An Investigation into Code Search Engines: The State of the Art Versus Developer Expectations

Shuangyi Li

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Eli Tilevich, Chair

Muhammad A. Gulzar

Francisco Servant Cortes

June 3, 2022

Blacksburg, Virginia

Keywords: Code search engines, User survey, Domain analysis

Copyright 2022, Shuangyi Li

An Investigation into Code Search Engines: The State of the Art Versus Developer Expectations

Shuangyi Li

(ABSTRACT)

An essential software development tool, code search engines are expected to provide superior accuracy, usability, and performance. However, prior research has neither (1) summarized, categorized, and compared representative code search engines, nor (2) analyzed the actual expectations that developers have for code search engines. This missing knowledge can empower developers to fully benefit from search engines, academic researchers to uncover promising research directions, and industry practitioners to properly marshal their efforts. This thesis fills the aforementioned gaps by drawing a comprehensive picture of code search engines, including their definition, standard processes, existing solutions, common alternatives, and developers' perspectives. We first study the state of the art in code search engines by analyzing academic papers, industry releases, and open-source projects. We then survey more than a 100 software developers to ascertain their usage of and preferences for code search engines. Finally, we juxtapose the results of our study and survey to synthesize a call-for-action for researchers and industry practitioners to better meet the demands software developers make on code search engines. We present the first comprehensive overview of state-of-the-art code search engines by categorizing and comparing them based on their respective search strategies, applicability, and performance. Our user survey revealed a surprising lack of awareness among many developers w.r.t. code search engines, with a high preference for using general-purpose search engines (e.g., Google) or code repositories (e.g., GitHub) to search for code. Our results also clearly identify typical usage scenarios and

sought-after properties of code search engines. Our findings can guide software developers in selecting code search engines most suitable for their programming pursuits, suggest new research directions for researchers, and help programming tool builders in creating effective code search engine solutions.

An Investigation into Code Search Engines: The State of the Art Versus Developer Expectations

Shuangyi Li

(GENERAL AUDIENCE ABSTRACT)

When developing software, programmers rely on source code search engines to find code snippets related to the programming task at hand. Given their importance for software development, source code engines have become the focus of numerous research and industry projects. However, researchers and developers remain largely unaware of each other's efforts and expectations. As a consequence, developers find themselves struggling to determine which engine would best fit their needs, while researchers remain unaware what developers expect from search engines. This thesis address this problem via a three-pronged approach: (1) it provides a systematic review of the research literature and major engines; (2) it analyzes the results of surveying software developers about their experiences with and expectations for code search engines; (3) it presents actionable insights that can guide future research and industry efforts in code search engines to better meet the needs of software developers.

Dedication

To my family and cats. Thanks for always being there for me.

Acknowledgments

I would like to express my deepest appreciation to *my advisor, Dr. Eli Tilevich,* for his mentoring, from which I have benefited tremendously. I appreciate how he is always available whenever I need help. I appreciate how he never hesitates to contribute his valuable time and expertise to support me by providing feedback on my research ideas, giving me constructive suggestions, and helping me polish my writing to make sure it achieves the highest possible quality. I appreciate how he always guides me through multiple ideas and directions, striving to identify a research path that is the most suitable for me. I appreciate how respectful he is, always listening and understanding, gently guiding me to my best.

I would also like to extend my deepest gratitude to *my research collaborator, Dr. Yin Liu*. I appreciate that he always gives me time and support, helping me brainstorm ideas, and sharing all his expertise, experiences, and resources to help me achieve my best.

I must also extend my sincere thanks to *my committee members*, Dr. Francisco Servant and Dr. Muhammad Ali Gulzar. I appreciate their timely feedback and valuable suggestions that guide me to improve this thesis's overall quality and future research ideas.

I'm also grateful to *all my CS@VT friends*. Thanks for their help in my research and daily life.

I'd also like to give some special thanks to *my family*. Thanks for their continuous encouragement and support.

Contents

List of Figures x

List of Tables xi

1 Introduction 1

2 Motivation, Research Questions, and Contributions 3

2.1 Research Questions 4

2.2 Contributions 5

2.3 Roadmap 6

3 Technical Background 7

3.1 Definitions 7

3.2 Assumptions 8

4 Code Search Engines in the Wild 9

4.1 Formal Definition 9

4.2 Standard Workflow 10

4.3 Taxonomy 12

4.3.1 Type of Inputs and Outputs 12

4.3.2 Search Strategies . . . 14
4.3.3 Usage Scenarios . . . 14
4.4 Information Retrieval (IR) Code Search . . . 15
4.5 Natural Language Processing (NLP)-based Engines . . . 18
4.6 Applying Deep Learning to Search Code . . . 20
4.6.1 Deep Learning Models . . . 21
4.6.2 Specific approaches . . . 21
4.7 Discussion and Findings . . . 23

5 Developers' Perspectives on Code Search Engines 26
5.1 Survey Methodologies . . . 27
5.2 Survey Results and Findings . . . 29
5.3 Discussion and Insights . . . 32

6 Future Work 34

7 Summary and Conclusions 36

Bibliography 37

Appendices 44

Appendix A Survey Material 45

A.1 Survey Info 45

A.2 User Survey 48

A.3 IRB Approval 53

List of Figures

4.1 General Process of Code Search Engines 11

5.1 Agreement Levels of Q5-Q8 31

List of Tables

2.1 Summary of related surveys. 4

4.1 Summary of IR-related works. 16

4.2 Summary of NLP-related works. 18

4.3 Summary of DL-related works. 20

5.1 Survey questions. 26

List of Abbreviations

DL Deep Learning

IR Information Retrieval

LDA Latent Dirichlet Allocation

NLP Natural Language Processing

OMT Object Modeling Technique

PDG Program Dependence Graph

POS Parts-of-speech

SAN Spreading Activation Network

TF/IDF Term Frequency/Inverse Document Frequency

VSM Vector Space Model

Chapter 1

Introduction

Most people find the concept of programming obvious, but the doing impossible.

Alan Perlis (1922–1990)

Consider a developer assigned to work on an unfamiliar project feature, such as encrypting some data. Alas, the developer is unsure which encryption algorithms would be appropriate to use and how to implement them correctly. Both issues are critically important: encryption algorithms possess vastly dissimilar properties that satisfy different requirements, and an improper implementation of a suitable encryption algorithm can cause serious security breaches [24]. It would be quite helpful to be able to examine how other real-world projects implement encryption, learn what the best practices are, and then to follow these practices when implementing this feature. At that point, as if reading the developer's mind, the browser flashes an advertisement across the screen: "Simple, comprehensive code search – helping you find real world examples of functions, API's and libraries in 243 languages across 10+ public code sources[10]."

