Generating Clarifying Questions for Query Refinement in Source Code Search

1st Anonymous Authors
Affiliation
Organization
City, Country
email address

***Abstract*—In source code search, a common information-seeking strategy involves providing a short initial query with a broad meaning, and then iteratively refining the query using terms gleaned from the results of subsequent searches. This strategy requires programmers to spend time reading search results that are irrelevant to their development needs. In contrast, when programmers seek information from other humans, they typically refine queries by asking and answering clarifying questions. Clarifying questions have been shown to benefit general-purpose search engines, but have not been examined in the context of code search. We present a method for generating natural-sounding clarifying questions using information extracted from function names and comments. Our method outperforms a keyword-based refinement method in synthetic and human evaluations.**

***Index Terms*—source code search, code retrieval, clarifying questions, query refinement, software maintenance**

I. Introduction

Source code search (SCS) engines are systems that return lists of source code fragments that are relevant to user-provided search queries [1]. Programmers use SCS engines during software maintenance for feature location, code reuse, bug triage, and more [2]. However, programmers often fail to find the information they need with a single search [3]. Search issues can be caused by a mismatch between the concepts and terms humans use to describe programming tasks, and those that search engines associate with relevant code [4]–[6].

When an initial search is unsuccessful, programmers can perform **query refinement**, which refers to the process of modifying a search query in order to retrieve more-relevant results. Over the course of a search session, programmers use information gleaned from previous search results to refine subsequent queries [3]. Many source code search engines offer features to improve the query refinement process, such as suggesting relevant keywords [7].

By contrast, when programmers seek information from other humans, they typically refine queries by asking and answering **clarifying questions** [8]–[10]. Clarifying questions (CQs) are questions intended to confirm or elicit information about some aspect of a query [11]. If a novice programmer were to ask an expert programmer for a function to “convert a float”, the expert would be uncertain whether to suggest a function that, e.g., converts a float to an int, converts a float to a string, or converts another data type to a float. By asking a clarifying question such as “Would you like to convert a float to an int or a string?” or “Would you like to convert a float, or convert something to a float?”, the expert would then be able to suggest code implementing the correct functionality.

An emerging trend in software engineering literature is the idea that tool support should emulate the kinds of support offered by human programmers [12]–[15]; in particular, leading researchers have championed technology to help programmers better express their information needs in search queries [16]. Meanwhile, a growing body of work supports the use of CQs for query refinement across a broad range of domains [11], including software engineering [14].

Nevertheless, clarifying questions remain understudied in the context of source code search. The more-general problem of query refinement in SCS has been studied extensively by Hill *et al.* [17], Wang *et al.* [18], Treude *et al.* [19], and Martie *et al.* [7]. Most existing approaches to query refinement rely on GUI elements (such as drop down menus or lists of tags), which are able to present numerous options for refinement simultaneously. By contrast, a CQ is communicated via natural language and must target a narrow aspect of the user’s query to clarify [20]. The challenge is that there is no clear way to identify potentially-relevant aspects of a SCS query or select which aspect to ask about.

In this paper, we propose an approach to interactively refine SCS queries using clarifying questions. Our approach discerns relevant aspects of a SCS query from the similarities and differences between the search results that query produces. We divide our approach into three components: 1) Identifying potentially ambiguous aspects of a code search query, 2) Selecting a query aspect to inquire about and generating a grammatically-appropriate CQ, and 3) Reranking the search results based on the user’s answer.

Our evaluation methodology is twofold: First, we perform a synthetic evaluation using the CodeSearchNet [21] dataset, which contains relevance ratings for code search results. This evaluation demonstrates that our approach quickly improves the rankings of relevant results. Second, we perform intrinsic and extrinsic human studies. We hire 10 programmers to rate the quality of CQs generated by our approach, and we hire 12 programmers to complete code search tasks aided by a CQ query refinement engine. We find that CQs reduce search duration compared to a keyword recommendation baseline.

To promote reproducibility, we release all relevant code and data via an online appendix (see Section V-F).

II. Background and Related Work

A. *Clarifying Questions for Query Refinement*

Clarifying questions can help information-retrieval (IR) systems resolve ambiguous queries in one of two ways: by **confirming** the system's interpretation of the user's information need (e.g., "is this what you are searching for?"), or by **eliciting** a missing piece of information (e.g., "which of these are you searching for?") [22]–[27].

A growing body of evidence indicates that these questions can improve user experience in IR systems; for instance, Bing users reported higher levels of confidence in their search results after answering CQs to refine their queries [24]. Users also have high rates of engagement with CQs [23], particularly compared to query suggestions that are not formatted as questions [24]. Other studies have observed CQs information-seeking conversations among programmers [10], [28], [29]. Gao *et al.* [30] analyzed over 2M posts on technical Q/A sites; they found that a large number of comments on posts contain CQs, and that posts with CQs were more likely to receive a correct answer than those without.

Despite these observations, few query refinement approaches in software engineering literature actually attempt to generate CQs. A key obstacle is that methods to generate CQs in broader domains rely on data that are not readily available for SCS. They typically work by identifying query **aspects** and query **facets** to enquire about [31]. A query aspect is a word or phrase describing a distinct information need relevant to a query; for example, given the query "JPEG image", one aspect could be *ways to process JPEG*. A query facet is a set of terms sharing a semantic relationship to a query aspect; the facet corresponding to the *ways to process JPEG* aspect could include {*convert*, *rotate*, *resize*}. A more specific aspect would be *image types to which JPEG can be converted*, with the facet {*PNG*, *GIF*, *TIFF*}. Those facet terms might also apply to the aspect *image types that can be converted to JPEG*.

Aspects and facets serve as logical categories and options for query refinement; furthermore, because they are defined by semantic roles, they can be used to generate grammatically-correct CQs [31]. For instance, the aspect *image types to which JPEG can be converted* could produce a question confirming the user's information need: "Do you want to convert a JPEG to a different image type?". Or, it could elicit a missing facet value for that aspect: "To which image type would you like to convert a JPEG? a) PNG, b) GIF, c)TIFF."

The main challenge for SCS is that these query aspects and facets are not known in advance [32]. Methods exist to extract query aspects and facets dynamically for general-purpose web search queries using a) reformulation data derived from query stream mining [24], or b) semantic patterns found in the search results themselves [31]. The former approach requires a volume of reformulation data that is not available for SCS. The latter approach is more feasible, but must account for the fact that there may be limited quantities of natural language text associated with code snippets. A method for generating clarifying questions for SCS needs to identify relevant query aspects and facets, select ones that will allow for meaningful clarification, create a natural language question, and use the answer to improve the search results. To inform our approach, we consider other query refinement methods for SCS.

TABLE I
Selection of related work in query refinement for search tasks in software engineering. R=Relevance Feedback-based, F=Facet-based, C=Concept-based, T=Task-based. The NL column indicates methods with a natural language interface.

