yu. 2010. Matching dependence-related queries in the system dependence graph. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. 457–466.

[133] Doug Wightman, Zi Ye, Joel Brandt, and Roel Vertegaal. 2012. SnipMatch: Using source code context to enhance snippet retrieval and parameterization. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST'12)*. Association for Computing Machinery, New York, NY, 219–228. https://doi.org/10.1145/2380116.2380145

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

[135] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's neural machine translationsystem: Bridging the gap between human and machine translation. arxiv.1609.08144. Retrieved from https://arxiv.org/abs/1609.08144.

[136] Ling Xu, Huanhuan Yang, Chao Liu, Jianhang Shuai, Meng Yan, Yan Lei, and Zhou Xu. 2021. Two-stage attention-based model for code search with textual and structural features. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'21)*. IEEE, 342–353. https://doi.org/10.1109/SANER50967.2021.00039

[137] 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 *Proceedings of the Web Conference 2020*. 2309–2319.

[138] Shufan Zhou, Beijun Shen, and Hao Zhong. 2019. Lancer: Your code tell me what you need. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. IEEE, 1202–1205. https://doi.org/10.1109/ASE.2019.00137

[139] S. Zhou, H. Zhong, and B. Shen. 2018. SLAMPA: Recommending code snippets with statistical language model. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC'18)*. 79–88.

Received 17 February 2022; revised 23 September 2022; accepted 29 September 2022