Answering Uncertain, Under-Specified API Queries Assisted by Knowledge-Aware Human-AI Dialogue

Qing Huang, Zishuai Li, Zhenchang Xing, Zhengkang Zuo, Xin Peng, Xiwei Xu, Qinghua Lu

Abstract—Developers' API needs should be more pragmatic, such as seeking suggestive, explainable, and extensible APIs rather than the so-called best result. Existing API search research cannot meet these pragmatic needs because they are solely concerned with query-API relevance. This necessitates a focus on enhancing the entire query process, from query definition to query refinement through intent clarification to query results promoting divergent thinking about results. This paper designs a novel Knowledge-Aware Human-AI Dialog agent (KAHAID) which guides the developer to clarify the uncertain, under-specified query through multi-round question answering and recommends APIs for the clarified query with relevance explanation and extended suggestions (e.g., alternative, collaborating or opposite-function APIs). We systematically evaluate KAHAID. In terms of human-AI dialogue efficiency, it achieves a high diversity of question options and the ability to guide developers to find APIs using fewer dialogue rounds. For API recommendation, KAHAID achieves an MRR and MAP of 0.769 and 0.794, outperforming state-of-the-art methods BIKER and CLEAR by at least 47% in MRR and 226.7% in MAP. For knowledge extension, KAHAID obtains an MRR and MAP of 0.815 and 0.864, surpassing ZaCQ by at least 42% in MRR and 45.2% in MAP. Furthermore, we conduct a user study. It shows that explainable API recommendations, as implemented by KAHAID, can help developers identify the best API approach more easily or confidently, improving inspiration of clarification question options by at least 20.83% and the extensibility of extended APIs by at least 12.5%.

Index Terms—Developers' API Need, Knowledge Graph, Human-AI Dialogue, API Recommendation, Multi-Round Question Answering.

INTRODUCTION

EVELOPERS' API (short for Application Programming Interface) needs are no longer just a matter of finding so-called best API for certain programming tasks. To minimize API misuse, they must consider several aspects such as the API's specific usage context, relations to cooperative APIs, and confusing APIs with subtle differences. [1]. As a result, it is desirable that API search methods should guide developers to clarify the vague question intent, provide diverse and suggestive APIs for different needs, interpret the search results, and extend other potentially useful API knowledge [2]. This expectation reveals some pragmatic API needs, which seek suggestive, explainable, and extensible API recommendation and knowledge discovery rather than simply presenting so-called best APIs. Meeting these pragmatic API needs not only helps developers select the ideal APIs for their needs, but it also inspires and broadens their thinking, such as exploring alternative or better solutions, discovering previously unknown API knowledge.

Existing API search methods, however, are incapable of meeting the above-mentioned pragmatic API needs. Given an API query, they measure query-API relevance based on keyword matching [3], [4], [5] or embedding similarity [6], [7], and return the top-k related APIs as a list of "best"

• X. Peng is with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University.

answers. These answers are only useful if the API query can clearly describe the actual query intent, but even then, they fall far short of the pragmatic API needs because the answers are viewed as independent of each other with little or no explanation. Furthermore, an API query in the real world is highly likely uncertain and under-specified. In order to clarify the query intent, the query can be extended based on word co-occurrence [8]. However, without any human intervention, the extended words are very likely irrelevant and may potentially introduce noise that blur the query intent [9], [10]. It follows that API search should shift from its current narrow focus on query-API relevance to enhancing the entire search process, from query definition to query refinement through intent clarification, to query results promoting divergent thinking about the results.

This query process is vividly reflected in the socialtechnical information seeking on online forums. For example, on Stack Overflow, a question-answering (Q&A) thread usually includes a number of comments that help clarify uncertain and under-specified API questions and discuss technical details and trade-offs [11], [12], in which the developer often gains more than just a single API, but becomes familiar with various APIs' specific usage contexts and their cooperation and differences [13]. Although socialtechnical information seeking aligns well with the pragmatic API needs and supports much better question clarification, intent understanding and answer explanation through developer interactions and knowledge sharing, the questionanswering process relies on human interaction and engagement, which cannot immediately respond to the developers'

1

Q. Huang, Z. Li, Z. Zuo are with School of Computer Information Engineering, Jiangxi Normal University, China.

[•] Q. Huang and Z. Li are co-first authors, Z. Zuo is the corresponding author(zuo803@jxnu.edu.cn).

[•] Z. Xing, X. Xu and Q. Lu are with the CSIRO's Data61, Australia.

needs.

In this work, we design a novel Knowledge-Aware Human-AI dialogue agent (KAHAID) as the first step towards integrating the immediate response capability of API research and the interaction, clarification, explanation, and extensibility capability of social-technical information seeking. Given an uncertain, under-specified query, KAHAID interacts with the developer to clarify the query intent through multi-round question answering and returns APIs with relevance explanation and extended knowledge for the clarified query.