Project	Year	Domain	Method	NL
Gu *et al* [32]	2004	API	F	X
Shepherd *et al* [33]	2007	Code	T	
Gay *et al.* [34]	2009	Code	R	
Eisengerb *et al* [35]	2010	API	F	
Hill *et al* [36]	2011	Code	T	
Roldan-Vega *et al* [37]	2013	Code	C, T	
Wang *et al* [18]	2014	Code	R	
Treude *et al* [19]	2014	API	C, T	
Martie *et al* [7]	2017	Code	F, C	
Li *et al* [38]	2018	API	C	
Sivaraman *et al* [39]	2019	Code	R	
Zhang *et al* [14]	2020	Web	F	X
Xie *et al* [40]	2020	API	T	
Eberhart and McMillan [13]	2021	API	C	X

B. *Query Refinement in Software Engineering*

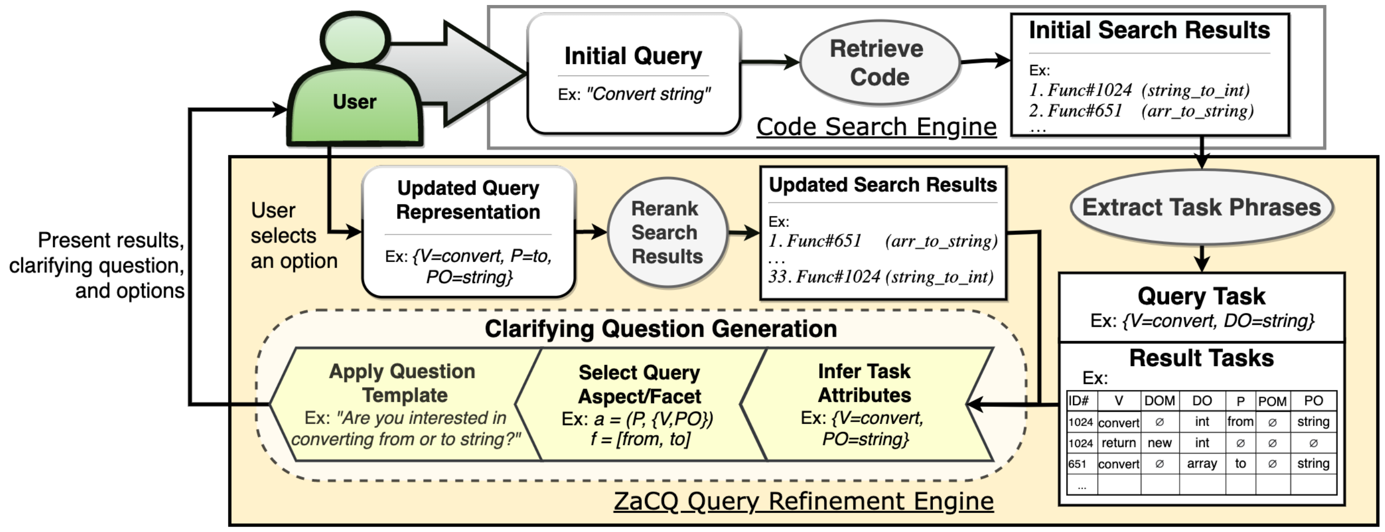
Table I summarizes key related work in query refinement for information retrieval in software engineering. We focus on four categories: *relevance feedback*-based, *facet*-based, *concept*-based, and *task*-based methods.

Relevance feedback-based methods simply ask users to rate individual search results as relevant or irrelevant. Gay *et al.* [34] and Wang *et al.* [18] implement relevance feedback using the Rocchio algorithm [41] to update a vector representation of the search query and rerank the results.

Facet-based methods let users filter the search results by selecting facet values for predefined query aspects. In SCS, these aspects are limited to explicitly-defined properties of the source code (e.g., return type, parameter types, parent class). Zhang *et al.* [14] proposed a facet-based method to generate targeted clarifying questions for StackOverflow post retrieval. However, this method relied on existing technical tags that SO users had assigned to the posts, and required the authors to manually identify 20 query aspects and categorize over 3700 tags into corresponding facets.

Concept-based methods overcome these limitations by extracting discriminative features from the search results. The simplest concept-based methods are keyword recommendation algorithms. Poshyvanyk and Marcus [42] use Latent Semantic Indexing (LSI) to find important keywords for a set of code snippets; they then use formal concept analysis (FCA) to let users iteratively filter their searches with increasingly-specific keywords. Other methods extract certain syntactic patterns from source code or documentation to recommend entire noun phrases (e.g., "JPEG image") [7], [19], [37].

Task-based methods extend this idea to verb phrases (e.g., "convert JPEG image to PNG"), enabling users to specify more-complex types of functionality. We refer to these


User
Initial Query
Ex: "Convert string"
Retrieve Code
Initial Search Results
Ex:
1. Func#1024 (string_to_int)
2. Func#651 (arr_to_string)
...
Code Search Engine
Present results, clarifying question, and options
User selects an option
Updated Query Representation
Ex: {V=convert, P=to, PO=string}
Rerank Search Results
Updated Search Results
Ex:
1. Func#651 (arr_to_string)
...
33. Func#1024 (string_to_int)
Extract Task Phrases
Query Task
Ex: {V=convert, DO=string}
Result Tasks
Ex:
Clarifying Question Generation
Apply Question Template
Ex: "Are you interested in converting from or to string?"
Select Query Aspect/Facet
Ex: a = (P, {V,PO}) f = [from, to]
Infer Task Attributes
Ex: {V=convert, PO=string}
ZaCQ Query Refinement Engine

ID#	V	DOM	DO	P	POM	PO
1024	convert	∅	int	from	∅	string
1024	return	new	int	∅	∅	∅
651	convert	∅	array	to	∅	string
...						

Fig. 1. Overview of the ZaCQ method for natural language query refinement.

functionality verb phrases as development tasks, or simply **tasks** [19]. Shepherd *et al.* [33] first developed an approach to use verbs and direct objects extracted from function identifiers and comments for interactive query reformulation. Hill *et al.* [36] presented a subsequent technique to generate tasks comprising verb, noun, and prepositional phrases from code snippets, and built a code navigation tool based on the resultant semantic hierarchies. Treude *et al.* [19] developed a technique to extract the same task phrases from API documentation by searching for specific syntactic patterns.

We observe that task phrases convey the similarities and differences among code snippets by way of the same structured semantic relationships that connect aspects and facets. Our approach takes advantage of this property to dynamically extract aspects and facets for clarifying question generation.

III. Approach

This section presents our approach: the *Zero-aspect Clarifying Question* (**ZaCQ**) system. ZaCQ is a natural language query refinement engine that works alongside a standard SCS engine. It targets the scenario where a SCS engine retrieves one or more results that satisfy a user's information need, but ranks them below other, less relevant results. Given a SCS query and results, ZaCQ generates a targeted clarifying question and provides options for refinement. Importantly, *ZaCQ does not rely on predefined query aspects*; instead, it derives potentially-relevant aspects and corresponding facets from tasks associated with the search results. Over multiple rounds of refinement, ZaCQ identifies a single development task relevant to the user's information need.

A. Overview

Figure 1 presents a high-level overview of our approach, initiated by the user in the upper-left corner. The white box to the right of the user represents a standard SCS engine. The user provides a text query to the SCS engine, which produces a ranked list of the top-k most-relevant functions from a source code repository. These results serve as the input to ZaCQ.

First, ZaCQ uses natural language processing techniques to extract tasks from the query and search results (Section III-B).

Second, ZaCQ generates a clarifying question based on the extracted tasks (Section III-C). In order to create a question targeting an unclear aspect of the user's information need, it attempts to infer probable task attributes (e.g., common actions or objects) from the query and results. It then selects a salient query aspect to enquire about – either by seeking confirmation that it is relevant to the user, or eliciting a value from the corresponding facet. Based on the chosen aspect/facet, ZaCQ applies an appropriate template to create a CQ.

Third, ZaCQ presents the search results, CQ, and refinement options. The user may select an option to refine his/her query. ZaCQ creates an updated query representation based on the user's selection, which it uses to rerank the search results (Section III-D). If aspects of the user's information need are still unclear, the process repeats, and the system generates a new CQ targeting a different query aspect.