This ad refers to what is commonly referred to as *a code search engine*. As it turns out, the question of what exactly a code search engine is, and whether it is similar to Google or Bing, is far from trivial. Based on a comprehensive sample of software developers, we were surprised to find out that about 44% of them have never used a code search engine. Despite a massive intellectual investment by the research community in creating ever more

sophisticated code search engines, a sizable percentage of software developer has remained unaware of the existence of this increasingly important tool in the development of modern software.

To learn more about this concept, one may search for the keyword "code search engine" online. However, among relevant advertisements of code search engines, one would have a hard time trying to separate hype from reality. Hence, finding answers to the following questions could be helpful: (1) What do existing code search engines do? How do they work? (2) Which criteria should a developer use to choose a code search engine, and which one would be the best fit for a specific software development scenario? (3) How do your peers use code search engines? What have been their experiences and preferences?

As it turns out, finding comprehensive answers to these questions is hard, with neither research papers nor technical articles providing definitive guidance. Therefore, the overriding objective of this thesis is to describe, compare, and discuss *Code Search Engines*.

Chapter 2

Motivation, Research Questions, and Contributions

The realities of the modern software development marketplace require that software be built both quickly and reliably. Striving to meet both of these objectives, developers are eager to take advantage of any tools that can help improve software development productivity and quality. Among these tools, code search engines feature prominently, as they assist developers in finding code snippets that can be either reused in a project as they are or easily adapted for the project's needs. Post-pandemic, working from home, at least on some days, has become the new normal for a large portion of the software development workforce. Away from their colleagues, developers more than ever find themselves in need of tools that allow to quickly draw from the collective wisdom contained in existing codebases. Armed with a usable, accurate, and flexible code search engine, any developer would be better equipped to meet the aforementioned objectives.

Not surprisingly, code search engines have been an important area of focus of numerous academic research and industry projects. Various code search engine research prototypes and commercial products differ in their respective search strategies, application scenarios, and execution performance. In light of this variability, developers are eager to determine how to select the most appropriate code search engine for a given scenario. At the same time, researchers are eager to receive actionable feedback from developers, as a way to determine

how to best apply their current efforts and identify future research directions.

As depicted in Table 2.1, several prior research efforts have also studied the state of the art of code search engines and what developers expect when it comes to searching for source code. Garcia et al. summarized a series of usage requirements from the research literature that describes existing code search engines [17]. Sadowski et al. surveyed developers to understand how they search for code and which search patterns they deploy [42]. Xia et al. investigated the frequency/difficulty of web code search tasks performed by developers [51]. Bajracharya et al. mined the search topics used by developers from a year-long usage log of a commercial code search engine [3, 5]. 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 2.1).

Table 2.1: Summary of related surveys.

Related Survey	Classify?	User Perspective?	Requirements?	Comparison?
Garcia et al.[17]	○	○	●	○
Sadowski et al.[42]	○	◐	○	○
Xia et al. [51]	○	◐	○	○
Bajracharya et al.[3, 5]	○	◐	○	○
This Thesis	●	●	●	●

●: Fully Covered; ◐:Partially Covered; ○: NOT Covered

2.1 Research Questions

To fill these knowledge gaps, we present our findings with the goal of drawing a comprehensive picture of code search engines by answering these questions:

- **RQ1:** What are the formalized definition and standard workflow of code search engines?
- **RQ2:** What are the unique search strategies, application scenarios, and execution performance of major existing code search engines?
- **RQ3:** How do developers use code search engines, which of their properties they find important, and which new features they'd like to have introduced?

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

2.2 Contributions

The contribution of this thesis is three-fold:

1. **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.

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

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

2.3 Roadmap

The rest of the thesis is structured as follows. Chapter 3 introduces the technical background of this research. Chapter 4 categorizes and explains existing code search engines. Chapter 5 describes the developer's perspective on code search engines. Chapter 6 presents future work direction, while Chapter 7 states our conclusion.

Chapter 3

Technical Background

In this chapter, we first define the main technical concepts referenced in this thesis and then outline the assumptions we have made when carrying out this research.

3.1 Definitions

Information Retrieval (IR): IR describes the process of extracting relevant information from an information resource system. In code search engines, IR is often used for matching information in a codebase required to extract relevant code snippets.

Natural Language Processing (NLP): NLP approaches are widely used to pre-process the unstructured information of text written in a natural language in order to extract the lexical and semantic information. In code search engines, NLP is commonly applied to better understand user needs, as source code can be treated as unstructured text written in a natural language.

Deep Learning (DL): DL refers to those machine learning techniques that are constructed with three or more layers of neural networks. DL imitates how humans gain certain types of knowledge. In code search engines, DL has become popular in much of recent research due to DL's ability to structure complex information through embedding. The searched source code usually contains a series of highly complex information. Source code and natural lan-

guage input information are embedded into a vector, and vector distances are calculated to approximate the semantic correlation between the source code and the input information. *Unsupervised approaches* in DL would only rely on a corpus of code, while *supervised approaches* embed source code and their natural language description jointly into a vector space.

3.2 Assumptions

Assumption 1: *We do not consider general-purpose search engines as code search engines.* A code search engine's repository contains codebases only, rather than all possible searchable resources on the web, as is the case of general-purpose search engines.

Assumption 2: *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, analogously to a general-purpose search engine searching any collection of resources.

Assumption 3: *We do not consider Question & Answer forums as code search engines.* A developers can post a question on a Question & Answer 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.

Chapter 4

Code Search Engines in the Wild

In this chapter, we first formally define the concept of code search engines and then present our findings derived from analyzing representative engines. The findings reported herein are based on **17 code search engines** (13 reproduction packages from research papers and 4 industry releases). To understand these engines' functionality and implementation, we have interacted with them as end-users and analyzed their inner workings, respectively.

4.1 Formal Definition

Because the overriding goal of this study is to systematically explore the subject of code search engines, we next propose the following formal definition of this concept.

Consider a code repository `R` and user search input `I`; `R` contains a finite set of codebases (each includes all the contained source code, metadata, configuration files, build scripts, 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 $\mathtt{S} \subset \mathtt{R}$.