Each dialogue round clarifies a certain aspect of the query by asking a question with a list of options. To automatically generate meaningful clarification questions with diverse options, we construct an API behavior knowledge graph (KG) that extracts API actions, objects, constraints and diverse functional and semantic relations from API documentation. To support efficient dialogue process, we design an information-gain based decision tree algorithm over the underlying knowledge graph to prioritize the questions posted to the developer and minimize dialogue rounds. Finally, KAHAID identifies APIs in the knowledge graph based on the developer's answers to its clarification questions, and present not only the most relevant API but also extended suggestions (e.g., alternative, cooperating or opposite-function APIs). All recommended APIs come with the explanation of how they are related to the query and other APIs. This result explanation can enhance the developer's trust in the recommended APIs and allow them to make informed choice among APIs.

We have implemented KAHAID for Java SDK APIs.

To evaluate the effectiveness of human-AI dialogue and the quality of API recommendations by KAHAID, we have developed three test sets for testing. The first is the dataset we reused from BIKER's manually created test dataset, the second is randomly selected 1k sample SO posts to alleviate potential human bias. To simulate questions and answers in real situations, we added a third test dataset, which contains manually selected 60 questions with ground-truth APIs from SO. In terms of dialogue efficiency, KAHAID achieves high semantically diversity of question options (the average diversity between any two options is 74.9%) and the ability to guide developers using fewer dialogue rounds to find APIs (the average number of rounds required for a query to find an answer is no more than three). Our evaluation confirms that KAHAID has a strong ability for API recommendation and knowledge extension. For API recommendation, KAHAID achieves a mean reciprocal rank (MRR) and mean average precision (MAP) of 0.769 and 0.794 respectively, and this outperforms the two state-of-theart API search approaches BIKER and CLEAR by at least 47% in MRR and 226.7% in MAP. In terms of knowledge extension, KAHAID achieves an MRR and MAP of 0.815 and 0.864 respectively, which surpasses the state-of-the-art conversation-based code search method, ZaCQ, by at least 42% in MRR and 45.2% in MAP.

Furthermore, we conducted a user study in which 12 Java developers were divided into two groups using different tools to find APIs for 8 programming tasks (derived from the selected 60 SO questions) using KAHAID and ZaCQ, respectively. Using KAHAID improves the inspiration of clarification question options by at least 20.83% and the extensibility of extended APIs by at least 12.5%. These results also confirm that explainable API recommendation can assist developers in selecting the best API method more easily and confidently.

Overall, this paper makes the following contributions:

1) Conceptually, we are aware of pragmatic API needs, which include suggestive, explainable, and extensible API recommendation and knowledge discovery. We believe that API search should shift from only finding the best APIs to enhancing the entire query process by providing a variety of potentially useful and enlightening knowledge. Driven by this conception, we design KAHAID to achieve an interactive, illuminating, explainable, and extensible exploratory discovery.

2) We create an API behavior knowledge graph that includes API actions, objects, constraints, functional relations and semantic relations to generate clarification questions with multiple options. These questions can prompt developers to discover better options or help them determine their actual needs.

3) Over the underlying knowledge graph, we design an information-gain based decision tree algorithm to prioritize the questions posted to the developer, minimize question answering rounds, and guide the developer gradually clarify the vague question intent.

4) We evaluate KAHAID's ability to meet pragmatic API needs in terms of variety, guidance, extensibility, and interpretability. Our data package can be found here ¹.

2 MOTIVATING EXAMPLE

2.1 API Search Status

In order to find the APIs for developers' needs, a common solution is to use the natural language description of the API need as a query, and use API search approaches to obtain some candidate APIs whose documentation is similar to the query. During the search process, developers may seek diverse, explainable, guided, and extensible API recommendations, rather than just a single API.

Consider a Stack Overflow question where a developer needs a Java API to get the current working directory. Let us consider how to satisfy this API need using API search over API documentation (assume that the answers like this SO post do not exist). If the search query is specific, the API search will be able to find the most relevant API, but it likely loses extensibility and miss other potentially useful APIs. For example, assume the developer issues a specific query "get absolute path string of current working directory in Java", BIKER [8] (a query expansion API search tool) can find the most relevant API directly (i.e., *java.io.File.getAbsolutePath*) by matching the query with the API description. However, its results does not include java.nio.file.Path.toAbsolutePath, a Java new IO API which may also satisfy the need "get the current work directory". But the description "Returns a Path object representing the absolute path" of java.nio.file.Path.toAbsolutePath is not similar to the very specific query, so it will not be returned. As a result, the developer may miss the opportunity to use a new API based on the API search results.

 $1.\ https://github.com/Answering-uncertain-under-specified-API-query-assisted-by-knowledge-aware-human-AI-dialogue$

Subgraph

Human-Al Dialogue



Note: The red text and lines in the Subgraph indicate the critical path necessary for obtaining the best API.