B. Task Extraction

ZaCQ begins by generating tasks associated with a user query q and the corresponding list of search results R. A task t is represented as a data structure with string values for up to six syntactical roles: a verb (V), a direct object modifier (DOM), a direct object (DO), a preposition (P), a prepositional object modifier (POM), and a prepositional object (PO). A value for a particular syntactic role s is referred to as a task attribute t_s. Some attributes may be missing, but each task must have an value for V and either a DO value or P and PO values.

We extract tasks from q and R using the method published by Treude *et al.* [19]. We refer readers to the original paper for algorithmic details, and to our repository (see Section V-F) for implementation details; here, we briefly summarize the method and adaptations made for our approach.

1) Preprocessing: For each function in R, we attempt to extract tasks from the function name and comment (if there is one). We split camelcase and snakecase function names into separate tokens and prepend them to the comment string with a period. We then preprocess the comment, adapting the steps used by Treude *et al.* for docstring-specific text formatting.

2) Dependency Parsing: We use *spaCy* [43] to parse syntactic dependencies. Extracting a task involves following specific dependencies and recording tokens in certain syntactic roles. All tasks start with a verb; we follow dependencies to find values for DO, P, and PO roles, and then continue parsing from objects to find DOM/POM modifiers. One modification we make is to follow prepositional dependencies on objects denoting collections of elements (e.g. "list of ints"); we append these phrases to the corresponding t_{DO} or t_{PO} strings.

3) Postprocessing: Like Treude *et al.*, we filter out tasks with attributes that are too generic to make for valuable clarification targets. We filter out a list of generic nouns, which including generic programming terms (e.g., "parameter", "function"), as well as a small list of generic verbs (e.g., "take", "be", "do", "have").

Using this approach, we attempt to extract a task from q, and as many tasks as possible from each function in R. We store tasks extracted from R in a task table T, where each row corresponds to a single task for a single result.

C. Clarifying Question Generation

Next, we use the task table T and user query q to generate a natural language CQ targeting a particular query aspect a. We define an aspect as a target syntactic role s given a set of defined task attributes d; e.g., the aspect *image types to which JPEG can be converted* would be represented as (s=POM, d={V=*convert*, DO=*jpeg*, P=*to*, PO=*image type*}). Because query aspects are not known in advance, we use the tasks in T to identify potential aspects. The set of all query aspects A_q is defined as $A_q = \{(s, d) \mid s \in S_d, d \in D\}$, where D is the set of all subsets of task attributes for each task in T and S_d is the set of all syntactic targets for d.

Possible syntactic targets include the six syntactic roles that define tasks, as well as three special types: 1) object (O), which targets a DO or PO, 2) object modifier (OM), which targets a DOM or POM, and 3) object role (OR), which targets a minimal verb phrase including a V and either a DO or a P and PO. The O and OM types enable ZaCQ to enquire about relevant objects without concern for the specific syntactic roles they play. The OR type allows ZaCQ to distinguish whether an O attribute serves as a DO or a PO.

We specify a set of rules to determine S_d for a given d. The purpose of these rules is to avoid asking users confusing or difficult CQs; e.g., when d is empty, a CQ should not elicit a preposition from the user. The rules are as follows:

```
1: **if** $d = \emptyset$ **then** $S_d \leftarrow \{\text{V}, \text{O}\}$
2: **else if** V $\notin d$.roles **then** $S_d \leftarrow \{\text{OM}, \text{OR}\}$
3: **else** $S_d \leftarrow \{\text{DO}, \text{P}, \text{PO}\}$ **end if**
4: **if** DO $\in d$.roles **then** $S_d \leftarrow S_d \cup \{\text{DOM}\}$ **end if**
5: **if** PO $\in d$.roles **then** $S_d \leftarrow S_d \cup \{\text{POM}\}$ **end if**
6: $S_d \leftarrow \{S_d \setminus d.\text{roles}\}$
```

ZaCQ can generate two kinds of CQs for an aspect: CQs confirming the aspect's relevance to the user's information need, and CQs eliciting a value for the aspect's syntactic target s. When eliciting a value, ZaCQ presents up to n options from the corresponding query facet f_a. The facet comprises the set of unique t_s attributes in T_d, where $T_d = \{t \in T \mid d \subseteq t\}$. The type of CQ generated depends on the number of options presented: $\leq 1 \rightarrow$ confirmation, $\geq 2 \rightarrow$ elicitation.

In order to select an appropriate query aspect/facet for a CQ, ZaCQ follows a hand-crafted refinement strategy. First, ZaCQ chooses a set of defined attributes d for a query aspect. A cautious refinement strategy would be to use a d comprising only the attributes that the user had previously accepted. However, this may not be the most efficient strategy, as it can lead to ZaCQ explicitly clarifying attributes that should be obvious in context, e.g. an attribute that appeared in the task extracted from the user query, or one that appears in tasks in the majority of search results. ZaCQ balances caution and efficiency by inferring task attributes from T and q.

1) Attribute Inference: The goal of attribute inference is to populate d with attributes that are likely to be relevant to the user's information need. We use the frequency of an attribute in T_d as a proxy for relevance; e.g., an attribute that appears in tasks for 6 functions is considered twice as relevant as one that appears in 3 functions.

To infer attributes, ZaCQ first sets d equal to the set of previously-accepted attributes and gets S_d. It then considers attributes in the query facet for each candidate aspect. If any of the attributes also appear in the task extracted from q, that attribute is inferred. Otherwise, if the most-common attribute meets a minimum support and confidence threshold (specified as hyperparameters), it is inferred. If an attribute is inferred, ZaCQ adds it to d and repeats the process for the new S_d.

2) Aspect/Facet Selection: Next, ZaCQ uses a greedy algorithm to select a query aspect a and facet f_a to ask about. For a, ZaCQ selects the syntactic role s in S_d that has the single most-frequent attribute in T_d. It then selects the top-n most-common attributes in f_a to present as refinement options.

3) Question Templating: Finally, we apply a natural language CQ template to the selected aspect/facet. We define 5 templates for different syntactic targets:

T1) "Are you interested in [verb phrase]?"
T2) "Are you looking for [object phrase]?"
T3) "What kind of [object phrase] are you interested in [verb phrase/none]?"
T4) "How do you want to [verb phrase]?"
T5) "Found [#] items related to [object/verb phrase]. Would you like to see them first?"

In elicitation questions, the target syntactical role is replaced with "any of the following". CQs for aspects targeting V, DO, or PO use T1; O, DO, and PO use T2; OM, DOM, and POM use T3; and P uses T4. Most confirmation questions use T1 or T2; T5 is a special case for confirming attributes inferred directly from the task extracted from q.

D. Result Reranking

Users may respond to elicitation CQs by selecting an option or "None." For confirmation CQs, users may select "Yes" or "No." ZaCQ records all selected/confirmed attributes and sets of rejected attributes, and identifies lists of candidate functions (associated with at least one task that contains all accepted attributes and no rejected sets of attribute) and rejected functions (associated with no suitable tasks and at least one task with attributes that were explicitly rejected).

We note that SCS results may not necessarily have comments or function names from which tasks can be extracted, but may still be relevant to a query; therefore, ZaCQ does not directly filter non-candidates from the results. Instead, it promotes all functions similar to candidate functions and demotes those similar to rejected functions using the Rocchio algorithm [41]. This mechanism works for most SCS engines that embed functions and queries in the same vector space by creating an updated vector representation of the query, and then reranking each result by cosine similarity.

IV. Synthetic Evaluation

We performed a synthetic evaluation to determine whether clarifying questions generated by the ZaCQ query refinement engine are effective at improving the relevance rankings of source code search results. The evaluation involved generating CQs for sets of code search results, automatically selecting answers using relevance data, and recording the improvement observed in the overall ordering of the results.

