Toward a Better Alignment Between the Research and Practice of Code Search Engines

Yin Liu
Faculty of Information Technology
Beijing University of Technology
Beijing 100124, China
yinliu@bjut.edu.cn

Shuangyi Li
Software Innovations Lab
Dept. of Computer Science, Virginia Tech
Blacksburg, Virginia 24060, USA
amnos@vt.edu

Eli Tilevich
Software Innovations Lab
Dept. of Computer Science, Virginia Tech
Blacksburg, Virginia 24060, USA
tilevich@cs.vt.edu

***Abstract*—When studying the research literature, one comes to the impression of code search engines as an essential software development tool that developers use regularly to accomplish their daily tasks. Driven by this impression, researchers primarily focus on improving the performance of code search. Nevertheless, as we argue in this paper, this impression is mostly unfounded. As a result, developers and researchers hold dissimilar perspectives on what code search engines are and their most important characteristics, with developers' perspectives and the state of the art often diverging widely.**

This paper aims at reconciling these divergent perspectives by drawing a comprehensive picture of code search engines, as reflected in developers' experiences and perspectives as well as the state of the art. To that end, we first survey more than 100 software developers to ascertain their usages of and preferences for code search engines. We then review the state of the art on this topic by analyzing academic papers, industry releases, and open-source projects. Finally, we juxtapose the results of our two investigations to synthesize a call-for-action for researchers and industry practitioners to better meet the demands of software developers when it comes to searching for code. Our findings can be used to better align the state of the art and practice of code search engines, leading to wider adoption and more effective use of this powerful software development tool.

***Index Terms*—code search engines, user survey, domain analysis**

I. Introduction

Most people find the concept of programming obvious, but the doing impossible.
——————————Alan Perlis (1922–1990)

The realities of the modern software development marketplace require software to be built quickly and reliably. Hence, it is natural to believe that, to meet both of these objectives, developers would be eager to take advantage of any software development tools, especially code search engines, as they assist developers in finding code snippets that can be either reused in a project or easily adapted for the project's needs. Based on this understanding, code search engines have become an important focus area of numerous academic research and industry projects, whose outcomes include various code search engine prototypes [2], [7], [10], [12], [15], [17]–[20], [22], [23], [25], [28] and commercial products [4], [6], [14], [16], all of which differ in their respective search strategies, application scenarios, and execution performance.

However, upon a closer examination, the impression of developers relying on code search engines in their daily tasks turns out to be unfounded. In other words, despite their potential to drastically improve programmer productivity, code search engines still have not become an integral part of the toolset of professional software development. We argue that this situation is a result of researchers being unaware of what developers need when it comes to searching for code.

To address this problem, this paper presents the results of our investigation, whose goal is to draw a comprehensive picture of code search engines from the perspectives of both researchers and developers by answering these questions:

RQ1: How is the term `code search engine` currently understood by developers and researchers? **RQ2:** In which scenarios are code search engines typically used in practice? How well does the state of the art cover these scenarios? **RQ3:** Which characteristics of code search engines do developers find essential, and which features they would like to see introduced? How well has the state of the art code addressed these developers' perspectives for code search engines?

To answer these questions, we (1) surveyed more than 100 software developers who come from dissimilar technical backgrounds, with different lengths of experience, and from several application domains; (2) systematically analyzed a substantial volume of major code search engines, drawing our sources from academic papers, industry releases, and open-source projects. In step (1), we survey developers about their usage of and preferences for code search engines, extracting new insights and unexpected opinions. In step (2), we first extract common characteristics from the investigated products, generalizing their definition and workflow. We then categorize and compare the engines' specific search strategies, typical application scenarios, and execution performance.

In summary, we discovered that developers' perspectives and researchers' foci tend to diverge. They happen to disagree even on what constitutes a code search engines, with developers considering general-purpose search engines (e.g., Google) or code repositories (e.g., GitHub) as code search engines rather than the traditional state of the art. Although the state of the art focuses on the most salient development scenarios, some important cases remain unaddressed. We identified a strong preference that is mostly neglected by the state of the art in supporting code bases in multiple languages and input code sizes of wider variety.

The contribution of this paper is three-fold:

(1) **A survey of software developers' perspectives on using code search engines**: we have identified how software developers define code search engines, how they search for code, which properties of code search engines they find most important, and which features they would most like to see.

(2) **A study that expands the breadth and depth of knowledge of the state of the art of code search engines**: we have studied a large representative set of code search engines not only to extract their common characteristics, but also to summarize their search strategies, usage scenarios, and execution performance.

(3) **A series of findings and insights that bridge the gap between the state of the art of code search and developers' perspectives**: we have analyzed both the knowledge gained from the study and the survey, identifying the mismatches between them and how they can be bridged.

II. Survey of Developers' Perspectives

In this section, we describe the survey we conducted to understand the perspectives of software developers on code search engines.

A. Survey Methodology

1. Challenges & Solutions. As we strove to draw a comprehensive picture of how software developers perceive code search engines, we faced the following challenges:

Challenge-1: how to obtain a representative population of developers to survey? We conducted this survey with the goal of revealing the common perspectives of software developers when it comes to their experiences with code search engines. Hence, we had to ensure that our survey takers come from diverse coding backgrounds and possess dissimilar levels of programming expertise. However, without explicitly selecting participants, an online survey may generate a large volume of useless information.

Solution: we created an invited survey sent to hundreds of developers, including employees of several renowned IT companies (i.e., to cover senior developers) as well as CS students, both undergraduate and graduate (i.e., to cover novices and intermediate). Since these participants come from diverse backgrounds, have different lengths of coding experience, and use dissimilar primary programming languages, we believe the population of developers that participated in our survey is representative to a large extent.