On the other hand, if the search query is too broad, the API search tool very likely cannot find the relevant APIs. For example, using the SO question title "how to get the current working directory in Java" as a query, BIKER returns a chaotic set of APIs. In top 10, only one API java.io.File.getCanonicalPath (returns the canonical path string of the current directory) is relevant but this API is ranked seventh. The poor search results actually reveal another issue of API search, i.e., lack of interpretability, making it difficult for the developers to interpret the search results and determine their relevance and trustworthiness. For example, the top-1 ranking API is *java.io.File.list*, but it is difficult to understand how java.io.File.list could be used to get the current working directory, as it only lists all files in the current working directory. It is also difficult to understand why this irrelevant API is ranked first while the relevant API java.io.File.getCanonicalPath is ranked seventh.

2.2 API Knowledge Seeking on Stack Overflow

Different from API search, question-answering process on Stack Overflow offers a completely different experience. The Stack Overflow question "How to get the current working directory in Java²" received 70 comments, many of which help to clarify the vague question intent. For example, one comment asks "What is it you're trying to accomplish by accessing the working directory? Could it be done by using the class path instead?". This comment clarifies the action is "accomplish", the object of this action is "what", and whether or not the constraint on this event is "by using the class path". Another comment says "Knowledge of the current working directory is important for all relative paths. If you think that is irrelevant make sure you always access files via some absolute path." This comment clarifies whether the constraint on the object "path" is "all relative" or "some absolute".

The question receives diverse answers, including directly relevant APIs such as *java.io.File.getAbsolutePath* and *java.io.File.getCanonicalPath*, as well as extended APIs like *java.nio.File.toAbsolutePath* with similar functionality, and *java.nio.File.Paths.get* as a cooperative API that converts the path string from *java.io.File.getAbsolutePath* to a path object. Meanwhile, these answers and comments on Stack Overflow also provide explanation of the recommended APIs, drawn from the API documentation. As a result, these explanations reinforce the developer's understanding and trust of the recommended APIs in the answers. However, as the Q&A process relies on human inputs, those diverse answers were provided across 4 years.

2.3 API Search Assisted by Human-AI Dialogue

In this work, we aim to assist API search with human-AI dialogue which simulates the capability of intent clarification and result explanation and extension as the Q&A process on Stack Overflow, and meanwhile can provide the immediate response to the search query. The Q&A process on Stack Overflow is underpinned by human knowledge of API behaviors and relations. In the same vein, API

^{2.} https://stackoverflow.com/questions/4871051/

search with human-AI dialogue needs to be supported by a knowledge graph of API behaviors which represents API actions, objects, constraints and various API functional and semantic relations in a graph like the example shown in Fig. 1. Given a search query, the agent interacts with the developer to clarify the query intent until it finds some APIs. It presents the found APIs with the explanation how they are related to the clarified query and each other.

Fig. 1 illustrates an example of this knowledge-graph supported human-AI dialogue process for API search. The developer Jack initially asks an under-specified question "How to get the current working directory in Java", for which the current API search tool (e.g., BIKER [8]) returns poor results. However, the agent KAHAID, based on the API behavior knowledge graph, determines it needs some clarification of actions first, because several different actions are somewhat related to "get working directory". It asks "what do you want to do?" with a list of options extracted from the knowledge graph, such as "return path", "return filesystem", "return codesource" or "convert path string to path". Jack replies "return path". With the clarified action, KAHAID determines it needs some further clarification about the type of path to be returned, either "path object" or "path string", again based on the knowledge graph. Jack replies "path string", which triggers another round of clarification "which kind of path string", either absolute or canonical. Jack replies "absolute". Through three rounds of human-AI dialogue, the intent of the initial under-specified question becomes clear, which leads KAHAID to an API java.io.File.getAbsolutionPath.

KAHAID goes beyond *java.io.File.getAbsolutionPath* to identify other relevant APIs using semantic relations in the knowledge graph. Specifically, KAHAID locates two functionally similar APIs, namely *java.io.File.getCanonicalPath* and *java.nio.file.Path.toAbsolutePath*, as well as two functionally cooperative APIs, namely *java.nio.file.Paths.get* and *java.nio.file.FileSystem.getPath*. Instead of simply presenting some APIs, KAHAID provides a concise explanation for each recommended API. First, it shows the API's functionality description and highlights the keywords that it uses as clarification options. This helps Jack determine why these APIs are recommended and how they are relevant to his question. Second, KAHAID shows the relations of the extended APIs and the most relevant API, such as function similarity and function cooperation.

To sum up, API search assisted by knowledge-aware human-AI dialogue will create an exploratory search process that is suggestive, explainable, and extensible. It can lead to effective and serendipitous API recommendation and knowledge discovery.