We compared the ZaCQ method to two baseline methods: a method that asks users to clarify only a verb and a direct object (we refer to this method as *V-DO*), and a keyword recommendation method (we refer to this method as *KW*).

A. Research Questions

We ask the following research questions (RQs):

RQ_1 How well does ZaCQ perform after asking 1 or more questions, compared to the default result ordering?

The purpose of RQ_1 is to quantify ZaCQ's reranking performance and determine whether subsequent CQs after the first are less effective. In [18], the authors found that requesting additional relevance feedback for individual code search results yielded diminishing returns, so it is valuable to measure the utility of increasingly-specific CQs.

RQ_2 How well does ZaCQ perform after asking 1 or more questions, compared to the baseline methods?

The purpose of RQ_2 is to compare ZaCQ to the V-DO and KW baselines. While ZaCQ is designed to select query aspects and refinement options for natural-sounding CQs, it is also important that it be as efficient as the baselines.

B. Baselines

We compared the ZaCQ method to two baselines methods: a Verb-Direct Object (V-DO) method and a Keyword (KW) method. These represent existing concept-based and task-based approaches to interactive query refinement for SCS.

The V-DO baseline is based on methods used by Shepherd *et al.* [33] and Hill *et al.* [17]. It uses the same task extraction and architecture as ZaCQ, but it can only clarify values for the verb and direct object syntactic roles. Furthermore, it always elicits the two attributes in the same order (verb $\rightarrow$ object); it cannot infer attributes from the search results or decide to clarify the object first. This baseline is intended to highlight whether those features in ZaCQ (detailed query aspects, inference, and decision-making) meaningfully improve its performance.

The KW baseline is based on the refinement method proposed by Poshyvanyk and Marcus [42]. In brief, this method uses Latent Semantic Indexing to identify 25 keywords in the search results, and it uses Formal Concept Analysis to suggest discriminative keywords over several rounds of refinement. As users select keywords, they restrict the candidate functions and subsequent keyword suggestions for that query.

We implement both baselines use the same relevance feedback-based reranking algorithm as ZaCQ.[1]

TABLE II
CodeSearchNet dataset used for the syntethetic evaluation.

Language	Python	Java	PHP	Ruby	Javascript	Go
# Functions	1.2M	1.6M	1.0M	.2M	1.9M	.7M
# Queries in Evaluation	55	33	8	7	6	0
% Results with Task(s)	57.9	63.3	66.2	48.8	34.0	N/A

C. Dataset

Our evaluation dataset consists of 1) a set of 99 programmer queries, 2) SCS results for those queries, and 3) relevance ratings for those search results, all of which were derived from CodeSearchNet [21]. CodeSearchNet is a collection of datasets and benchmarks for code search evaluation. The datasets comprise about 6 million functions scraped from GitHub repositories, 2.1 million of which are paired with a comment. There are separate datasets for six programming languages: Python, Javascript, Ruby, Go, Java, and PHP.

1) Queries: CodeSearchNet provides a set of 99 general natural-language programming queries, such as "convert int to string" and "k means clustering". The queries were collected from common Bing searches with high click-through rates to code, and the CodeSearchNet authors manually filtered out queries that clearly included specific technical keywords.

2) Search Results: We generated the top 50 search results for each query and dataset using the state-of-the-art neural bag-of-words model packaged with CodeSearchNet. This was the best-performing model in the original CodeSearchNet publication, and remains one of the best-performing models on the CodeSearchNet benchmarks[2]. We chose to generate 50 results in line with the CodeSearchNet benchmarks.

3) Relevance Ratings: Each dataset includes relevance ratings for SCS results for each of the 99 queries. To create these ratings, the CodeSearchNet authors used a baseline code

[1]The KW baseline uses an adapted procedure to determine candidate and rejected functions. When keywords are rejected, any functions associated with those keywords are considered to be rejected. The set of candidates comprises functions associated with any selected keywords, excluding any rejects. Rejects are accounted for when suggesting subsequent keywords.

[2]https://wandb.ai/github/codesearchnet/benchmark/leaderboard

Query:	convert integer to text
Language:	Java
Method:	ZaCQ

Initial Relevance-Annotated Results:

RANK	SCORE	EXTRACTED TASK PHRASES
3	1	[“convert text to integer”]
10	3	[“convert int to string value”, “display text to screen”]
24	4	[“convert int to string value”]

Reciprocal Rank: .10 **Average Precision:** .09
NDCG: .77

Clarification Aspect:	POM | V, DO, P, PO
Clarifying Question:	What kind of value are you interested in converting int to?
Options:	float, datetime, string, null
Selected Response:	string

Reranked Relevance-Annotated Results:

RANK	SCORE	EXTRACTED TASK PHRASES
1	3	[“convert int to string value”, “display text to screen”]
3	4	[“convert int to string value”]
41	1	[“convert text to integer”]

New Reciprocal Rank: 1 **New Average Precision:** .83
New NDCG: .94

Fig. 2. Synthetic query refinement demonstration.

search method to retrieve the top-10 results for each query for each dataset; they then hired programmers to rate the relevance of individual results to the corresponding query on a scale from 1 (least relevant) to 4 (most relevant). Not all results received relevance ratings, and some results with ratings do not appear in our dataset because we used a different retrieval method.

4) Filtering: Certain queries did not have an adequate amount of relevance data for use in our synthetic evaluation. Therefore, we filtered queries that didn’t meet all three of the following criteria from the dataset:

- At least three search results have ratings
- At least one search result has a positive rating (3 or 4) and contains both a task phrase (used by ZaCQ and V-DO) and a keyword (used by KW)
- Rated search results are not already in the optimal order (i.e., ratings do not decrease monotonically with ranking)

Table II presents the final composition of the dataset.

D. Methodology

Our methodology for answering the RQs involved using each query refinement method (ZaCQ, V-DO, and KW) to iteratively rerank the search results of each query in our dataset. The synthetic refinement procedures were as follows:

1) Given a query and a list of search results, the refinement method generated a set of up to 5 refinement options. For ZaCQ and V-DO, these options were facet terms for a single query aspect (e.g., a set of verbs, or a set of direct objects that a certain verb acts upon). For KW, these were keywords.

2) We simulated a user’s response by selecting a relevant option. An option was considered “relevant” if it and any inferred attributes were associated with a result that was relevant to the query (i.e., rated 3 or 4). If there were multiple relevant options, the one associated with the fewest results was chosen. An option was also available to indicate that no relevant option was presented.

3) The simulated response was used to update the query representation and rerank the results.

4) The refinement method generated a new set of options, and repeated the process until no further refinement could occur (i.e., until ZaCQ had clarified a complete task, V-DO had clarified a V-DO pair, or KW had narrowed down a set of functions with identical keywords).

Figure 2 illustrates the synthetic evaluation’s query refinement process with the ZaCQ method. Initially, the most relevant search result is poorly-ranked. ZaCQ infers from the search results that the user’s information need involves converting an int to some kind of value, and chooses to clarify what kind of value the user is interested in. “String” is automatically selected in the evaluation because it is associated with a relevant result. ZaCQ uses this information to rerank all search results. The complete task phrase “convert int to string value” has been clarified, so the evaluation concludes.

E. Metrics

We evaluated refinement performance using three common metrics in code retrieval and refinement literature: 1) Mean Reciprocal Rank (MRR), 2) Mean Average Precision (MAP), and 3) Normalized Discounted Cumulative Gain (NDCG).

MRR is a metric for evaluating processes that produce lists of possible responses to a query. It considers only the rank of the first relevant (i.e., rated 3 or 4) result in a list, measuring the average reciprocal rank (the multiplicative inverse of the rank of the first relevant result) across a set of queries.