4.2 Standard Workflow

From the end-user's perspective, a code search engine can be seen as a search engine that takes code and/or natural language as input, and returns code snippets as its search results. Architecturally, a typical code search engine is structured around three major components: (1) user, (2) search data, and (3) search machinery. Figure 4.1 shows the general process followed by major code searching engines. In the discussion below, we will explain each of the depicted components in turn.

(1) User Component: The 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: The 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: The 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.

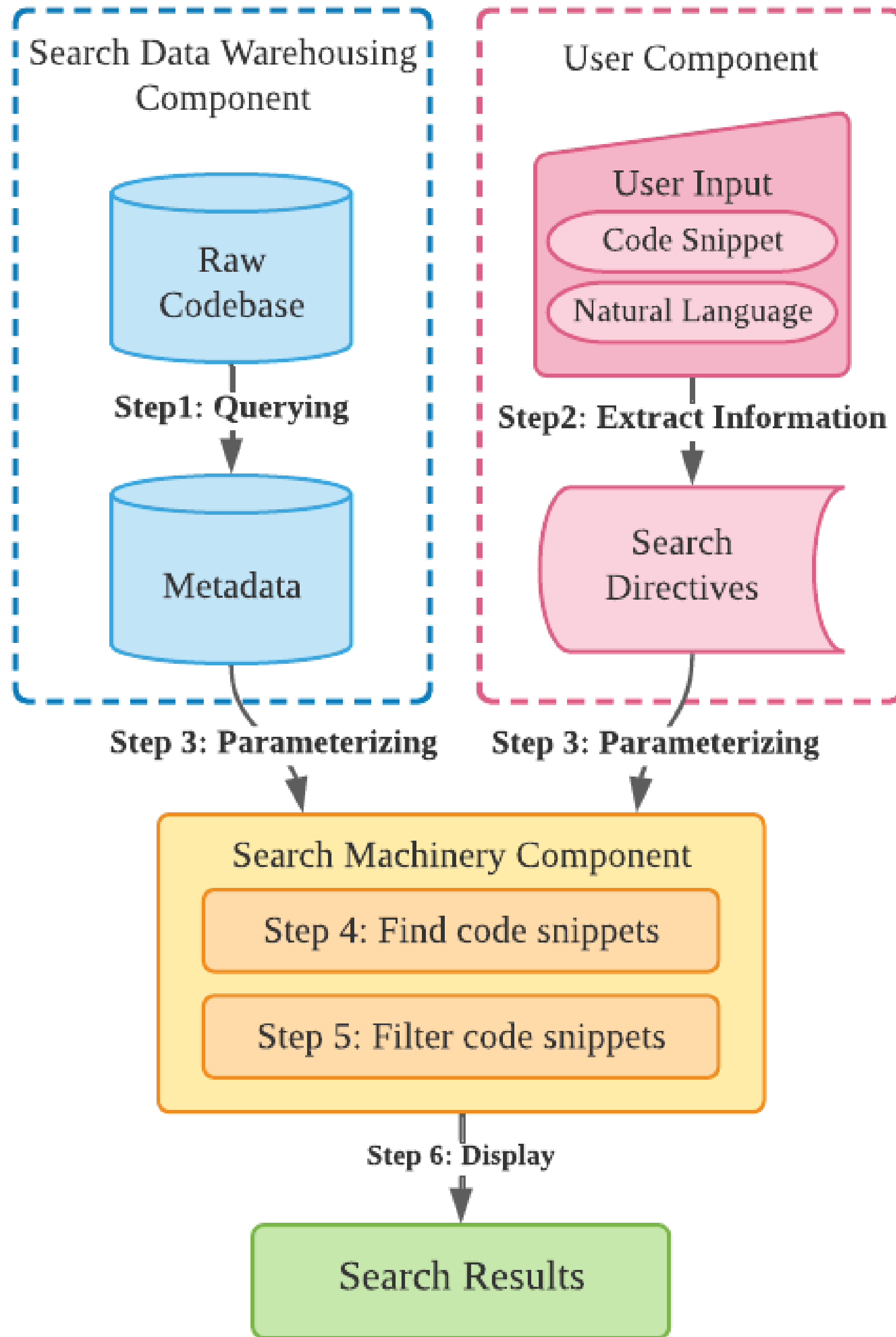

Figure 4.1: General Process of Code Search Engines

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.
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.

4.3 Taxonomy

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

4.3.1 Type of Inputs and Outputs

From the end user's perspective, code search engines fall into two general categories: (1) code-to-code and (2) natural language-to-code.

(1) Code-to-code Engines take source code as input and return a set of matched code fragments. A representative usage scenario for this type of engine would be a programmer having a code snippet and using a search engine to discover how to use that snippet. Another scenario would involve using an engine to discover how a code snippet that implements a given functionality in one language can be implemented in other languages. Yet another scenario could involve discovering functionalities similar to that expressed by an input code snippet. As specific example, consider a CS student needing to learn how to use the numpy function *numpy.vectorize()* in a programming assignment. A code-to-code engine will allow the student 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. These engines make it possible for the programmer to express their search query as a statement in a natural language. In response, such engines would return those source code snippets that match the input natural language statements. 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.

4.3.2 Search Strategies

Since the requirements that search engines needs to fulfill tend to vary widely, the specific search strategies followed by different engines also differ in a variety of ways. However, the three major search strategies used by state-of-the-art engines are deep learning and query processing that we discuss next.

(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 networks, 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.

4.3.3 Usage Scenarios

To use a code search engine, developers typically find themselves in one of these scenarios:

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. In such scenarios, a code-to-code search engine would be most suitable, with its exact or close matches of the given input code.

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. In such scenarios, a natural language-to-code search engine would be most suitable, 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.

4.4 Information Retrieval (IR) Code Search

Information retrieval (IR) has become an intrinsic part of various searching processes. Hence, it is not surprising that many code search engines integrate this technology. 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 4.1 shows a summary of existing works.

String-based strategies represent a source code file as a contiguous sequence of strings, usually divided by lines. Two code fragments are considered similar if all or part of their string sequences match. Baker's Dup method [6] is one of the representative examples that locates exact or near duplicated code via a parameterized matching algorithm. String-based

[1] *Precision* and *recall* are the standard information retrieval metrics for assessing code search engines. *Precision* is the ratio of the number of relevant records retrieved to the total number of irrelevant and relevant records retrieved. *Recall* is the ratio of the number of relevant records retrieved to the total number of relevant records available. Generally, a high precision or recall score indicates better accuracy. Some engines only have either one of these scores as measuring metrics due to their specific application foci.