3 APPROACH

Fig. 2 depicts the three stages of KAHAID design: API Behavior KG Construction, Human-AI Dialogue, and Explainable and Extensible API Recommendation.

In the first stage, we extract API entities (such as API actions, objects, events and constraints) and diverse functional relations from API documents using natural language processing (NLP) techniques, and then enrich the various semantic relations of API entities. Finally, the API behavior KG is constructed.

In the second stage, given an under-specified API query, KAHAID uses the subgraph search method to retrieve the Query-related API behavior subgraph containing various functional relations from KG. Based on the subgraph, KA-HAID builds a decision tree using the information-gain based decision tree algorithm, and initiates a dialogue based on this tree. If the user don't decide to continue the dialogue, KAHAID generates the result APIs directly and proceeds to the third stage; otherwise, KAHAID generates a clarification question with a variety of options to assist the user in clarifying the uncertain aspects of the query. After the user selects an option that represents a specific branch of the decision tree, KAHAID updates the decision tree based on the user's choice by merging and pruning the decision tree.

In the third stage, KAHAID returns the explainable result API and any extended APIs that have API semantic relations with the result API.

3.1 API Behavior KG Design and Construction

As stated in Section 2.3, API search with Human-AI dialog need to be supported by a knowledge graph of API behaviors which represents API actions, objects, constraints and various API functional and semantic relations in a graph. The following section describes how to design and build the API behavior knowledge graph.

3.1.1 API Knowledge Source

The API documentation contains a wealth of knowledge about API behavior, typically documented in API descriptions. These descriptions are frequently used to answer questions on Stack Overflow. For example, a question titled "How to get the path of a running JAR file?"³ is posted. The sixth respondent endorsed API *"java.nio.file.Paths.get()"* by referencing the functional event mentioned in the API description⁴, thereby supporting the recommendation.

In this work, we download the JDK 1.8 API reference documentation ⁵ and parse each API class's HTML file to extract API methods and their descriptions following the method mentioned in [14], resulting in API method-description pairs. Finally, we build a Java API dictionary with 30,200 API method-description pairs.

3.1.2 KG Design

Our KG design is driven by two types of knowledge: clarification question related knowledge and result extension related knowledge.

Clarification question related knowledge: our KG includes the API behavior knowledge necessary for clarifying the following three types of questions in the under-specified queries.

The first type of question is event-oriented and serves to clarify the user's goals. For example, given a question "What do you want to do?", we can derive the **Event** entity ("convert a path string to a path"), and further divide the Event entity into the **Action** entity ("convert") and the **Object** entity ("path string" or "path") because the event

^{3.} https://stackoverflow.com/questions/320542/

^{4.} https://docs.oracle.com/javase/8/docs/api/java/nio/file/Pathshtml#gat_iava_lang_String_iava_lang_String

[.]html#get-java.lang.String-java.lang.String...

^{5.} https://docs.oracle.com/javase/8/docs/api



Fig. 2: Overall Framework of KAHAID

is composed of the action and the object. As a result, the <u>Act Has Event</u> relation naturally arises between the entities <u>Action and Event</u>, while the <u>Has Direct Object</u> and <u>Has</u> <u>Preposition Object</u> relations naturally arises between the entities Event and the Object. For example, "path string" and "path" are the direct and prepositional objects of "convert".

The second type of question is object constraint-oriented and serves to clarify the type or statue of object. For example, given a question "What type of the path string do you convert?", we can derive the Object Constraint entity, despite the fact that there is no modifier for "path string" in this real API description. The Object Constraint relation will naturally arise between the Object and the Object Constraint entities. However, this relation can be refined into two relations based on object modifiers. The definitions and examples for two relationships are listed, with the type of relation underlined and the Object Constraint entity italicized. The one is the Has Status relation when the object modifiers are adjectives (ADJ), verbs (VERB), quantifiers (NUM) or adverbs (ADV). For example, "absolute path string" and "canonical path string". The another is the Has Type relation when the object modifiers are noun (NOUN) or proper noun (PROPN), such as java built-in data types ["byte","int/integer", "float", "char", "boolean", "double", "long", "short"]. For example, "writes a double value, which is comprised of four bytes, to the output stream."

The third type of question is event constraint-oriented and serves to clarify the type or statue of event. For example, given a question "What is the condition under which a path string is converted to a path?" from this question, we can derive the **Event Constraint** entity ("when joined form a path string"). To be converted into "a path", "the sequence of strings" must meet this constraint. The Event Constraint relation naturally arises between the Event entity and the Event Constraint entity. And this relation can be refined into nine types based on the semantic roles (such as Locatives, Directional, Manner, Extent, Temporal, Goal, Purpose Clauses, Secondary Predication, Adverbials) proposed by PropBank [15]. Each relation's definitions and examples are listed, with the underlined part representing the type of relation and the italicized part representing the Event Constraint entity.