MAP is similar, but it looks at the ranks of all relevant results in the list. In other words, it rewards a system for ranking multiple relevant items highly.

NDCG considers the relevance ratings and ranks of all rated results, such that highly ranked relevant results are rewarded, while highly ranked irrelevant results are penalized (and vice versa). Typically NDCG is calculated for all results in a list of results. However, to account for the sparse relevance data in the CodeSearchNet dataset, we calculate NDCG over the subset of results with relevance ratings.

F. Results

Table III summarizes the results of the synthetic evaluation. All three refinement methods (V-DO, KW, and ZaCQ) were evaluated using the same set of queries and search results from the CodeSearchNet dataset, and their performance was measured in terms of MRR, MAP, and NDCG. The metrics were calculated for the results of the initial query, and then re-calculated after each round of refinement. We used grid search

TABLE III
RESULTS OF THE SYNTHETIC EVALUATION. THE BEST RESULTS FOR EACH ROUND ARE BOLDED. RESULTS PRODUCED BY ZACQ HAVE ASTERISKS FOR EACH BASELINE METHOD THEY SIGNIFICANTLY OUTPERFORM.

Metric	Refinement Method	# Rounds Refinement					
		0	1	2	3	4	>=5
NDCG	Keyword	.800	.876	**.915**	.917	.917	**.917**
	V-DO	.800	**.879**	.914	**.918**	**.918**	**.917**
	ZaCQ	.800	**.879**	.892	.907	.906	.913
MRR	Keyword	.590	.684	.872	**.971**	**.971**	**.971**
	V-DO	.590	.701	**.898**	.938	.943	.943
	ZaCQ	.590	**.765****	.843	.925	.943	.967
MAP	Keyword	.437	.494	**.634**	**.692**	**.692**	.692
	V-DO	.437	.503	.631	.666	.669	.670
	ZaCQ	.437	**.540****	.593	.659	.673	**.700***

to explore different hyperparameter configurations for each method; Table III presents the results of the best-performing configurations after each round of refinement.

1) RQ_1 (Reranking Effectiveness): After asking a single CQ, ZaCQ improved the MRR, MAP, and NDCG of the search results by 24%, 30%, and 10%, respectively. These improvements are statistically significant according to a two-sided Wilcoxon signed-rank test (p-value $< .05$). Asking additional CQs allowed for further improvement, though the average improvements were attenuated. The greatest total improvements (64% MRR, 60% MAP, and 14% NDCG) were achieved after fully clarifying all queries.

2) RQ_2 (Comparison to Baselines): Like ZaCQ, the V-DO and KW baselines improved upon the initial ordering of the search results. The greatest single-turn improvements were seen in the first round of refinement, with subsequent rounds yielding attenuated improvements. After one round of refinement, ZaCQ significantly outperformed both baselines in terms of MRR and MAP, and tied V-DO in terms of NDCG. In subsequent rounds, both baseline methods significantly outperformed ZaCQ in terms of all metrics. After about 4 rounds, ZaCQ started to catch up with the baselines, and when all three methods are allowed to run to completion, ZaCQ did not perform significantly worse than either of the baselines.

We attribute this pattern to ZaCQ's ability to infer task attributes. Consider the query "buffered file reader read text" and the initial refinement options presented by each method:

Method	Target	Options
V-DO	V	["read", "parse", "return", "set", "open"]
KW	Keyword	["character", "input", "line", "occur", "stream"]
ZaCQ	V, DO, P, PO	["read text from file"]

ZaCQ correctly inferred all four task attributes, allowing the refinement to conclude after just one round. For the other methods, additional refinement was necessary. ZaCQ inferred attributes most frequently in the first round of refinement; when successful, these inferences allowed ZaCQ to quickly narrow down the set of candidate results, accounting for improved first-round performance. In cases where ZaCQ's inferences were incorrect, the system had to spend additional rounds establishing the correct attributes. The V-DO and KW baselines lagged behind the first-round improvements of ZaCQ, but achieved their greatest improvements sooner by avoiding wasted rounds of inference.

The total improvement different methods achieved reflected the degree to which they were able to narrow the list of candidate results. ZaCQ was designed to clarify a complete task phrase, comprising 6 syntactic roles. In the sets of 50 search results used for the evaluation, it was rare for multiple results to share identical task phrases, meaning that ZaCQ was often able to identify a single relevant function. The V-DO baseline only clarified two semantic roles, allowing for more irrelevant results to wind up in the final list of candidates. By contrast, the KW baseline was able to narrow its candidate set down to a single relevant function for most queries, allowing it to achieve the highest final MRR.

G. Threats to Validity

As in any study, this evaluation carries a number of threats to validity; we have taken steps to acknowledge and address these threats where appropriate. The queries, search results, and relevance scores used present threats to internal and external validity. We mitigated these threats by choosing a large, standard dataset for code search tasks with fairly reliable relevance annotations (Cohen's κ = 0.47). Other threats include the selection of evaluation parameters. We limited the number of refinement options to 5, emulating Zamani *et al.* [24], and the number of keywords to 25, which was the maximum number investigated by Poshyvanyk and Marcus [42]. Selecting different parameters may have affected the observed results. Reporting the results of only the best-performing hyperparameter configurations after each round of refinement may be seen as a threat to construct validity, as that information is only available post hoc. We note that it is valuable to be able to specifically target the highest expected performance after n rounds, particularly for the first round. Further research is necessary to dynamically optimize hyperparameters throughout multiple rounds of refinement.

V. HUMAN STUDIES

We performed two human studies: a study to evaluate the *intrinsic* quality of the clarifying questions generated using the ZaCQ system, and a study of the *extrinsic* utility of CQs to real programmers during code search tasks. In the intrinsic study, we hired 10 annotators to rate the intrinsic quality of CQs generated by ZaCQ. In the extrinsic study, we hired 12 programmers to complete code search tasks, assisted by either ZaCQ or the Keyword recommendation baseline.

A. Research Questions

1) Intrinsic Study: For the intrinsic human study, we asked the following research questions:

RQ_3 Does ZaCQ generate meaningful questions?
RQ_4 Does ZaCQ generate natural questions?
RQ_5 Does ZaCQ generate grammatical questions?
RQ_6 Does ZaCQ generate logical questions?
RQ_7 Do users prefer clarifying questions generated by ZaCQ over keyword recommendations?

The purpose of RQ_3-RQ_6 is to evaluate the intrinsic properties that make for a "good" CQ. These quality metrics are based on an analysis of CQs by Stoyanchev *et al.* [20]. "Meaningful" means that a CQ seeks to make a meaningful distinction; "Natural" means that a CQ sounds like something a human might ask; "Grammatical" means that a CQ is grammatically-correct; and "Logical" means that a CQ logically targets a missing piece of information.

The purpose of RQ_7 is to compare users' first impressions of CQs generated by ZaCQ to keyword recommendations.

2) *Extrinsic Study:* For the extrinsic study, we asked the following research questions:

- RQ_8 Do users engage more with clarifying questions generated by ZaCQ than keyword recommendations?
- RQ_9 Does ZaCQ help users find relevant results faster than the Keyword baseline?
- RQ_{10} Does ZaCQ help users feel more confident in their search results than the Keyword baseline?

The purpose of RQ_8 is to examine whether users are more likely to engage with (i.e., click on) CQ-based or keyword-based refinement options. Although the keyword-based method can theoretically present a more discriminative set of refinement options, we hypothesize that a user-friendly natural-language interface will encourage more engagement

The purpose of RQ_9 and RQ_{10} is to investigate the utility of CQs generated by ZaCQ during code search. We examine search duration and user confidence, as CQs have been shown to improve both metrics in general-domain search [44].