Challenge-2: how to ascertain what comes to the mind of developers when they encounter the term "a code search engine"? As explained in § I, one of our research questions is whether researchers and developers mean the same thing when referring to "a code search engine." We could have asked the surveyed developers to define "a code search engine." However, this strategy would be ineffective for those developers who understand code search engines only vaguely or are even unaware of their existence.

Solution: we are inspired by the Theory of Natural Language Construction posited by Ludwig Wittgenstein. That is, "the meaning of a word is its use in the language [30]." Put differently, a word is not only defined by its textual representation but also by the set of usages of this word in the language. Similarly, "a code search engine" can also be defined by its usage in developers' daily activities. Thus, in our survey questions, we use the term "a code search engine" without defining it and ask participants to describe their typical scenarios of using a code search engine. Then, we examine the described scenarios to understand what developers mean by "a code search engine."

2. Data collection. We invited about 1500 developers to take the survey, and about 114 of them accepted our invitation. The survey takers came from two main groups: IT companies and CS students, while three channels served as venues for disseminating the invitation: mailing lists for companies and students (about 1400 people), company managers requesting their subordinates to participate (the number of subordinates is unknown); and direct contacts with company employees who were personal acquaintances (about 100 developers). Note that, due to our survey being anonymous, the response rates of participants are hard to ascertain. Based on the participant-reported lengths of coding experience, only 6.14% of them had 0-2 years experience, 22.81% more than 10 years, 38.60% 5-10 years, and 32.46% 2-5 years. Based on these responses, we inferred that the majority of the survey takers must have been employed by technology companies, so their coding experience was substantially more extensive than that of a typical CS student.

TABLE I: Survey questions.

Q1 - How long have you been writing code?
Q2 - What is your primary programming language?
Q3 - Do you use a code search engine in your programming pursuits?
Q3-1 - If yes, then which one?
Q3-2 - If no, why not?
Q4 - Which of the following scenarios best describes how you typically use a code search engine
For Q5-Q8, How much do you agree with the following statement:
Q5 - when using a code search engine, how fast it returns its results is the most important criteria
Q6 - Only a highly accurate search engine would be helpful in my software development activities
Q7 - It is important for a search engine to support multiple programming languages
Q8 - It is important for a search engine to be able to work with input of all sizes (from extra short code snippets to large program portions)

3. Survey questions and their purpose. Table I shows eight questions we sent out for our survey takers. The rationale behind this survey design is as follows:

Q1 and Q2 collect a developer's technical background and programming expertise. Specifically, for Q1, we provide four options for the length (i.e., "l") of a developer's programming experience: 0 to 2 (i.e., $0 < l \leq 2$), 2 to 5 (i.e., $2 < l \leq 5$), 5 to 10 (i.e., $5 < l \leq 10$), and more than 10 years (i.e., $l > 10$). For Q2, we provide seven options, six for popular programming languages and one for user-customized input. We select these six languages based on their typical application domains and developers: Python for AI developers, JavaScript for Web developers, C/C++ for system developers, Scratch for CS education developers and programming novices, and Java for the rest of the developers (because of its enormous application domain). Mandatory for all participants, these two

questions make it possible to reveal the information if the surveyed developers are representative.

Q3 and Q4 collect a developer's practices of using code search engines. Specifically, for Q3, we investigate if code search engines are widely used in a developer's programming pursuits (Q3), and which engines are they used (Q3-1). For those developers who claim not to use any code search engine, we aim at understanding why they find search engines unnecessary (Q3-2). For Q3, we provided three predefined reasons and 1 option for customized input. The predefined reasons include: "I am unaware of search engine existence;" "The ones that I tried were not returning useful results;" "I am too busy to learn how to use a search engine." For Q3-1 and Q3-2, we provided customized input only.

For Q4, our target is to unveil scenarios of using code search engines from the end user's perspective. Hence, we ask our survey takers to specify how they typically use a code search engine or select from four pre-defined scenarios: "I have a piece of code that I don't know how to use or am experiencing problems with, so I'd like to search for usage examples;" "I want to implement a certain functionality but do not know how, so I'd like to search for code that matches my needs;" "I use code search engines for both of the two scenarios above;" "I never use code search engines in my programming practices."

Q5 to Q8 collect a developer's preferences for code search engines. By analyzing the research literature, we found that prior works usually focus on improving code search engines' performance in terms of execution time and accuracy. In addition, obtaining an acceptable performance with a small amount of input and supporting different languages are also popular research directions for code search engines. Hence, we designed four survey questions that focus specifically on these four characteristics of code search engines (i.e., execution speed, accuracy, multi-language support, and input size.) Specifically, Q5 surveys a developer's opinion on code search engines' execution speed, Q6 on accuracy, Q7 on supporting multiple programming languages, and Q8 on input size. For each question, the given five agreement levels "strongly disagree," "disagree," "neutral," "agree,"and "strongly agree."

B. Survey Results and Findings

Recall that the survey collected 114 responses, which contained information about developers' technical background, usage of, and preferences for code search engines. Note that some survey takers may have chosen not to answer all eight questions. Hence, for each question, the response may be $<=$ 114. We discuss our survey results in turn next.

1. Technical background: Q1 and Q2 received 114 valid responses, whose analysis revealed that indeed the survey takers came from a wide spectrum of technical backgrounds. For the length of programming experience, 22.81% of our participants have more than 10 years, 38.60% 5-10 years, 32.46% 2-5 years, and 6.14% 0-2 years. For their primary programming language, Python is the most popular language, with 34.21% of our participants, 20.18% for Java, 5.26% for JavaScript, 19.30% for C, 10.53% for C++, 0 for Scratch, and 12% for other languages (i.e., C#, MATLAB, GO, GOlang, Smalltalk, Rust, and Scala.)