Table 4.1: Summary of IR-related works.

Related work	Techniques	Accuracy	Execution speed	Multi-language support	Open source	Input size
Dup [6]	text-based matching, exact match, parameterized match	/	1.1M LOC / 7.2 min	○	○	min 15 LOC
CCFinder [22]	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 [43]	Bag-of-tokens, sub-block overlap filtering, partial index	91% precision, 100% recall [1]	1M LOC / 90 sec; 100M LOC / 1d 12h 54m 5s	○	●	min 6 LOC
Sourcerer [4]	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 [30]	tree-based representation, AST, Intermediate Representation Language (IRL)	/	992256 LOC / 15 min	●	○	~50k LOC for case studies tested
Aroma [28]	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 [38]	Program Expression Graph, equality saturation, equivalence graph, DeMorgan's law	All found matches are correct in the selected codebases	/	●	●	2-3 wordds queries

●: Fully Covered; ◐:Partially Covered / open source link or repository inaccessible; ○: NOT Covered;

code search was very popular in the early stages of creating technologies that detect code clones or search for code similarities. However, due to this strategy's computational cost and inflexibility, it has become less popular recently.

Token-based strategies parse a program into a sequence of tokens. The resulting sequence is then scanned for duplicate subsequences of tokens with the goal of identifying the potential presence of code clones. As compared to string-based methods, a token-based strategy is often more resilient against code changes, such as formatting and spacing. Token-based systems are also easy to deploy for different programming languages. A down side of token-based methods is that they could potentially be more computationally expensive than the text-based methods, as a single line of code typically contains multiple tokens. However, multiple optimization techniques have been proposed to improve the computational efficiency of token-based methods. CCFinder [22] is one of the earlier token-based systems for detecting code clones; it applies multiple optimizations, including aligning token sequence, concatenating tokens, etc., as a way to to increase search efficiency, making it scalable for larger codebases. SourcerCC [43] is also a token-based code clone detector. Its bag-of-tokens strategy allows for simple and fast searching of large codebases.

Tree-based strategies represent a program's source code as a parse tree or an abstract syntax tree (AST). These strategies then compare subtrees within the resulting parse tree or AST to identify exact or close matches of subtrees [8, 47]. Tree based method are known for their ability to detect all TYPE-1, TYPE-2, and TYPE-3 code clones [9]. Inspired by the AST strategy, Sourcerer [4] introduces a relational representation method that consists of a program entities table and the entity relations table to allow for efficient querying. Similar to the token-based strategies, tree-based strategies are also known to be easy to deploy for multiple languages. The Datrix assessment framework [30] translates an AST into an Intermediate Representation Language (IRL) to support multiple languages. Although as compared to more recent works, Datrix's efficiency is not particularly striking, but its IRL abstraction that uses the Object Modeling Technique (OMT) object model [40] leads the way for supporting multi-lingual searching scenarios. Alternatively, Aroma's simplified parse tree can be used uniformly across various programming languages [28].

Semantics-based strategies find semantically similar code rather than code that is lexically similar. Komondoor and Horwitz [23] introduced the program dependence graphs (PDGs) [16] and program slicing [50] as an approach to finding isomorphic PDG subgraphs that represent clones. YOGO [38] applied program expression graph (PEG) that represents a program's semantics, including mutation and loops, as pure data-flow. Despite their high precision, semantics-based strategies are generally considered inapplicable to large codebases, due to their high computational costs. YOGO, as an example, was experiencing timeout issues even for small codebases due to it usage of a high complexity routine for generating an expanded equivalence graph (E-PEG).

Table 4.2: Summary of NLP-related works.

Related work	Techniques	Accuracy	Execution speed	Multi-language support	Open source	Input size
Portfolio [32]	Keyword matching, stemming, identifier splitting, PageRank, random surfer, TF-IDF, Spreading Activation Network (SAN)	76% precision	/	○	◐	1-2 sentences, roughly 10-20 words
CodeHow [29]	Extended Boolean Model, text similarity, stemming, text normalization, stop word removal	79.4% precision	/	○	○	single short sentence, 2-11 words
Exemplar [31]	Program analysis, query overlap, S^3 architecture	45% mean precision	/	○	◐	sequence of keywords, exact length unspecified
SNIFF [14]	Free-form query search, stemming, bag-of-words	88% correct top ranked result	40% faster than Prospector and Google Code Search	○	○	sequence of keywords, exact length unspecified
Query Expansion [27]	Query expansion, identifier expansion, stemming	66 %, 83%, 74% min, max, mean precision; 56% 76%, 67% min, max, mean recall	/	○	○	sentence-long query, exact length unspecified

●: Fully Covered; ◐:Partially Covered / open source link or repository inaccessible; ○: NOT Covered;

4.5 Natural Language Processing (NLP)-based Engines

A common strategy for processing codebases is Natural Language Processing (NLP). Unlike IR strategies, NLP strategies focus on not only extracting text information at lexical and syntactic levels, but also on analyzing the extracted information at the semantic and even discourse levels [15]. These engines work particularly well for ***Type II*** usage scenario. To match the user input and the searched codebase in a meaningful way, NLP-based engines have to be able to accurately model a programming language's structure and semantics. Table 4.2 summarizes representative NLP-based engines.

NLP techniques are usually applied to extract information from the searched source code. The most commonly discussed and widely used approach during this phase is *stemming*. Stemming is a computational procedure that is applied to all words that share the same root (if prefixes are unchanged, then the same stem) with the goal of reducing them a common form. Stemming typically entails stripping the derivational and inflectional suffixes of words [26]. Popular engines, including Portfolio [32] and CodeHow [29], stem all the source code in their code repositories, while SNIFF [14] only stem all the keywords in the code comments. Other NLP techniques, such as identifier splitting [32], text normalization,

stop word removal [29], query overlap [31], camel case, and underscores splitting [44] have also been found helpful across different engines. As NLP techniques preprocess information, they store the result—the preprocessed data—available for further analysis to be performed during the information matching phase.