B. Intrinsic Study Methodology

Our methodology to answer RQ_3-RQ_7 involved creating a survey asking programmers to rate the CQs generated by ZaCQ. As in the synthetic evaluation (see Section IV-C), we used queries and search results from CodeSearchNet. We generated CQs for all 99 CodeSearchNet queries, using search results from the Java dataset. For each query, we generated a single question with ZaCQ and KW; that is, we did not examine subsequent questions after an initial round of refinement. We chose to only look at the first question because our synthetic evaluation found that the first question yielded the greatest improvement in result reranking.

We hired 10 participants on the crowdsourcing website *Prolific.co* to complete the survey. Participants were pre-screened for English fluency. Because this survey did not require participants to read or understand code, it fell in line with literature tasking non-programmers with reading software documentation [45]. Nevertheless, we limited the participant pool to participants reporting experience with "Computer Programming." We also excluded participants who failed attention check questions throughout the survey.

Survey participants were first shown task instructions and examples of appropriate ratings. After completing a training exercise, they were brought to the main rating interface, which consisted of four elements: first, a text box displaying a programmer query and the corresponding CQ; second, a box containing four input fields asking the user to rate the CQ on a 1-5 Likert scale in terms the four quality criteria; third, a text box introducing the refinement prompt and options generated by the KW baseline; and finally, an input field asking the participant to indicate which refinement method they preferred.

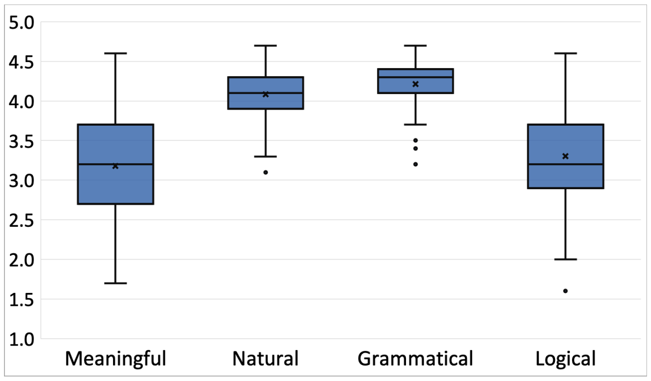


Fig. 3. Average quality ratings for ZaCQ clarifying questions in the intrinsic human study.

C. Intrinsic Study Results

We collected 10 user surveys rating 99 CQs generated by ZaCQ. We analyzed interrater reliability using intraclass correlation (ICC(3,k)). Reliability was fair to good for all quality attributes; the "Logical" and "Meaningful" qualities saw the highest levels of agreement (r=.74, 95% CI [.65, .81] and r=.76, 95% CI [.68, .82], respectively), while the "Natural" and "Grammatical" qualities had lower agreement (r=.41, 95% CI [.22, .57] and r=.51, 95% CI [.35, .64]).

Figure 3 summarizes the quality ratings that participants provided, addressing RQ_3-RQ_6. For most questions, the "Natural" and "Correct" qualities were rated highly; this consistency is not surprising, given that the questions were generated using a limited set of hand-crafted templates. The "Meaningful" and "Logical" qualities saw more variation from question to question. Table IV shows the CQs that received the highest- and lowest- average ratings for each quality metric. In general, confirmation questions (i.e., questions confirming inferred task attributes) were highly rated across all 4 metrics, while questions in which the system failed to extract task attributes from the user's query and those where semantic roles in the options appeared inconsistent received lower ratings.

To answer RQ_7, we looked at how frequently participants preferred CQs to KW recommendations. Overall, 63% of all ratings indicated a preference for the CQ, and a majority of participants preferred the CQ for 63% of queries. Confirmation questions were the most preferred (70%), while elicitation questions targeting verbs were the least preferred (54%).

D. Extrinsic Study Methodology

To answer RQ_8-RQ_{10}, we ran an experiment in which programmers completed a series of code search tasks aided by either the KW or the ZaCQ refinement method. We recruited 12 programmers to participate in the experiment. All participants were Java programmers with 4 to 17 years of general programming experience (average 8.8 ± 4.2).

TABLE IV

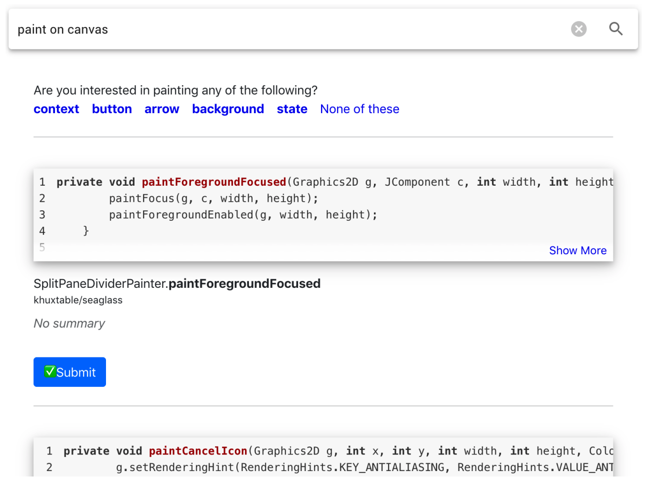
THE HIGHEST- AND LOWEST-RATED CLARIFYING QUESTIONS IN THE INTRINSIC EVALUATION.

Metric	Highest-Rated CQ	Lowest-Rated CQ
Meaningful	**Query**:“convert string to number” **CQ**: Found 1 function that specifically mentions converting string to number. Would you like to see it first? (Avg. score = 4.6)	**Query**:“find int in string” **CQ**:Are you interested in finding in any of the following: half of the node, range, range start, right, or text? (Avg. score = 1.7)
Natural	**Query**:“convert string to number” **CQ**: Found 1 function that specifically mentions converting string to number. Would you like to see it first? (Avg. score = 4.7)	**Query**:“encode url” **CQ**: Are you looking for any of the following: characters, method, string, unsupportedencodingexception, or url? (Avg. score = 3.1)
Grammatical	**Query**:“unzipping large files” **CQ**: Are you interested in doing any of the following: copying files, extracting files, handling files, overwriting files, or reading files? (Avg. score = 4.6)	**Query**:“encode url” **CQ**: Are you looking for any of the following: characters, method, string, unsupportedencodingexception, or url? (Avg. score = 3.1)
Logical	**Query**:“priority queue” **CQ**: Are you interested in doing any of the following: changing priority, getting priority, removing priority, returning priority, or setting priority? (Avg. score = 4.6)	**Query**:“find int in string” **CQ**: Are you interested in finding in any of the following: half of the node, range, range start, right, or text? (Avg. score = 1.6)

We designed a custom search interface (Figure 4) for the CodeSearchNet Java dataset (see Section IV-C) using neural bag-of-words search method. When users submit queries, the system retrieves the top 50 search results (displayed across 5 pages of 10 results each). The system also generates a query refinement prompt and options to display to the user. User interactions with the interface (e.g., queries, changing page, selecting refinement options) are logged in a database.

We gave programmers 8 code search tasks created by Martie *et al.* [7] for evaluating query refinement techniques for code search. Each task consists of a scenario, like “You are building a sketching application” and a request, like “Find 4 snippets of Java source code that you think will help.” We instructed programmers to use the interface to search as though they were actually programming a solution. When they found a function that they would want to use/replicate, we instructed them to press a “Submit” button. The participant would then provide confidence ratings from 1-5 indicating 1) how confident they felt that this function would *correctly* address their needs, and 2) how confident they felt that they had found the *best* available function for the task.