2. Usage of code search engines: We obtained 113 responses for Q3. Our results and findings are discussed as follows:

Results of Q3: To our surprise, a considerable amount of participants (44.25%) do NOT use code search engines in their programming practices.

Results of Q3-2: When asked as to why they do not use code search engines, 60% of them pointed out that they were unaware of code search engine existence, 8% complained that the engines fail to return useful results, 2% explained that their business prevented them from learning how to use a code search engine, 4% said they exclusively use Google or Github to search for code, and 20% did not provide a reason.

> **Finding-1 Code search engines are neither widely known nor used:** 44.25% of surveyed developers claimed NOT to use code search engines at all. Moreover, 60% of them did NOT even know that search engines existed.

Results of Q3-1: More interestingly, among the participants who do use code search engines (55.75%), 15.87% of them selected Google[1], 31.75% Github, 20.63% Stack Overflow, 12.70% OpenGrok/Grok, 11.11% for others (e.g., Gerrit, Eclipse, Intellij), and 19.04% left unspecified.

> **Finding-2 When it comes to searching for code, developers use a variety of tools rather than specialized code search engines:** although claiming to "to use code search engines," the survey takers ended up using general-purpose search engines (i.e., Google), code repositories (i.e., GitHub), IDEs (e.g., Eclipse), Q&A websites (i.e., Stack Overflow), rather than specialized code search engines (i.e., OpenGrok/Grok).

Results of Q4: We obtained 90 responses for Q4, and learned that 10% participants use code search engines because they experience problems with a code snippet or do not know how to use it; 23.33% because they want to find code that matches their needs to implement a certain functionality; 36.67% select both of these two reasons, 27.78% say they never use code search engines. Besides that, 2.22% (two participants) specify two other scenarios: one said they use engines during the code review process to understand the implementation of methods that need to be reviewed, and the other said they use engines just for checking where things are.

> **Finding-3 Code search engines can be tailored for these usage scenarios:** (a) having an unfamiliar code snippet whose usage is unclear or problematic, (b) needing to implement an unfamiliar functionality, (c)

[1]There are some overlaps: someone may input both Google and Github.

understanding the implementation for code review, and (d) locating the file that contains a given code snippet. Among them, (a) and (b) are common scenarios (70%, 10%+23.33%+36.67%), (c) and (d) are corner cases (2.22%, 1.11% for each.)

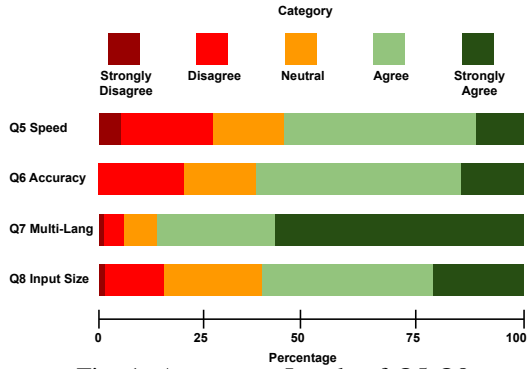

Fig. 1: Agreement Levels of Q5-Q8

3. Preferences for properties of a code search engine: Figure 1 shows the agreement levels of Q5-Q8, for which we obtained 91 responses for Q5 and 90 responses for Q6,7,8, respectively. Most of the participants agree or strongly agree that execution speed (58.24% of our participants), accuracy (65.56%), multiple programming languages support (86.67%), and input of all sizes (62.22%) are crucial criteria for code search engines. In contrast, few participants disagree or strongly disagree with these statements: 25.27% for execution speed, 17.78% for accuracy, 5.55% for multi-language support, and 15.55% for input sizes. Among these four criteria, multi-language support is the most important (86.67% of participants agree or strongly agree, only 5.55% disagree or strongly disagree) and execution speed the least (58.24% agree or strongly agree, 25.27% disagree or strongly disagree.)

Finding-4 The participants found all four characteristics of code search engines important (more than a half of participants agree/strongly agree), **with the support for multiple languages being the most important** (86.67% of participants agree/strongly agree).

C. Threats to Validity

The internal validity is threatened by the different response rate for each question. That is, except for the mandatory questions (i.e., Q1 and Q2), participants could pick and choose which questions they wanted to answer (e.g., answer Q4 but skip Q3). Hence, our findings for these questions may have been derived from the responses whose number is fewer than the total number of participants (i.e., 114). The relative independence of our survey questions might have mitigated this validity threat.

The external validity is threatened by the number of surveyed software developers. We obtained about 100 responses in total. Although we sent out thousands of surveys, the number of responses is not particularly large. Fortunately, these responses cover participants with various technical backgrounds, which can mitigate this validity threat. It is worth mentioning that our survey remains available online, continuously obtaining new responses, which we plan to use in our future research endeavors.

III. Code Search Engines in the Wild

This section discusses the study we conducted to understand the characteristics of code search engines.

A. Study Methodology

1. Challenges & Solutions. To understand the state of the art of code search engines, we had to overcome the following challenges:

Challenge-1: how to determine which keywords can be used to retrieve the relevant work? The meaning of the term "code search engine" is somewhat vague, as the research community thus far as not agreed upon a standard definition. In this study, one of our goals is to explore how the research community uses the term "code search engine." Hence, when searching the literature, we avoided the usage of the keywords that pertain to any particular code search engines. Also, general-purpose search engines (e.g., Google), code repositories (e.g., GitHub), Q& A website (e.g., StackOverflow) can be used to search for source code as well. At this time, our intent was not to decide whether or not exclude these general tools from our study.