NLP applied to topic models. An important part of NLP-based engines is making use of NLP-preprocessed data. Some engines apply well known topic models to preprocessed data. Topic models are statistical models that provide a means to automatically index, search, cluster, and structure unstructured and unlabeled documents, such as code [46]. Latent Dirichlet Allocation (LDA) is a three-level hierarchical Bayesian model that models a collection of text items over an underlying set of topic probabilities to explicitly represent a document [11]. LDA is a popular probabilistic topic model for discovering a set of themes that describe the entire code corpus. However, some studies point out that applying NLP techniques might not make significant differences on the results [44]. Another popular topic model is Vector Space Model (VSM) that processes the preprocessed data. A bag-of-words retrieval technique, VSM, models a document as a vector of all words it contains. The vectors are given a weight based on the Term Frequency/Inverse Document Frequency (TF/IDF). Exemplar [31] uses the VSM model as the foundation of their Searching, Selecting, and Synthesizing (S^3) architecture. Portfolio effectively incorporated Lucene Java Framework — a variation of VSM — with PageRank and Spreading Activation Network (SAN) for analyzing the preprocessed data[32]. An Extended Boolean model combines the standard Boolean model, returned by most query term matches, and the VSM, returned by higher term frequency matches, for improved accuracy [27].

NLP with lexical database. Although topic models are widely used for analyzing NLP-preprocessed data, additional code search options are being explored. Introduced by Lu et. al., query expansion extends user input terms by finding synonyms [27]. The engine

Table 4.3: Summary of DL-related works.

Related work	Techniques	Accuracy	Execution speed	Multi-language support	Open source	Input size
NCS[41]	Unsupervised learning, 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[25]	Unsupervised learning, query expansion, parts-of-speach (POS), beam search, attention	0.284 MRR for query length 1; 0.543 MRR for query length 2	/	○	○	1-3 words
UNIF[13]	Supervised learning, 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[18]	Supervised learning, 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[48]	Supervised learning, CNN, attentive pooling model	65.7% precision for top 1 result	/	●	○	average 9 words

●: Fully Covered; ◐:Partially Covered / open source link or repository inaccessible; ○: NOT Covered;

identifie the parts-of-speech (POS) of each word in the preprocessed data set and expands the extracted terms through a lexical database of English words called WordNet [34]. The expanded user input information is then passed into query matching for similarity scores, and the rest of the searching process matches the process introduced in Figure 4.1. Free-form query search generates a set of small and highly relevant code snippets, and effectively reuses them by eliminating the requirement of much prior knowledge about APIs. This ability not only increases the performance, but also the reliability of the searched code.

4.6 Applying Deep Learning to Search Code

Deep learning techniques are becoming increasingly common as building blocks of code search engines. The value of these techniques lies in their ability to accurately capture the most salient code features while incurring only modest computational costs. Deep learning based engines work particularly well for ***Type II*** usage scenarios. Table 4.3 shows a summary of representative deep learning engines.

4.6.1 Deep Learning Models

When it comes to handling large volumes of data, deep learning (DL) is considered one of the most effective techniques. Although most commonly applied to image classification, DL has recently been applied successfully to solving problems in software and programming. BAYOU [36] and DeepCoder [7] use deep learning models to solve problems in program synthesis. Numerous approaches have also applied DL to detect bugs [35, 37, 49], predict code types and properties [19, 39], and represent code in vector space [1, 2, 20].

Researchers and practitioners alike have also explored the possibility of applying DL to various code searching tasks. Although traditional code search engines can effectively match target code against given input via text similarity, their unawareness of program semantics often leads to search results that are either repetitive or inaccurate. In contrast, DL-based code search engines allow retrieval of semantically accurate code fragments that precisely match user input. To that end, these engines employ the concept of *embedding*, representing input as shared vector space. Then the search process involves computing vector similarity across the embeddings.

4.6.2 Specific approaches

As mentioned above, an embedding approximates the semantic correlation across the searched code and the input information. When it comes to learning these embeddings, the prevailing approaches fall into two categories: *(1) unsupervised* and *(2) supervised* methods.

Unsupervised approaches only rely on a corpus of code. The vector representation of code is stored, so the words sharing similar context are located closer to each other in the vector space. word2vec [33] is a two-layer dense neural network model that computes vector representation of words on a large corpus. It generates a lossy encoding of words in a lower-

dimensional vector space. These embedding vectors retain each word's meaning with proper dimensionality and training. FastText [12], a variant of word2vec, builds word embeddings based on a skipgram model, representing each word as a bag of character n-grams, and associating a vector representation with each character n-gram. NCS [41] combines FastText with FAISS [21], an efficient comprehensive similarity search algorithm, to achieve highly accurate search results. To further improve searching performance, NQE [25], a query expansion extension (encoder-decoder) model for NCS, uses a neural model that inputs a set of keywords and predicts an extended set of keywords with the goal of expanding the input query to NCS. NQE's encoder converts an input query into an embedding and then feeds it into a Recurrent Neural Network (RNN) decoder to compute the probability distribution of next output. To maximize accuracy, the decoder uses a beam search and an attention mechanism.

Supervised approaches embed code snippets joined with their natural language descriptions into a vector space. CODEnn, a representative of supervised code search models, uses a sequence-to-sequence-based recurrent network to embed both the code and their natural language description into a high-dimensional unified vector space [18]. It can effectively retrieve code snippets based on their vectors. Given the preliminary success of CODEnn's supervised learning, some researchers have pointed out some of its problems, such as slow training speed and semantic inaccuracy in the absence of an accurate code description [13, 48]. UNIF's bag-of-words-based network significantly lowers complexity, as compared to sequence based networks [13]. Similarly, COSEA uses a parallelized convolutional neural network (CNN) to increase training speed. COSEA also solves the question of accurately capturing semantic information by leveraging the CNN with layer-wise attention to capture the valuable code's intrinsic structural logic [48].

4.7 Discussion and Findings

Based on the coverage of the code search engines in Sections 3.3–3.5, we next discuss our findings.

Finding-1: Code-to-code Engines:

- Code-to-code search engines best apply to find code duplicates in a large codebase in order to fix bugs or get usage examples, with both token-based and tree-based approaches showing promising results.
- Although token-based approaches usually exhibit lower execution speed due to their high computational cost, combining the bag-of-tokens approach with SourcerCC's sub-block overlap filtering [43] enables token-based strategies to reach high accuracy and execution speed.
- Tree-based approaches show outstanding performance for all types of code clones; approximate searches with inexact matches show the best performance.