- <u>Has Location relation</u> (Location of event occurrence), for example, "finds all the keys of the streams *in this applet context*."
- <u>Has Direction relation</u> (Direction of event occurrence), for example, "moves the focus *down one focus traversal cycle.*"
- <u>Has Manner relation</u> (Manner of event executed), for example, "adds the specified component to the layout, *using the specified constraint object.*"
- <u>Has Extent relation</u> (Scope of event), for example, *"fully* parses the text producing a temporal object."
- <u>Has Temporal relation</u> (Temporal of event occurrence), for example, "converts a path string, or a sequence of strings that *when joined form a path string*, to a path."
- <u>Has Goal relation</u> (Target object of event), for example, "fetches the command list *for the editor*."
- <u>Has Purpose relation</u> (The effect event can achieve), for example, "destroys the orb *so that its resources can be reclaimed.*"
- <u>Has Result relation</u> (The form of the event outcome), for example, "gets a representation of the current choice *as a string*."
- <u>Has Condition relation</u> (Condition of event occurrence), for example, "returns the window object representing the full-screen window *if the device is in fullscreen mode.*"

As for the other semantic roles in PropBank, we do not consider any of them for three reasons. First, not all those semantic roles are Event constraints, for example, the use of Adjectival (ADJ) is limited to noun annotations. Second, Some semantic roles have no practical meaning, such as Discourse (DIS), which is a token (such as "however," "to," "as well as,") that connects one sentence with the previous sentence. Third, some semantic roles, such as Cause Clauses (CAU), which are used to explain causes, are not part of the API's behavior.

Furthermore, the API entity is derived because all of the entities mentioned above are related to the API *"java.nio.file.Paths.get()"*. As a result, the <u>API Has Event</u> relation will naturally emerge between the API and the Event entities.

So far, we've deduced API entities, event relations, event constraint relations, and object constraint relations. And we summarize these three types of relations as the functional relation (also known as intra-relation), which describes the API's functionality from fourteen aspects.

Result extension related knowledge: In addition to the API behavior knowledge, our KG also needs the API semantic knowledge necessary for clarifying the relationship between the result API and other APIs. As a result, we employ seven types of API semantic relations proposed by Yuan et al.[16], namely, Function Similarity, Function Opposite, Function Replace, Function Collaboration, Logic Constraint, Behavior Difference, Efficiency Comparison. Here we only consider these seven method-level API semantic relations and ignore the other non-method-level API semantic relations mentioned in Yuan's work[16] because the Human-AI dialogue we propose is focused on API methods.

3.1.3 KG Construction

Following KG design, we extract API entities and relations required for KG from API descriptions based on semantic and syntactic roles.

Step 1: Semantic and Syntactic Role Annotation. Given a specific API and its description, we annotate the sentence with semantic roles using the natural language tool AllenNLP[17], and obtain the functional parts with functional semantic roles and the constraint parts with constraint semantic roles. The functional parts are then assembled into a functional statement in the correct order, and it is partof-speech tagged to yield grammatical parts with six different syntactic roles, such as verb, direct object, preposition, preposition object, direct object's modifier, and preposition object's modifier.

Step 2: Entity and Functional Relation Extraction based on Annotation. We organize these grammatical parts and constraint parts into six entities and fourteen functional relations based on the following rules:

(1) The API entity is the qualified name of a API method that corresponds to the API description.

(2) The Event entity is made up of grammatical parts with the syntactic roles "verb + direct object" or "verb + preposition + preposition object".

(3) The Action entity is the grammatical part with the syntactic role "verb".

(4) The Object entity is the grammatical part with the syntactic role "direct object" or "preposition object".

(5) Along with the formation of four entities (API, Event, Action, Object), the four event relations (API Has Event, Act Has Event, Has Direct Object, Has Preposition Object) are formed naturally.

(6) The Object Constraint entity is the grammatical part with the syntactic role "direct object's modifier" or "preposition object's modifier"; its object modifier determines the type of object constraint relation which it forms with the Object entity. If the object's modifier is an adjective (ADJ), verb (VERB), quantifier (NUM) or adverb (ADV), the object constraint relation is the Has Status relation; otherwise, it is the Has Type relation.

(7) The Event Constraint entity is the constraint part with constraint semantic roles; its semantic role determines the type of event constraint relation which it forms with the Event entity. One constraint semantic role corresponds to one event constraint relation, that is, ARGM-LOC corresponds to Has Location, ARGM-DIR to Has Direction, ARGM-MNR to Has Manner, ARGM-EXT to Has Extent, ARGM-TMP to Has Temporal, ARGM-GOL to Has Goal, ARGM-PRP to Has Purpose, ARGM-PRD to Has Result, and ARGM-ADV to Has Condition.