Half of the search tasks requested a single function, while the others request four. Additionally, half of the tasks requested the user find functions implementing relevant algorithms/data structure, while the rest requested any “helpful” functions. The combinations of response number/type pairs comprised four task categories of two tasks each.


paint on canvas

Are you interested in painting any of the following?

context button arrow background state None of these

```
private void paintForegroundFocused(Graphics2D g, JComponent c, int width, int height
        paintFocus(g, c, width, height);
        paintForegroundEnabled(g, width, height);
    }

```

SplitPaneDividerPainter.**paintForegroundFocused**

khuxtable/seaglass

No summary

```
private void paintCancelIcon(Graphics2D g, int x, int y, int width, int height, Colo
        g.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANT
```

Fig. 4. User interface for the extrinsic study. Clarifying questions and potential answers appear below the search bar.

Like Martie *et al.* [7], we used a mixed experimental design in which each participant completed four tasks using KW and four with ZaCQ. Specifically, each participant completed one task in each task category using one refinement method, and the other task in that category with the other method. We assigned refinement methods to tasks such that each task-refinement method pair was assigned to a unique set of 6 participants. As participants completed tasks, they alternated between the KW treatment and the ZaCQ treatment. To address ordering effects, half of the participants started with ZaCQ and the other half started with KW. The order of tasks within each treatment were randomized.

E. Extrinsic Study Results

To answer RQ_8, we analyze how frequently individual participants engaged with ZaCQ and KW. To answer RQ_8 and RQ_9, we consider the search duration and confidence measures across all tasks and participants. Figures 5 and 6 summarize these results.

1) RQ_8 (Engagement): The majority of participants engaged with more CQs than keyword recommendations. Figure 6 shows the total number of times each user selected a refinement option from the ZaCQ or KW interface across the 8 code search tasks. Overall, 75% of the participants selected more responses to CQs than keywords. Participants selected an average of 7.2 CQ answers and 5.2 keywords over the course of the study. Furthermore, the average engagement rate per search query was .366 for ZaCQ and .191 for KW.

We observe that 4 of the 12 participants only refined a single search query using either method. One of these participants reported in the exit survey that they “often didn’t even read” the refinement suggestions throughout the study, noting that rewriting queries manually was “just how I do things.” Of the 8 participants who refined multiple queries, 7 of them engaged with ZaCQ more than KW.

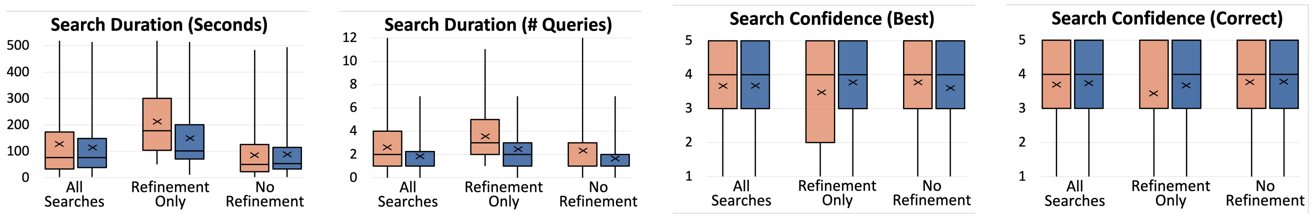


Fig. 5. 🟧=KW, 🟦=ZaCQ. Search duration and confidence ratings in the extrinsic human study.

2) RQ_9 (Duration): We measured two metrics to evaluate RQ_9: the amount of time spent and the number of search queries participants used to find each submitted function (not counting time spent providing ratings/explanations). The duration of search sessions varied greatly, the quickest taking only 3 seconds and the longest 3 exceeding the 10 minute time limit. Overall, we do not observe a significant difference between the ZaCQ treatment and the KW treatment in terms of the average time and number of queries ($p > .05$ for a two-sample, two-tailed t-test).

We note that in two-thirds of all searches, participants did not select any refinement options. To gauge the actual refinement effectiveness, we analyze the subset of search sessions in which participants selected at least one refinement option. Of the 237 total search sessions, 35 were refined by keywords and 49 were refined by CQs. We make two observations: first, the time and number of queries used for searches involving refinement were significantly higher than for searches with no refinement ($p < .01$). Second, the time and number of queries used for searches refined by CQs were significantly lower than for those refined by keywords ($p < .05$).

These observations suggest that participants looked to the refinement interface for assistance during difficult searches. In that context, the shorter duration of searches refined by CQs suggests that CQs helped resolve difficult searches more efficiently than keyword recommendations.

3) RQ_{10} (Confidence): We had participants rate two measures of search confidence: confidence that an answer was correct, and confidence that an answer was the best available. As with search duration, we do not observe a significant difference in either confidence metric between the two treatments when averaging over all search sessions. The subset of searches that involved refinement have lower confidence scores than for the subset with no refinement; however, we do not observe a significant difference in search confidence between the two treatments for the refined subset ($p > .05$).

F. Threats to Validity

Each human study carries threats to validity. For the intrinsic study, the selection of evaluation queries and search results present a threat to external validity; in particular, search results for other programming languages may have produced higher- or lower-quality questions. The selection of participants is another core threat; we aimed to reduce the threat by recruiting from a reliable crowdsourcing service [46] and including attention checks to filter low-quality participants.

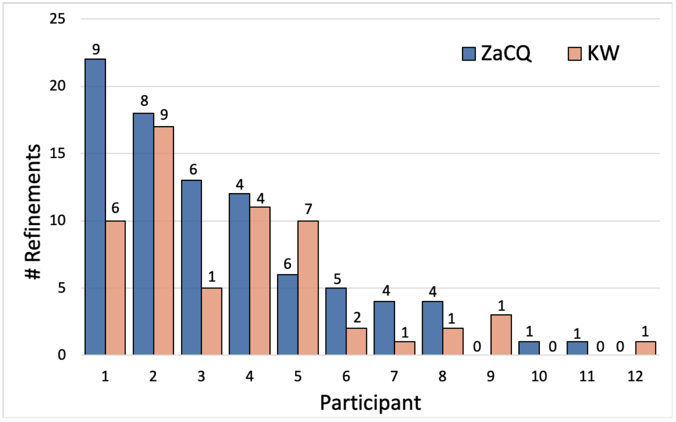


Fig. 6. Engagement in the extrinsic human study. The number above each bar indicates the number of searches in which refinement took place.

For the external study, the selection of search tasks is a threat that may reduce generalizability. Individual differences among programmers play a role in their search behavior and preferences. We attempted to distribute the bias of individual differences by assigning a different set of programmers to complete each task with each treatment. Nevertheless, differences between programmers and tasks may bias the result averages, particularly for the subset of searches that had been refined using the experimental methods.

VI. Conclusion

We have presented an approach to refine source code search queries using natural language clarifying questions. Prior CQ generation methods rely on data that is not readily available for SCS. Our approach uses a task extraction algorithm to identify query aspects, and then follows a rule-based procedure for question generation. We use a feedback relevance algorithm to elevate relevant search results, including those for which descriptive task phrases are not extracted. We performed a synthetic study and two human studies to evaluate our method. It generally creates useful and natural-sounding clarifying questions; however, the inflexible rules can sometimes lead to stilted-sounding questions or repetitive options for refinement.

Overall, we believe that CQs will play a significant role in intelligent tools for developer support. Future work should aim to incorporate more sophisticated models for result salience in the aspect inference/selection process and experiment with different result reranking algorithms.