Solution: In essence, a code search engine is a tool that searches in computer source code. Therefore, to identify the relevant literature, we used simple search keywords, such as "code search," "code detection," "code matching," and "code search engine." With this specific focus, the results of our study can help counteract some preconceived but overly broad definitions of code search engines.

Challenge-2: how to obtain a manageable yet representative sampling of related work? Parameterized with the aforementioned keywords, literature searches return a massive volume of related work. In an ideal world, we would include all the returned results into our analysis dataset. However, our aim is to study the state of the art both as described in the literature and by interacting with their reproduction packages as end-users to truly understand the operation and inner workings of the studied engines. Hence, digging into all the related work, setting up and running their released solutions would lead to burdensome workloads.

Solution: We follow a strategy that we call "look back and taxonomize." That is, not only do we focus our analysis on the latest research papers, but we also "look back" to understand how these latest examples developed from the historical perspective. To that end, we also analyze some older but classic papers. By reproducing the historical development of code search engines, our goal is to deepen our understanding of how researchers have improved on the state of the art over time. Further, we create a taxonomy of different code search engines by including only the typical and classical ones for

each type. This strategy has helped us significantly reduce the amount of rote work, without sacrificing the relevance of our analysis' findings.

B. Data Collection

As mentioned above, we mainly used the keywords "code search," "code detection," "code matching," and "code search engine", respectively, in Google scholar (for research papers) and GitHub (for industry releases and open-source projects).

Finally, we analyzed and reported on **17 code search engines** (13 reproduction packages from research papers and 4 industry releases), with two different types of in/outputs, three distinct search strategies, and two typical usage scenarios. Moreover, to understand these engines' functionality and implementation, we interacted with them as end-users and analyzed their inner workings.

C. Taxonomy

We classify the considered code search engines based on their usage scenarios, input/output formats, and search strategies, as discussed next.

1. Usage Scenarios. As described in the studied works, the following scenarios exemplify the typical usage scenarios of code search engines:

Type I: Developers have an existing piece of code, but are unsure how to use or are experiencing problems with the code. The ability to consult some usage examples could remediate the situation.

Type II: Developers want to implement a certain functionality, but are unsure how. So they would like to search for suitable code matches. Notice that developers might not have a clear idea of what code to search for.

2. Type of Inputs and Outputs.

(1) Code-to-code Engines take source code as input and return a set of matched code fragments. Typically, a code-to-code search engine would be applied to Type I scenario, with its exact or close matches of the given input code. As a specific example, consider a novice developer needing to learn how to use the numpy function *numpy.vectorize()* in a programming assignment. A code-to-code engine will allow the novice to paste the *"numpy.vectorize()"* string into the search box, with the engine returning a set of occurrences of that function in other projects/repositories.

(2) Natural language-to-code Engines make it possible to discover code based on textual description, thus accommodating those use cases in which the programmer is unaware what code they need for a particular programming task. Typically, a natural language-to-code search engine would be applied to Type II scenario, with its input format in which developers describe the desired functionality in natural language, with the engine returning the code snippets that best match the description. As a specific example, consider an introductory CS student who is assigned to implement a Python project that needs to detect faces. Unfortunately, the student is quite clueless and not even sure what would be a reasonable starting point for implementing this project. A Natural language-to-code engine would make it possible to type in phrases like "how to face detection in python" into the search box, with the engine returning a set of sample face detection code snippets implemented in Python.

3. Search Strategies. In summary, code search engines leverage these major search strategies:

(1) Information Retrieval (IR) Strategies distill the important information from the user input. Before any search can take place, these strategies ensure that the given input provides informative key points that can be effectively searched for in a codebase. These strategies often reformulate or expand the given input with the goal of making the subsequent search process more accurate and effective.

(2) Natural Language Processing-based strategies work with the semantic information of a given text or code snippet. They extract and model information based on its lexical and semantic meanings. Theses strategies work well for searches that involve natural language input.

(3) Deep Learning Strategies make use of deep learning models, such as Recurrent Neural Network (RNN), Convolutional Neural Network (CNN), etc. Deep learning has been applied successfully to extract code features from large codebases. In particular, deep learning strategies excel at automatically capturing relevant code snippets in scenarios that involve vague input or the need to generalize output for unanticipated options. We will expand the discussion of these search strategies in the following sections.

D. History of Code Search Engines

1. Code Search Engine vs. Code Clone. Although it would be hard to pinpoint the exact origin of code search engines, this research topic is inextricably linked to that of detecting code clones. The related work retrieved given the aforementioned keywords contains numerous studies of code clones and source code similarity detection. Indeed, the scope of "code search engines" overlaps with that of "detecting code clone:"

(1) Both code search engines (especially the code-to-code engines) and code clone detectors focus on detecting code similarity. However, unlike code clone techniques, code search engines also cover the scenarios of "natural language to code."

(2) The purpose of a code search engine is searching for code, while that of a code clone detector is searching for code cloned from others. At any rate, both of them search code to accomplish their objectives.

(3) Some approaches employed by code search engines and code clone detectors can be used interchangeably. For example, code clone detection approaches can be applied to a code search engine to search for similar code snippets. Also, some approaches in code search engines are adapted for detecting code clones.

2. Development of Code Search Engines To capture the development of code search engines, we summarized our surveyed code search engines as based on their techniques and publish/release date. We excluded those engines that lacked clear setting up and usage instructions, so we ended up with 17 engines. Although a relatively small sample size, it is representative of the main developments (we discuss it as an external validity threat in § III-H.)