So far, six types of entities and fourteen kinds of functional relations (known as intra-relation) have been extracted from an API and its description. To help accurately retrieving functional relations related to a specific API in subsequent sections, we have set up a FUNCTION property for each API entity and stored the extracted functional relations from the API description in this property, as shown in Fig.3-a.

Step 3: Semantic Relation Extraction. As for the seven types of API semantic relations (known as inter-relation), we refer to the API-Task knowledge graph proposed by Yuan et al.[16], which consists of many API relation triples, each of which is of the form (API name, semantic relation Name, API name). If the API names of two API method entities match two API names in a triple, this triple's semantic relation becomes the relation between these two entities. Despite the fact that the API name in our API entity is a full qualified name (FQN) while the API name in the triple of API-Task KG is a simple name, we can successfully match because the simple name can be converted to the FQN based on the three points listed below.

(1) the method-level APIs to match all have parentheses, a key distinguishing symbol, for example, "*update()*".

(2) For the simple method name made up of two or more tokens, one token represents a method and the other represents a class. The package can be reasoned out from the method and the class, and then we combine method, class, and package to get FQN. For example, we can infer the "java.io" package from the "InputStream" class and the "read()" method of "*InputStream.read(*)".

(3) Even if two simple method names have only one token, they belong to the same class if they appear together. As a result, we can reason class, then package from the class and the method, and finally FQN. For example, for the triple (*update(*), Function Similarity, *do-Final(*)), since two methods "update" and "doFinal" appear together, we can infer two classes "*javax.crypto.Cipher*" and "*java.crypto.Mac*", and further two triples with FQNs, namely, (*javax.crypto.Cipher.update(*), Function Similarity, *javax.crypto.Cipher.doFinal(*)) and (*java.crypto.Mac.update(*), Function Similarity, *java.crypto.Mac.update(*),

3.2 Human-Al Dialogue Process

The Human-AI dialogue process is underpinned by Subgraph Search and Clarification Question Generation, which



Note: Figure (a), (b), and (c) illustrate the step-by-step generation of clarifying dialogues in Figure (d) through KAHAID, with emphasis on the blue partition.

Fig. 3: Clarification Question Generation Process

makes KAHAID to generate clarification questions alongside diverse options in each round of dialogue, assisting users in clarifying their under-specified API query intent.

3.2.1 Subgraph Search

KAHAID, with the help of the API behavior KG, can generate a subgraph for each query. This subgraph contains extensive API function knowledge, allowing multiple rounds of human-AI dialogue from fourteen aspects (technically, fourteen different types of functional relations) to clarify ambiguous queries and recommend appropriate APIs.

Given an under-specified API query, KAHAID searches for the top-N candidate APIs using established API search models (such as DeepAPI [9], BIKER [8], and RreMA [10]). The API search model is set up on the same data source as our KG (Java SE 8 API document) for keyword matching, enabling alignment of candidate APIs with at least N API entities in our KG. It should be noted that a single candidate API may correspond to multiple API entities in our KG, as candidate API names lack parameters while API entities in our KG include them. For example, the candidate API "*java.nio.file.paths.get*" can match multiple APIs with different arguments in our KG, such as "*java.nio.file.paths.get(java.net.URI)*" and "*java.nio.file.paths.get(java.lang.string, java.lang.string)*".

KAHAID extends the matched APIs to gather more query-related API entities by fetching API Has Event relations from the FUNCTION property of matched API entities and searching our KG for entities with these relations.

Finally, KAHAID considers the retrieved API entities and their functional relations in FUNCTION properties as a query-related API behavior subgraph, similar to the example in Fig. 1 (left). The functional relations are represented as $\langle e1, r, e2 \rangle$, indicating a relation called r between API entities e1 and e2. Note that the *API Has Event* relation is excluded of the subgraph as it has been used to extend the API (as mentioned above).

3.2.2 Clarification Question Generation

In this process, we use the subgraph built-in 3.2.1 to generate clarification questions and a variety of options for each round of the dialogue process. For each functional relation $\langle e1, r, e2 \rangle$ in the subgraph, e1 and r form a specific aspect (a aspect that under-specified query needs to clarify), and e2 is the option of that aspect. This means that we can generate clarification questions and options by selecting the best aspect from the subgraph.

To generate clarification questions (CQs) from the subgraph, we refer two types of CQ generation approaches: the task extraction approach[18] and the decision treebased approach[19]. The task extraction approach identifies grammatical roles from the comment and selects the most frequent ones as aspects to clarify. The templates are then created based on the aspects to generate the CQs. However, this approach lacks guidance as it does not consider aspect priorities. In contrast, the decision tree-based approach can classify function aspects with uncertain options by estimating probabilities of candidate APIs[20], providing better guidance. In our current work, we employ the decision tree-based approach. This approach comprises four steps, with the first three involving the construction of a complete decision tree, and the fourth step entailing the generation of CQs based on the decision tree.