For reproducibility, we make all source code and experimental material available online:
https://anonymous.4open.science/r/ZaCQ-A3BF/

References

[1] S. P. Reiss, "Semantics-based code search," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 243–253.

[2] S. Yan, H. Yu, Y. Chen, B. Shen, and L. Jiang, "Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 344–354.

[3] 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.

[4] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program understanding and the concept assignment problem," *Communications of the ACM*, vol. 37, no. 5, pp. 72–82, 1994.

[5] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais, "The vocabulary problem in human-system communication," *Communications of the ACM*, vol. 30, no. 11, pp. 964–971, 1987.

[6] 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, pp. 111–120.

[7] L. Martie, A. v. d. Hoek, and T. Kwak, "Understanding the impact of support for iteration on code search," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 774–785.

[8] L. Tavakoli, H. Zamani, F. Scholer, W. B. Croft, and M. Sanderson, "Analyzing clarification in asynchronous information-seeking conversations," *Journal of the Association for Information Science and Technology*, 2021.

[9] M. P. Kato, R. W. White, J. Teevan, and S. T. Dumais, "Clarifications and question specificity in synchronous social q&a," in *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, 2013, pp. 913–918.

[10] A. Wood, P. Rodeghero, A. Armaly, and C. McMillan, "Detecting speech act types in developer question/answer conversations during bug repair," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 491–502.

[11] M. Aliannejadi, H. Zamani, F. Crestani, and W. B. Croft, "Asking clarifying questions in open-domain information-seeking conversations," in *Proceedings of the 42nd international acm sigir conference on research and development in information retrieval*, 2019, pp. 475–484.

[12] E. Shihab, S. Wagner, and M. Aurélio Gerosa, "Summary of the 2nd international workshop on bots in software engineering (botse 2020)," *ACM SIGSOFT Software Engineering Notes*, vol. 46, no. 1, pp. 20–22, 2021.

[13] Z. Eberhart and C. McMillan, "Dialogue management for interactive api search," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021.

[14] N. Zhang, Q. Huang, X. Xia, Y. Zou, D. Lo, and Z. Xing, "Chatbot4qr: Interactive query refinement for technical question retrieval," *IEEE Transactions on Software Engineering*, 2020.

[15] T. Xie, "Intelligent software engineering: Synergy between ai and software engineering," in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer, 2018, pp. 3–7.

[16] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez *et al.*, "On-demand developer documentation," in *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 479–483.

[17] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 232–242.

[18] S. Wang, D. Lo, and L. Jiang, "Active code search: incorporating user feedback to improve code search relevance," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 677–682.

[19] C. Treude, M. P. Robillard, and B. Dagenais, "Extracting development tasks to navigate software documentation," *IEEE Transactions on Software Engineering*, vol. 41, no. 6, pp. 565–581, 2014.

[20] S. Stoyanchev, A. Liu, and J. Hirschberg, "Towards natural clarification questions in dialogue systems," in *AISB symposium on questions, discourse and dialogue*, vol. 20, 2014.

[21] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[22] F. Radlinski and N. Craswell, "A theoretical framework for conversational search," in *Proceedings of the 2017 conference on conference human information interaction and retrieval*, 2017, pp. 117–126.

[23] J. Zou, E. Kanoulas, and Y. Liu, "An empirical study on clarifying question-based systems," in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 2020, pp. 2361–2364.

[24] H. Zamani, S. Dumais, N. Craswell, P. Bennett, and G. Lueck, "Generating clarifying questions for information retrieval," in *Proceedings of The Web Conference 2020*, 2020, pp. 418–428.

[25] I. Sekulić, M. Aliannejadi, and F. Crestani, "Towards facet-driven generation of clarifying questions for conversational search," in *Proceedings of the 2021 ACM SIGIR International Conference on Theory of Information Retrieval*, 2021, pp. 167–175.

[26] M. Aliannejadi, J. Kiseleva, A. Chuklin, J. Dalton, and M. Burtsev, "Building and evaluating open-domain dialogue corpora with clarifying questions," *arXiv preprint arXiv:2109.05794*, 2021.

[27] A. M. Krasakis, M. Aliannejadi, N. Voskarides, and E. Kanoulas, "Analysing the effect of clarifying questions on document ranking in conversational search," in *Proceedings of the 2020 ACM SIGIR on International Conference on Theory of Information Retrieval*, 2020, pp. 129–132.

[28] S. Gottipati, D. Lo, and J. Jiang, "Finding relevant answers in software forums," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 323–332.

[29] E. Knauss, D. Damian, G. Poo-Caamaño, and J. Cleland-Huang, "Detecting and classifying patterns of requirements clarifications," in *2012 20th IEEE International Requirements Engineering Conference (RE)*. IEEE, 2012, pp. 251–260.

[30] Z. Gao, X. Xia, D. Lo, and J. Grundy, "Technical q8a site answer recommendation via question boosting," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 1, pp. 1–34, 2020.

[31] W. Kong and J. Allan, "Extracting query facets from search results," in *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, 2013, pp. 93–102.

[32] M. Gu, A. Aamodt, and X. Tong, "Component retrieval using conversational case-based reasoning," in *International Conference on Intelligent Information Processing*. Springer, 2004, pp. 259–271.

[33] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *Proceedings of the 6th international conference on Aspect-oriented software development*, 2007, pp. 212–224.

[34] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in ir-based concept location," in *2009 IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 351–360.

[35] D. S. Eisenberg, J. Stylos, and B. A. Myers, "Apatite: A new interface for exploring apis," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010, pp. 1331–1334.

[36] E. Hill, L. Pollock, and K. Vijay-Shanker, "Improving source code search with natural language phrasal representations of method signatures," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 524–527.

[37] M. Roldan-Vega, G. Mallet, E. Hill, and J. A. Fails, "Conquer: A tool for nl-based query refinement and contextualizing code search results," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 512–515.

[38] J. Li, A. Sun, Z. Xing, and L. Han, "Api caveat explorer–surfacing negative usages from practice: An api-oriented interactive exploratory search system for programmers," in *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, 2018, pp. 1293–1296.

[39] A. Sivaraman, T. Zhang, G. Van den Broeck, and M. Kim, "Active inductive logic programming for code search," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 292–303.

[40] W. Xie, X. Peng, M. Liu, C. Treude, Z. Xing, X. Zhang, and W. Zhao, "Api method recommendation via explicit matching of functionality verb phrases," in *Proceedings of the 28th ACM Joint Meeting on European

Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 1015–1026.

[41] J. Rocchio, “Relevance feedback in information retrieval,” *The Smart retrieval system-experiments in automatic document processing*, pp. 313–323, 1971.

[42] D. Poshyvanyk and A. Marcus, “Combining formal concept analysis with information retrieval for concept location in source code,” in *15th IEEE International Conference on Program Comprehension (ICPC’07)*. IEEE, 2007, pp. 37–48.

[43] M. Honnibal. (2021) spaCy. Version 3.0.0. [Online]. Available: https://spacy.io

[44] H. Zamani, B. Mitra, E. Chen, G. Lueck, F. Diaz, P. N. Bennett, N. Craswell, and S. T. Dumais, “Analyzing and learning from user interactions for search clarification,” in *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020, pp. 1181–1190.

[45] Z. Eberhart, A. LeClair, and C. McMillan, “Automatically extracting subroutine summary descriptions from unstructured comments,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 35–46.

[46] E. Peer, L. Brandimarte, S. Samat, and A. Acquisti, “Beyond the turk: Alternative platforms for crowdsourcing behavioral research,” *Journal of Experimental Social Psychology*, vol. 70, pp. 153–163, 2017.