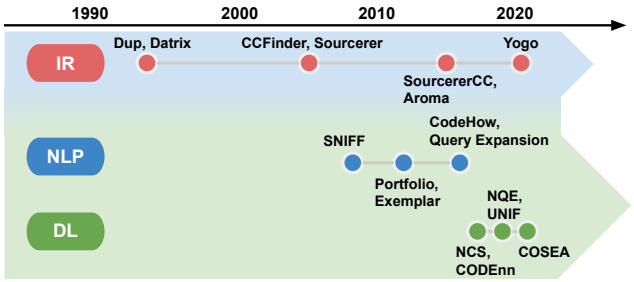

Fig. 2: Historical Development of Code Search Engines

As is shown in Figure 2, not surprisingly, the oldest technique for searching for similar code was string-based IR, introduced before 2000. However, after 2005, NLP and DL-based works started emerging to assist in code search. In the last five years, DL-based techniques tended to become prevalent, but IR-based approaches were still developing. For example, the latest IR-based approach, Yogo, applied a programming analysis approach to search for code. In the future, we believe that NLP and DL will become increasingly widespread, while program analysis will continue to be combined with the IR, NLP, and DL-based approaches.

E. Standard Workflow

We found that a typical code search engine is structured around three major components: (1) user, (2) search data warehousing, and (3) search machinery. Figure 3 shows the general process followed by major code search engines in our study. We will explain each of the components in turn.

(1) User Component: represents engine users and how they interact with the search engine. Users provide search input, which typically comes in the form of either code snippets or natural language. The engine first converts the provided input into search directives. The conversion process involves parsing the input strings and extracting their semantics. Search input can be mapped into complex semantic graphs for use by various machine learning approaches, increasingly common in modern engines.

(2) Search Data Warehousing Component: represents transforming raw codebase(s) into searchable artifacts, described by relevant metadata. In essence, the search process maps the received user input to the parts of the data matching it. To that end, search engines need the ability to access and iterate through massive amounts of data quickly, so the original codebase(s) need to be preprocessed and summarized if a search engine is to provide a responsive user experience.

(3) Search Machinery Component: performs the actual searching operations. It is parameterized by the user and data warehousing components to form the search queries and execute them to return the expected search results. To provide a more meaningful user experience, modern search code engines often also provide additional filtering.

Standard Searching Process:

The operation of a modern code search engine involves the following 6 processes:

1) Convert source code into easily searchable metadata.
2) Transform user input into search directives.

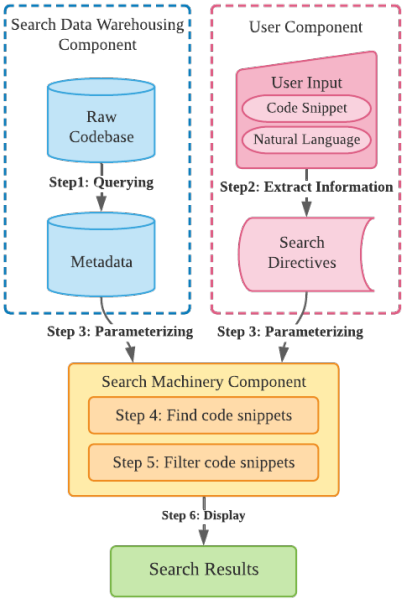

Fig. 3: General Process of Code Search Engines

3) Parameterize the searching machinery with the metadata and search directives, as described above.
4) Find the code snippets that most closely match the input parameters.
5) Filter the found snippets to present more relevant results to the user.
6) Display the final search results to the user.

Typically, process 2 to 5 are performed interactively, while process 1 can be performed as a pre-processing procedure.

F. Summary & Comparison of Code Search Engines

Based on the three different search strategies, we summarize and compare with the studied code search engines as follows:

1. Information Retrieval (IR) Code Search

IR engines work best for ***Type I*** usage scenario and employ one of the following four code retrieval strategies: **string-based**, **token-based**, **tree-based**, and **semantics-based**. We describe these strategies in turn next, while Table II shows a summary of existing works.

String-based strategies treat the source code as a series of string arrays and consider two code fragments similar if all (or part of) their string sequences match ([4] in TableII). However, due to this strategy's computational cost and inflexibility, it has become less popular recently.

Token-based strategies treat the source code as a sequence of tokens, and detect source code similarity via matching duplicated token sequences (or subsequences). Token-based systems are easy to deploy for different programming languages, but they could potentially be more computationally expensive than the text-based methods, as a single line of code typically contains multiple tokens. CCFinder [12] is an earlier token-based systems for detecting code similarities (and code clones); it applies optimization techniques (e.g., aligning token sequence, concatenating tokens) to increase search efficiency and scalability for larger codebases. SourcererCC [25], another

TABLE II: Summary of IR-related works.
●: YES; ◐: Partially YES (have the link to source code, but it is inaccessible); ○: NO