Step 1: Organize each API and functional relation into an attribute table. Constructing an attribute table is a pivotal step in the construction of a decision tree, where APIs and functional relations are organized in a two-dimensional format. The attribute table facilitates convenient comparison of different aspects, aiding in the identification of the most critical aspect that requires clarification in a query. We refer to API entities and their functional relations as API{ $\langle e1, r, e2 \rangle$,...} (as shown in 3-a), which are then converted to the attribute table with one API column and multiple aspect columns. This process is depicted in the progression from Fig. 3-a to b.

As shown in Fig. 3-b, the first column is API column, which contains all API number in Fig.3-a. Columns after first column are aspect columns. For a functional relation $\langle e1, r, e2 \rangle$ in Fig.3-a, we use its aspect "e1#r" as the column name and option "e2" as the column value. Note that if r in $\langle e1, r, e2 \rangle$ is an *Action Has Event* relation, we use "action#Has Event" directly as the column name (rather than "e1#r"). For example, in Fig.3-a, the APIs enclosed in blue boxes all have the same functional relationship $\langle 15$, Has Event, 7824 \rangle . We use "action#Has Event" (instead of "15#Has Event") as the second column name in Fig.3-b.

Step 2: Select critical aspect based on the attribute table and split the subgraph to get sub-datasets. When construct a decision tree, it is necessary to the critical aspect(the first line of aspect column) from the attribute table. A critical aspect assists in minimizing the height of a decision tree, resulting in improved efficiency in the question-answering process. Two decision tree strategies, ID3 [21] and C4.5[22], are available for selecting aspect. C4.5 employs information gain ratio for aspect dialogue round selection, but increases average dialogue rounds compared to ID3, which may not expedite user intention clarification. In our current implementation, we use ID3 to generate the CQ by selecting the aspect column with the highest information gain. Higher information gain corresponds to more options or APIs associated with the question, which helps reduce dialogue rounds and guide users to clarify their intentions quickly.

The information gain of the aspect column is calculated by Eq.1, which is the difference between the information entropy of all candidate APIs and the information entropy of the current aspect column. Entropy measures the uncertainty of a random variable and characterizes the impurity of examples. Higher entropy indicates more information content [21].

The information entropy of all candidate APIs is calculated by Eq.2 where m is the number of candidate APIs and P_i is the likelihood that the i-th API appears in all candidate APIs. The information entropy of current aspect column is calculated by Eq.3, where k is the type of column values in the aspect column, " API_{1j} , ..., API_{mj} " represents m APIs associated with the j-th column value.

$$Gain(aspect) = I(API_1, ..., API_m) - E(aspect)$$
(1)

$$I(API_1, ..., API_m) = -\sum_{i=1}^{m} P_i \log_2 P_i$$
 (2)

$$E(aspect) = \sum_{j=1}^{k} \left[\frac{|API_{1j}, ..., API_{mj}|}{|API|} \times I(API_{1j}, ..., API_{mj}) \right]$$
(3)

Based on the calculation, we identify the aspect column with the highest information gain from the attribute table. We refer the aspect of this column as the current node in the decision tree, with the different column values serving as edges connecting to current node (including null values).

In order to generate child nodes for each edge, it's need to partition the subgraph. We group API{ $\langle e1, r, e2 \rangle,...$ } with the same column value together to form sub-datasets. For example, the three API{ $\langle e1, r, e2 \rangle,...$ } enclosed in the blue box in Graph A form a sub-dataset, which grouped together based on the column value 7824. At this point, we have completed the construction of one layer of nodes and edges in the decision tree.

Step 3: Recursively repeat Step 1 and Step 2 to construct a complete decision tree that supports dialogue. In order to construct a complete decision tree, for each subdataset, we repeat the Step 1 and Step 2 recursively to build child nodes until the following stopping criteria are met. 1) The sub-dataset only contains one API. 2) The sub-dataset contains multiple APIs with identical functionalities. When the stopping criteria are met, we refer all APIs in current sub-dataset as the current node, which is set as a leaf node in the decision tree. It's worth noting that when building a new attribute table, we remove the aspects that have already been selected to prevent generating redundant clarification questions.

Step 4: After building the decision tree, KAHAID generates the human-AI dialogue process. Specifically, we generate the CQ and diversity options by using the decision tree's current node and all of its edges. Based on different aspect of the current node, CQ can be generated in the following templates:

- *action#Act Has Event*: What do you want to do?
- *object#Has Status*: What kind of the {object} do you want?
- *object#Has Type*: Which type of the {object} do you want?
- *event*#*Has Location*: Where the {event} will be done?
- *event*#*Has Direction*: Where is the direction of {event}?
- event#Has Manner: How would you prefer to {event}?
- event#Has Extent: How far would you want to {event}?
- event#Has Temporal: When do you have to {event}?
- event#Has Goal: Which object do you want to serve by {event}?
- event#Has Purpose: Which purpose do you want to satisfy by {event}?
- event#Has Result: What is the form of the results of {event}?
- event#Has Constraint: Under which condition can {event}?