Finding-2: Natural language-to-code Engines:

- Natural language-to-code search engines best apply when a functionality can be described verbally, but it is unclear how implement it.
- Code-to-code and language-to-code engines would return dissimilarly formatted results for the same search scenario: the former would return all matches of the given code snippet, while the latter would return a variety of implementation examples.
- The two main varieties of language-to-code engines are NLP-based and DL-based. 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. In contrast, DL-based engines show better performance as they match input by accurately modeling query dependencies.

- Due to an extra layer of natural language embedding, supervised learning can enable better accuracy for DL-based engines; however, supervised learning DL engines are yet to achieve higher accuracy than NCS, a state-of-the-art unsupervised engine.

Finding-3: Training Data Matters: The quality and quantity of training data for DL models can greatly impact the search results of DL-based engines[45]. 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.

Finding-4: Need More Performance Metrics: We also notice that some aspects of search 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-based and DL-based engines never considered this metric. Input length—ranging from small (a few words) to large (multiple lines of code or sentences)—can also affect an engine's search performance. Out of all the engines we studied, only NQE [25] tested its performance specifically against small inputs, and no other engines tested against different ranges of input length.

Finding-5: 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, SourcerCC, Yogo, CCFinder, CODEnn), and 2 of them 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.

Chapter 5

Developers' Perspectives on Code Search Engines

In this section, we describe the survey we conducted to understand the perspectives of software developers on code search engines. We explain the survey's methodologies, data collection, and results. Finally, we discuss the new insights that we extracted from the survey's results.

Table 5.1: 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
Q5 - How much do you agree with the following statement: 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)

5.1 Survey Methodologies

1. Selection of participants. 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, to take our survey, the participants must have had some coding experience, albeit of different lengths. An important goal of our survey was also to understand if the length of a developer's programming experience might influence their usage of and preferences for code search engines. Hence, we had to ensure that our survey takers would come from diverse coding backgrounds and possess dissimilar levels of programming expertise. To that end, we invited hundreds of developers to take our survey. The developers who ended up accepting our invitation to take the survey included employees of a renowned IT company as well as graduate and undergraduate CS and ECE students. When asked about their primary programming language, our survey takers reported a wide variety of languages that ranged from Java to MATLAB.

2. Survey questions and their purpose. Table 5.1 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 of a developer's programming experience: 0 to 2, 2 to 5, 5 to 10, and more than 10 years. For Q2, we provide seven options, six for popular programming languages and one for user-customized input. These languages include Python, Java, JavaScript, C, C++, and Scratch. These two questions identify a developer's programming background, and we analyze these answers to understand how a developer's background might affect their usage of and preferences for code search engines.

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,

and which engines are the most popular. For those developers who claim not to use any code search engine, we aim at understanding why they find search engines unnecessary. To that end, 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," and "I am too busy to learn how to use a search engine."

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. (Code input)," "I want to implement a certain functionality but do not know how, so I'd like to search for code that matches my needs. (Natural language input)," "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 agreement levels range from "strongly disagree" to "strongly agree."

5.2 Survey Results and Findings

Recall that our survey collected information about developers' technical background, usage of, and preferences for code search engines. We discuss our survey results in turn next.

1. Technical background: For Q1 and Q2, we obtained 114 valid responses from our survey takers, software developers with diverse 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. To our surprise, a considerable amount of participants (44.25%) do NOT use code search engines in their programming practices. When asked as to why they do not use code search engines, 75% of them pointed out that they were unaware of code search engine existence, 10% complained that the engines fail to return useful results, 2.5% explained that their business prevented them from learning how to use a code search engine, 7.5% said they exclusively use Google or Github to search for code, and 5% did not provide a reason for not using a search engine.

Finding-1: Code search engines need better publicity: Among the surveyed developers who claimed NOT to use code search engines (44.25%), the majority (75%) did not know that search engines existed.

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% for OpenGrok or Grok, 11.11% for others (e.g., Gerrit, eclipse, Intellij), and 19.04% left unspecified which engine they used.

Finding-2 Code search engines need to be defined more precisely: Many of the developers who claim to use code search engines, in fact use general-purpose search engines or code repositories.

We obtained 90 responses for Q4. Based on these responses, we 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 code search 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 the common usage scenarios: (a) having an unfamiliar code snippet whose usage is unclear or problematic, and (b) needing to implement an unfamiliar functionality.

3. Preferences for properties of a code search engine: Figure 5.1 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%),