Related work	Techniques	Accuracy	Execution speed	Multi-language support?	Input size	Open source?
Dup [4]	text-based matching, exact match, parameterized match	/	1.1M LOC / 7.2 min	○	min 15 LOC	○
CCFinder [12]	token-based representation, transformation rules	23% more clones compared to line-by-line method	2600k LOC / 250 sec	●	min 50 tokens, min 12 token types	●
SourcererCC [25]	Bag-of-tokens, sub-block overlap filtering, partial index	91% precision, 100% recall	1M LOC / 90 sec; 100M LOC / 1d 12h 54m 5s	○	min 6 LOC	●
Sourcerer [2]	Relational model, text-based ranking, Lucene, structure-based search	67% recall for top 10 results, 74% recall for top 20 results	/	○	2-3 words queries	◐
Datrix [18]	tree-based representation, AST, Intermediate Representation Language (IRL)	/	992256 LOC / 15 min	●	∼50k LOC for case studies tested	○
Aroma [16]	Structural code search, similarity score, parse tree, static scoping, light-weight search	retrieved the original method as top-rank for 99.1% of contiguous and 98.3% of non-contiguous queries	1.3s median response time, 95% queries complete in 4s	●	min 3 tokens, less than 20 LOC	●
Yogo [22]	Program Expression Graph, equality saturation, equivalence graph, DeMorgan's law	All found matches are correct in the selected codebases	/	●	2-3 wordds queries	●

TABLE III: Summary of NLP-related works.
●: YES; ◐: Partially YES (have the link to source code, but it is inaccessible); ○: NO

Related work	Techniques	Accuracy	Execution speed	Multi-language support?	Input size	Open source?
Portfolio [20]	Keyword matching, identifier splitting, PageRank, random surfer, TF-IDF, Spreading Activation Network (SAN)	76% precision	/	○	1-2 sentences, roughly 10-20 words	◐
CodeHow [17]	Extended Boolean Model, text similarity, text normalization, stop word, removal	79.4% precision	/	○	single short sentence, 2-11 words	○
Exemplar [19]	Program analysis, query overlap, S^3 architecture	45% mean precision	/	○	sequence of keywords, exact length unspecified	◐
SNIFF [7]	Free-form query search, bag-of-words	88% correct top ranked result	40% faster than Prospector and Google Code Search	○	sequence of keywords, exact length unspecified	○
Query Expansion [15]	Query expansion, identifier expansion	66 %, 83%, 74% min, max, mean precision; 56% 76%, 67% min, max, mean recall	/	○	sentence-long query, exact length unspecified	○

token-based code clone detector, enables fast searching of large codebases via its bag-of-tokens strategy.

Although token-based approaches usually exhibit lower execution speed due to their high computational cost, combining the bag-of-tokens approach with SourcererCC's sub-block overlap filtering [25] enables token-based strategies to reach high accuracy and execution speed.

Tree-based strategies treat the source code as a tree, especially, the abstract syntax tree (AST) and detect code matches by comparing subtrees. Sourcerer [2] represents the source code with its entities table and the entity relations table to increase query efficiency. Similar to the token-based strategies, one can easily deploy tree-based strategies for multiple languages: Datrix [18] translates a source code's AST into an Intermediate Representation Language (IRL) for multiple languages support. Alternatively, Aroma's simplified parse tree can be used uniformly across various programming languages [16]. Tree-based approaches show outstanding performance for all types of code clones; approximate searches with inexact matches show the best performance.

Semantics-based strategies find semantically similar code rather than lexically similar one. To that end, Komondoor and Horwitz [13] introduced the program dependence graphs (PDGs) [8] and program slicing [29], and YOGO [22] applied program expression graph (PEG). All these program graphs represent a program's semantics to some extent. However, due to their high computational costs, semantics-based strategies are generally considered inapplicable to large codebases.

2. Natural Language Processing-based Engines

Unlike IR strategies, NLP strategies work particularly well for ***Type II*** usage scenario. To match the user input and the searched codebase, NLP-based engines usually model source code's structure and semantics. Table III summarizes representative NLP-based engines.

NLP applied to topic models. Based on statistics, topic models help unlabeled documents' indexation, search, cluster, and structuration [27]. Hence, NLP-based engines can leverage these topic models on searching source code: Exemplar [19] used Vector Space Model (VSM) model, a kind of topic model, to help search, select, and synthesize (S^3) during the code search. Portfolio incorporated a variation of VSM to enhance its preprocessing [20]. CodeHow combined the standard Boolean model and VSM to improve accuracy [17].

NLP with lexical database. Besides the topic models,

TABLE IV: Summary of DL-related works.
●: YES; ◐: Partially YES (have the link to source code, but it is inaccessible); ○: NO

Related work	Techniques	Accuracy	Execution speed	Multi-language support?	Input size	Open source?
NCS [23]	distributional hypothesis, FastText, FAISS, TF-IDF	68.9% accuracy for top 1 result; 94.6% accuracy within top 9 results	/	○	1 short sentence, under 20 words	○
NQE [14]	query expansion, attention, parts-of-speach (POS), beam search	0.284 MRR for query length 1; 0.543 MRR for query length 2	/	○	1-3 words	○
UNIF [6]	bag-of-words based network, attention	60.8% precision for top 1 result	1:11.72 time inference for code; 1:103.83 time inference for query, compared to CODEnn	○	1 short sentence, under 20 words	○
CODEnn [10]	sequence-based network, code embedding, description embedding	46% accuracy for top 1; 76% accuracy for top 5; 86% accuracy for top 10	/	○	under 15 words	●
COSEA [28]	CNN, attentive pooling model	65.7% precision for top 1 result	/	●	average 9 words	○

NLP-based code search engines can use a lexical database to help search for code. Query expansion [15] extended user input terms through a lexical database of English words (i.e., WordNet [21]), which can help match similar code. SNIFF [7] used a free-form query search to generate a set of small, highly relevant, and reusable code snippets to increase the performance and reliability of searching code.

In general, NLP-based engines are similar to IR-based engines, with an extra NLP layer enabling better matching to extract crucial information. Easier to implement, NLP-based engines show high accuracy results.

3. Applying Deep Learning to Search Code

Increasingly common as building blocks of code search engines, deep learning techniques work particularly well for ***Type II*** usage scenarios. Table IV summarizes representative deep learning engines.