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

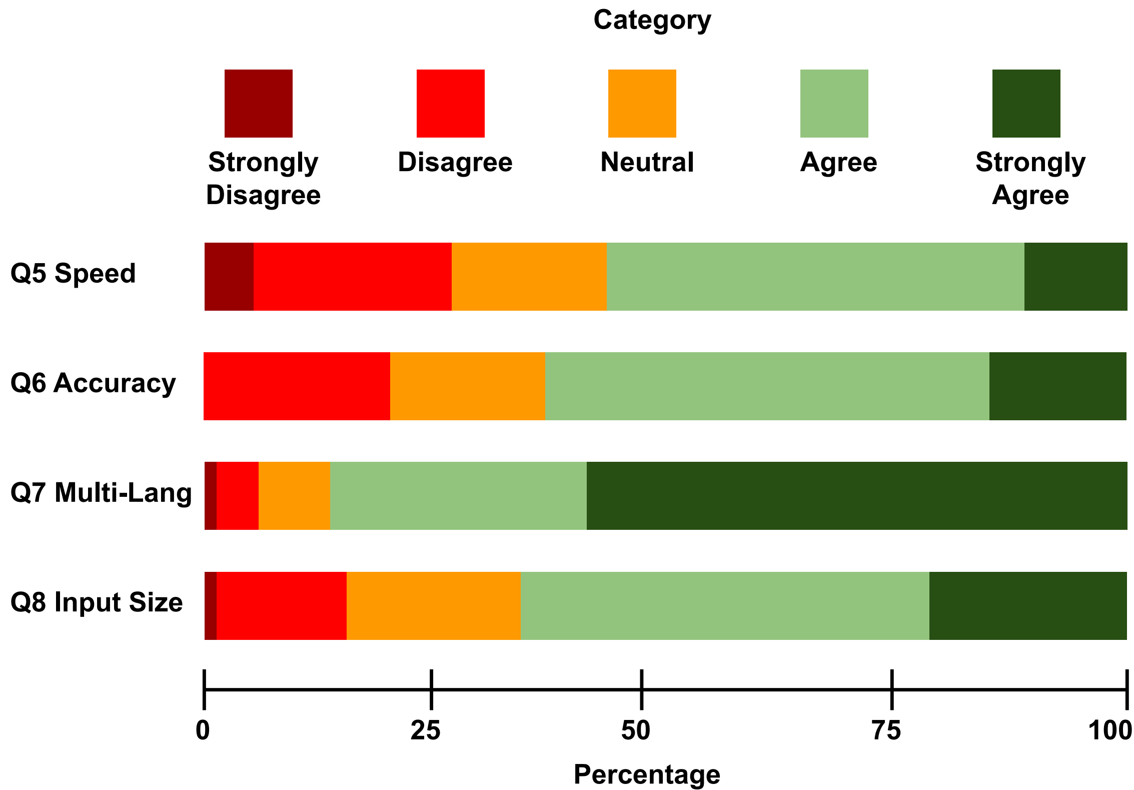

Figure 5.1: Agreement Levels of Q5-Q8

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 ability to support multiple languages and input of all sizes is as important as achieving high speed and accuracy: Developers prioritize the importance of supporting multiple languages and input of all sizes over the ability to deliver high accuracy results quickly.

5.3 Discussion and Insights

Supporting multiple programming languages has a great potential benefit as a future work direction for code search engines As we discovered, the majority of our survey takers agreed that a code search engine should support multiple programming languages to maximize its utility. To that end, one possible strategy is *input normalization*: converting the input's code snippets to an intermediate language-agnostic representation. For example, an engine can represent its input as an Abstract Syntax Tree (AST) or an LLVM Intermediate Representation (LLVM IR). Since the code written in different languages can be converted into such a language-agnostic representation, the necessity to support different languages would no longer require an extra engineering effort proportional to the number of languages that a search engine supports. Furthermore, an engine can *vectorize* its input (code snippets or natural language text) by applying word embedding techniques. Having vectorized the input, a code search engine can operate without taking into account whether the original input was provided in natural or programming languages.

Threats to Validity The internal validity is threatened by a lack of a commonly accepted definition of the term *a code search engine*. In lieu of such definition, our survey takers might not have differentiated true code search engines from general-purpose engines. Hence, our findings may be based on the takers' experiences with both code and general-purpose search engines. This realization prompted us to state our definitions and assumptions up front.

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.

Chapter 6

Future Work

As discussed in the previous chapters, this thesis approaches the topic of code search engines from two difference perspectives: (1) the state of the art as depicted in the research literature as well as major existing engines, and (2) developer perspectives as assessed by their knowledge and expectations. We see as the most actionable contribution of this thesis is our attempt at reconciling these perspectives to present our findings. Both researchers and practitioners can benefit from our findings, as they inform about the standard workflow of a code search engine, main search engine types and strategies, as well as the prevailing expectations that developers have for code search engines. We also hope that our findings can be of use for researchers in identifying promising future research directions.

Nevertheless, it would be unrealistic for a single thesis to cover all aspects of code search engines, approached from all possible angles. Having built a fundamental knowledge base of code search engines, this work leaves out several promising future work directions, some of which we outline next:

1. *Vertical knowledge.* Although we provide horizontal knowledge, which covers all main existing mainstream code search strategies, we only study the most representative engines and their techniques for each strategy. To deeper understand specific search strategies, each of them could be studied in greater depth in order to uncover possible new techniques.

2. *Control variables.* Code search engines we investigate in this thesis are trained (for DL engines) and tested on different datasets due to language-specific and/or method-specific requirements. It is likely that dataset quality can greatly affect the performance of search engines. To address this problem, one can implement a modular search engine, in which each search component would be replaced with different modules, so the training and testing data would be set consistent while control variables would be set for each component.

3. *Multi-language support for DL engines.* Although DL engines are becoming increasingly popular as evidenced by the recent publications, the number of multi-language supporting DL engines is limited due to the massive training required to support multiple programming languages. To better inform researchers and meet developer expectations, one can design and implement a model that learns and categorizes lexical and semantic structures for multiple mainstream programming languages.

Chapter 7

Summary and Conclusions

In this thesis, 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. Moreover, we identified four insights that can be helpful in guiding future research. 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.

Bibliography

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

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

[4] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina 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*, pages 681–682, 2006.

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

[6] Brenda S Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 86–95. IEEE, 1995.

[7] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[8] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[9] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9):577–591, 2007.

[10] ben boyter. searchcode, 2022. https://searchcode.com/.

[11] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[12] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[13] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish 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*, pages 964–974, 2019.

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

[15] KR1442 Chowdhary. Natural language processing. *Fundamentals of artificial intelligence*, pages 603–649, 2020.

[16] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph

and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[17] Vinicius C Garcia, Eduardo S de Almeida, Liana B Lisboa, Alexandre C Martins, Silvio RL Meira, Daniel Lucrédio, and Renata P de 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)*, pages 61–70. IEEE, 2006.

[18] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

[19] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 152–162, 2018.

[20] Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 163–174, 2018.

[21] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2019.

[22] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE transactions on software engineering*, 28(7):654–670, 2002.

[23] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *International static analysis symposium*, pages 40–56. Springer, 2001.

[24] Lisa Vaas. Samsung shattered encryption on 100m phones, 2022. https://threatpost.com/samsung-shattered-encryption-on-100m-phones/178606/.

[25] Jason Liu, Seohyun Kim, Vijayaraghavan Murali, Swarat Chaudhuri, and Satish Chandra. Neural query expansion for code search. In *Proceedings of the 3rd acm sigplan international workshop on machine learning and programming languages*, pages 29–37, 2019.

[26] Julie Beth Lovins. Development of a stemming algorithm. *Mech. Transl. Comput. Linguistics*, 11(1-2):22–31, 1968.

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

[28] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28, 2019.

[29] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun 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)*, pages 260–270. IEEE, 2015.

[30] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *icsm*, volume 96, page 244, 1996.

[31] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, 2011.

[32] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120, 2011.

[33] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

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

[35] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian specification learning for finding api usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 151–162, 2017.

[36] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. *arXiv preprint arXiv:1703.05698*, 2017.

[37] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

[38] Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1066–1082, 2020.

[39] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from" big code". *ACM SIGPLAN Notices*, 50(1):111–124, 2015.