In general, training on source code requires a vector as the input. Hence, DL-based code search engines need to convert the source code into a series of vectors, i.e., *code embedding*. Finally, DL-based engines search code by comparing similarities across the converted vectors. In fact, DL-based engines' core process is their code embedding process.

Specifically, NCS [23] combined a code embedding model (FastText [5]) with a similarity search algorithm (FAISS [11]) to improve search accuracy. NQE [14] used Recurrent Neural Network (RNN) to compute the output's probability distribution and adopted a beam search and an attention mechanism to maximize search accuracy. CODEnn [10] embedded both the source code and their corresponding natural language description into a vector space, which can further improve the accuracy but decrease training speed, and lead to semantic inaccuracy when the natural language description is inaccurate [6], [28]. To improve the training speed, UNIF used a bag-of-words-based network that can significantly lower complexity [6], and COSEA introduced the layer-wise attention to the convolutional neural network (CNN) that can enhance the convergence speed [28].

As compared with IR and NLP-based engines, DL-based engines show better performance as they match input by accurately modeling query dependencies. However, the quality and quantity of training data for DL models can greatly impact the search results of DL-based engines [26]. Existing DL-based engines often train their models on a self-cloned code corpus obtained from open-source code repositories like GitHub, but the quality of their obtained training datasets remains hard to evaluate systematically. We observed that some models that claim to have higher performance in theory fail to demonstrate a significant performance bump in reality, while the high accuracy of engines like NCS can be explained by their combining of a good conceptual foundation and high quality training data.

G. Discussion and Findings

Based on the coverage of the code search engines in above Sections, we next discuss our findings.

Finding-5 Need More Performance Metrics: We notice that some aspects of engine performance have not been covered adequately. With a universal focus on **accuracy**, **execution speed** has become de-emphasized in recent years. Execution speed had been the most crucial evaluation metric for IR-based engines, but the evaluations of many recent NLP and DL-based engines never considered it. **Input length**, ranging from small (a few words) to large (multiple lines of code/sentences), can also affect an engine's search performance. Out of all the engines we studied, only NQE [14] tested its performance specifically against small inputs, and no other engines tested against different ranges of input length. **Multi-Language support** is also neglected—less than 30% of our studied engines provided such support.

Finding-6 Studying State-of-the-Art Code Search Engines is Hard: Many of them are either outdated or unavailable. We tested over 30 open-source engines, but only 5 of them would actually execute without errors (Aroma [16], SourcererCC [25], Yogo [22], CCFinder [12], CODEnn [10]), and 2 of them (Aroma and Yogo) would return any search results. The reasons that prevented the engines under test from executing included outdated package environments, evolving libraries, and operating system differences. The engines that executed but would not return results had missing or incomplete codebases to search.

TABLE V: Comparison between developer's perspectives and existing code search engines

	Usage Scenarios	Importance in daily work	Preferences for properties
Developers' perspectives	(a) having an code snippet with unclear/problematic usage; (b) needing to implement an unfamiliar functionality; (c) understanding the implementation for code review ; (d) locating a given code snippet (c and d are corner cases.)	neither widely known nor used; use other tools	1st: Multi-lang support; 2nd: Accuracy; 3rd: Input size; 4th: Exec. speed
Existing code search engines	Have an existing piece of code: (a) being unsure how to use (b) experiencing problems with the code. Have no existing code: (c)trying to implement a certain functionality, but are unsure how.	motivate the research: programmers heavily rely on it.	Focus on: Accuracy; De-emphasized: Exec. speed; Neglect: Multi-lang support, Input size

H. Threats to Validity

The internal validity is threatened by our experimental environment. In Tables II, III, and IV, we collected the accuracy, execution speed, and input size of different code search engines. However, these measurement results come from dissimilar experimental environments. That is, our surveyed code engines were evaluated on different code bases, test cases, experimental machines, etc. This threat could have been mitigated if our surveyed code engines run without errors. However, as mentioned above, most of them proved hard to deploy and operate, the root cause of this threat.

The external validity is threatened by the number of surveyed code search engines. In this paper, we summarized and compared 17 code search engines, a representative sample but not sufficiently large to draw definitive conclusions. To ensure further progress, we plan to open-source our experimental data collection, thus allowing other researchers to expand on it.

IV. Comparing Apples and Oranges

Based on the results and findings presented in § II and § III, developers' perspectives and the state of the art exhibit both commonalities and distinctions, which we discuss in turn next.

A. Commonalities

As shown in Column "Usage Scenarios" (Table V), developers expect the same usage scenarios as the existing code search engines provide. Hence, we extract our definition of code search engines from these scenarios.

1. Clarifications: Based on the usage scenarios, we differentiate code search engines from other related tools as follows:

(a) *We do not consider general-purpose search engines as code search engines.* Based on their usage scenarios, the goal of a code search engine is searching code, rather than all possible resources on the web, as is the case of general-purpose search engines (e.g., Google).

(b) *We do not consider code repositories as code search engines.* Code repositories provide source control and management services; they might provide simple search facilities, but it is not their raison d'être. In contrast, a code search engine is specifically designed to search any collection of codebases.

(c) *We do not consider Question & Answer forums as code search engines.* A developers can post a question on a Q & A forum (e.g., stack overflow), with some other developers answering that question, with the answer preserved for future referencing. In contrast, a code search engine interactively returns a set of code snippets given a search input, without a human actor behind the process.

2. Definition: Consider a code repository `R` and user search input `I`; `R` contains a finite set of codebases (each includes the source code, metadata, config files, etc.); `I` can be either code snippets or natural language tokens. `E`, a code search engine, processes and transforms `R` and `I` to make them searchable and matchable, respectively, and then outputs the results as a set of code snippets `S`. Thus, `E` matches `I` to $S \subset R$.