[40] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William E. Lorensen, et al. *Object-oriented modeling and design*, volume 199. Prentice-hall Englewood Cliffs, NJ, 1991.

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

[42] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 191–201, 2015.

[43] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168, 2016.

[44] Xiaobing Sun, Xiangyue Liu, Jiajun Hu, and Junwu Zhu. Empirical studies on the nlp techniques for source code data preprocessing. In *Proceedings of the 2014 3rd international workshop on evidential assessment of software technologies*, pages 32–39, 2014.

[45] Zhensu Sun, Li Li, Yan Liu, and Xiaoning Du. On the importance of building high-quality training datasets for neural code search. *arXiv preprint arXiv:2202.06649*, 2022.

[46] Stephen W Thomas. Mining software repositories using topic models. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1138–1139, 2011.

[47] Vera Wahler, Dietmar Seipel, J Wolff, and Gregor Fischer. Clone detection in source

code by frequent itemset techniques. In *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*, pages 128–135. IEEE, 2004.

[48] Hao Wang, Jia Zhang, Yingce Xia, Jiang Bian, Chao Zhang, and Tie-Yan Liu. Cosea: Convolutional code search with layer-wise attention. *arXiv preprint arXiv:2010.09520*, 2020.

[49] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 708–719, 2016.

[50] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

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

Appendices

Appendix A

Survey Material

A.1 Survey Info

Information Sheet for Participation in a Research Study

Principal Investigator: Eli Tilevich
IRB# and Title of Study: IRB# 423 - Comparing the Apples and Oranges of Code Search: a Survey of State-of-the-Art Code Search Engines

You are invited to participate in a research study. This form includes information about the study and contact information if you have any questions.

I am a graduate student at Virginia Tech, and I am conducting this research as part of my course work.

- **WHAT SHOULD I KNOW?**

If you decide to participate in this study, you will complete a ***survey.*** *As part of the study, you will complete a quick survey that contains 10 multiple choice questions asking you about your experience using code search engines. Only your answers to the survey questions will be recorded, and all your personal information will be kept anonymous.*

The study should take approximately 10 minutes of your time.

We do not anticipate any risks from completing this study.

You can choose whether to be in this study or not. If you volunteer to be in this study, you may withdraw at any time without consequences of any kind. You may also refuse to answer any questions you don't want to answer and remain in the study. The investigator may withdraw you from this research if circumstances arise which warrant doing so.

- **CONFIDENTIALITY**

We will do our best to protect the confidentiality of the information we gather from you, but we cannot guarantee 100% confidentiality.

Your responses are anonymous, so no one can associate your answers back to you. Please do not include your name or other identifying information in your responses that can identify you.

- **WHO CAN I TALK TO?**

If you have any questions or concerns about the research, please feel free to contact Sherry via amnos@vt.edu. You are not waiving any legal claims, rights or remedies because of your participation in this research study. If you have questions regarding your rights as a research participant, contact the Virginia Tech HRPP Office at 540-231-3732 (irb@vt.edu).

Please print out a copy of this information sheet for your records.

If you would like to participate in this survey, click on the following link to begin or no to exit.

https://virginiatech.qualtrics.com/jfe/form/SV_1BuvexvB15pz7dI

A.2 User Survey

Default Question Block

Q1.
How long have you been writing code?

- 0-2 years
- 2-5 years
- 5-10 years
- 10+ years

Q2. What is your primary programming language?

- Python
- Java
- JavaScript
- C
- C++
- Scratch
- Other

Q3.
Do you use a code search engine in your programming pursuits?

- Yes
- No

Q4. If yes, then which one?

Q5. If no, why not?

- 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
- Other

Q6. How much do you agree with the following statement: when using a code search engine, how fast it returns its results is the most important criteria

- Strongly agree
- Somewhat agree
- Neither agree nor disagree
- Somewhat disagree
- Strongly disagree

Q7.
Which of the following scenarios best describes how you typically use a code search engine:

- 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. (Code input)

- I want to implement a certain functionality but do not know how, so I'd like to search for code that matches my needs. (Natural language input)
- I use code search engines for both of the two scenarios above -- a.) and b.).
- I never use code search engines in my programming practices
- I use code search engines in other scenarios. [provide a reason]

Q8. Only a highly accurate search engine would be helpful in my software development activities

- Strongly agree
- Somewhat agree
- Neither agree nor disagree
- Somewhat disagree
- Strongly disagree

Q9. It is important for a search engine to support multiple programming languages

- Strongly agree
- Somewhat agree
- Neither agree nor disagree
- Somewhat disagree
- Strongly disagree

Q10.

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)

- Strongly agree
- Somewhat agree
- Neither agree nor disagree
- Somewhat disagree
- Strongly disagree

Powered by Qualtrics

A.3 IRB Approval

Shuangyi Li <amnos@vt.edu>

IRB #21-423: Approval Letter

1 message

VT IRB Protocol Management <irb@vt.edu> Fri, Jun 4, 2021 at 8:11 AM
To: Liesl Baum Walker <lmbaum@vt.edu>, Eli Tilevich <tilevich@vt.edu>, Shuangyi Li <amnos@vt.edu>, Yin Liu <yinliu@vt.edu>

"Liesl Baum Walker" <lmbaum@vt.edu>,"Eli Tilevich" <tilevich@vt.edu>,"Shuangyi Li" <amnos@vt.edu>,"Yin Liu" <yinliu@vt.edu>
Dear Researcher:

The Virginia Tech Institutional Review Board (IRB) has approved the IRB application referenced in the attached approval letter for the protocol titled "Comparing the Apples and Oranges of Code Search:a Survey of State-of-the-Art Code Search Engines". Read the approval letter carefully as it contains IRB-related requirements and retain a copy for your records.

Please use the attached IRB-stamped documents.

Visit the following link to request an amendment to approved IRB application materials, and to report unanticipated problems: https://secure.research.vt.edu/irb/

The IRB wishes you success with your research.

HRPP office

We'd greatly appreciate your feedback with regard to your experience related to your recent IRB protocol submission. Please consider completing our short (under 10 questions) "Protocol-specific feedback" survey: https://secure.research.vt.edu/external/irb/review-survey

3 attachments

VT IRB-21-423 Approval Letter.pdf
99K

information-sheet-for-studies-without-consent (17).pdf
681K

Filled-IRB-21-423-Research Protocol (HRP 503).pdf
1496K