B. Distinctions

As shown in Table V, developers and researchers (i.e., designers of existing code search engines) tend hold different opinions on how important code search engines are in the performance of daily development tasks and which properties developers prioritize, as we discuss in turn next.

1. Code search engines have not yet become a standard software development tool: Despite the claims made in the research literature about being motivated by the significance of code search engines role in the development process and day-to-day programming activities, a considerable amount of surveyed developers never used or were even unaware of code search engines (Findings 1,2). Even worse, among the survey takers who claim to use "code search engines", the majority end up using general-purpose engines, code repositories, and Q&A websites rather than "real" code search engines.

The possible reasons of this phenomenon could be: (a) existing tools (e.g., general-purpose search engines, code repositories, and Q&A websites) perform largely the same role as code search engines. So developers are not compelled to spend time on learning how to use a new tool. (b) existing code search engines are unavailable (Finding-6) or return useless results (Results of Q3-2 in § II-B).

2. Researchers often leave unaddressed what developers find important in the functioning of code search engines: Although accuracy is ranked highly in both developers' perspectives and the state of the art, existing engines tend to under-emphasize the support for multiple languages (ranked first by developers) and input size (ranked higher than execution speed by developers).

This phenomenon implies that researchers may be unaware of what their end-users (i.e., developers) expect from code search engines. Our findings suggest that researchers may benefit from focusing more on multi-language support and input size as a way to better meet developer expectations.

V. Related Work

As depicted in Table VI, several prior research efforts have also studied state-of-the-art code search engines and what developers expect when searching for code. Garcia et al.

summarized a series of usage requirements from the research literature that describes existing code search engines [9]. Sadowski et al. surveyed developers to understand how they search for code and which search patterns they deploy [24]. By analyzing frequency/difficulty when it comes to searching for code online, Xia et al. studied developers' behaviors of web code search [31]. By tracking an online code search engine's logs, Bajracharya et al. mined the search topics used by developers [1], [3]. Despite uncovering numerous interesting insights, these prior works have not specifically focused on systematically studying existing code search engines in terms of their common characteristics and unique functionalities. Furthermore, to the best of our knowledge, no prior user studies have set the goal of identifying the perspectives of software developers with respect to their current usage patterns of and future preferences for code search engines (Table VI).

TABLE VI: Summary of related surveys.
●: Fully Covered; ◐:Partially Covered; ○: NOT Covered

Related Survey	Classify?	User Perspective?	Requirements?	Comparison?
Garcia et al. [9]	○	○	●	○
Sadowski et al. [24]	○	◐	○	○
Xia et al. [31]	○	◐	○	○
Bajracharya et al. [1], [3]	○	◐	○	○
This Paper	●	●	●	●

VI. Conclusion

In this paper, we conducted (1) a study of state-of-the-art code search engines and (2) a developer survey of more than a 100 developers. We found that a considerable percentage of developers never use code search engines and are even unaware of their existence. We hope that the results of this research will help developers to benefit from using search engines in their professional practices, and researchers to uncover future directions that would have the most potential for practical impact.

Acknowledgements

The authors thank the anonymous reviewers, whose insightful comments helped improve this paper. This research is supported by NSF Grants #1717065 and #2232565 and Cisco Research Grant CG# 1367161.

References

[1] S. Bajracharya and C. Lopes, "Mining search topics from a code search engine usage log," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 111–120.

[2] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "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*, 2006, pp. 681–682.

[3] S. K. Bajracharya and C. V. Lopes, "Analyzing and mining a code search engine usage log," *Empirical Software Engineering*, vol. 17, no. 4, 2012.

[4] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of 2nd Working Conference on Reverse Engineering*. IEEE, 1995, pp. 86–95.

[5] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[6] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 964–974.

[7] S. Chatterjee, S. Juvekar, and K. Sen, "Sniff: A search engine for java using free-form queries," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2009, pp. 385–400.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

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

[10] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[11] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, 2019.

[12] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE transactions on software engineering*, vol. 28, no. 7, 2002.

[13] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *International static analysis symposium*. Springer, 2001, pp. 40–56.

[14] J. Liu, S. Kim, V. Murali, S. Chaudhuri, and S. Chandra, "Neural query expansion for code search," in *Proceedings of the 3rd acm sigplan international workshop on machine learning and programming languages*, 2019, pp. 29–37.

[15] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan, "Query expansion via wordnet for effective code search," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 545–549.

[16] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

[17] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 260–270.

[18] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics." in *icsm*, vol. 96, 1996, p. 244.

[19] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1069–1087, 2011.

[20] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[21] G. A. Miller, *WordNet:An electronic lexical database*. MIT press, 1998.

[22] V. Premtoon, J. Koppel, and A. Solar-Lezama, "Semantic code search via equational reasoning," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 1066–1082.

[23] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: a neural code search," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018, pp. 31–41.

[24] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: a case study," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 191–201.

[25] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1157–1168.

[26] Z. Sun, L. Li, Y. Liu, and X. Du, "On the importance of building high-quality training datasets for neural code search," *arXiv preprint arXiv:2202.06649*, 2022.

[27] S. W. Thomas, "Mining software repositories using topic models," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 1138–1139.

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

[29] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[30] L. Wittgenstein, *Philosophical investigations*. John Wiley&Sons, 2009.

[31] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing, "What do developers search for on the web?" *Empirical Software Engineering*, vol. 22, no. 6, pp. 3149–3185, 2017.