Following the user's selection of options, the decision tree is pruned, leaving only the user-selected branch's subtree. If the sub-tree has only one node, the answer will be recommended directly. Otherwise, a new round of CQ and options will be generated. During the dialogue process, users can manually stop generating new round of CQ and options at any point. In such cases, we consider all APIs included in the sub-tree as the recommend API sequence.

Fig. 3-c to d illustrates how CQs are generated for the query "get the current working directory?" based on the decision tree. KAHAID generates two rounds of dialogue

to clarify events and objects separately. Two round of CQ are generated by two node (action#Act Has Even, 7823#Has Status), and options for CQ are generated by node's edge.

3.3 Result Extension and Interpretation

After multiple dialogue rounds, KAHAID would further extend the result APIs and interpret all result APIs. Specifically, KAHAID locates the corresponding API entity in the sub-graph for each result API and searches for the semantic relations (e.g., Function Similarity, Function Collaboration) that contain this API from KG. The extended API is then the API corresponding to another entity in these semantic relations. For example, in Fig. 1 (left), the result API java.io.File.getAbsolutePath exists in a semantic relation (java.io.File.getAbsolutePath, Function Similarity, java.io.File.getCanonicalPath, where java.io.File.getCanonicalPath serves as the extended API. Note that duplicate APIs, whether among the extended APIs or with the result APIs, are removed. These extended APIs enable users to be inspired to find new ways to solve problems and meet their API needs more comprehensively.

Meanwhile, KAHAID provides API descriptions and highlights keywords to make the result APIs more interpretable. There are two types of highlighted keywords. The first type includes semantic relation names between result APIs and extended APIs, which explain how the result APIs are related to each other. These keywords appear after the extended API. For example, "Function Similarity" appears after the extended API java.nio.file.Path.toAbsolutePath, indicating that it implements functionality similar to API java.io.File.getAbsolutePat. The second type of highlighted keywords come from functional aspects with various options that were clarified during the dialogue process. KA-HAID searches the decision tree for the path that includes the API in question and other nodes and edges in between. It then obtains keywords from this path and highlights them in the API description. For example, if the query is "How to get the current working directory in Java?" and the API java.io.File.getAbsolutePat is included in the path, KAHAID obtains keywords such as "returns", "absolute", and "path string" from this path and highlights them in the API description. This result explanation helps increase user trust in the result APIs and enables informed API selections.

4 EVALUATION DESIGN

4.1 Research Questions

To evaluate the performance of KAHAID, we conduct a series of experiments to answer the following research questions:

RQ1: How Performance is KAHAID compare with Existing API Search Approaches [8], [23]

RQ2: How Performance is KAHAID compare with Existing Human-AI Dialogue Approach [23]

RQ3: How semantically diverse are the question options generated during the Human-AI Dialogue?

RQ4: What factors can improve the dialogue efficiency?

4.2 API Behavior Knowledge Graph Building

We collect data from JAVA API documentation to construct an API Behavior KG in order to evaluate KAHAID. First, we download the JDK 1.8 API reference documentation ⁶ and parse each API class's web page to extract API methods and their descriptions, yielding 30,200 API method-description pairs for building a Java API dictionary.

Second, following the knowledge graph construction method described in Section 3.1, we extract 48,420 entities (which includes 25,020 API entities and 23,400 other entities such as Action, Event, Object, Object Constraint and Event Constraint) and 89,160 relations from the 30,200 API method-description pairs, and save 25,020 API descriptions. Note that if an event entity cannot be extracted from a description for each API method-description pair, we do not treat this API method as an API entity and do not save this description. For a more in-depth discussion, see Section 6's threat analysis.

Finally, we group these extracted entities and relations into an API Behavior KG, which was stored in Neo4j⁷.

4.3 Dataset

To comprehensively evaluate the performance of KAHAID, we obtained two types of data from Stack Overflow (SO) to gather experimental queries and ground-truth APIs. The first type of data consists of queries paired with a single ground-truth API for each query, allowing us to assess KA-HAID's ability to recommend the most appropriate API. The second type of data comprises queries paired with multiple ground-truth APIs, including the best one and related ones, which enables us to evaluate KAHAID's effectiveness in knowledge expansion.

4.3.1 Dataset with Single ground-truth API

For dataset with single ground-truth API, we reuse SO data from the state-of-the-art approach BIKER [8], which were collected from the official data dump of SO by following criteria: 1) the question is related to Java JDK programming, 2) the question should have a positive score, and 3) at least 1 answer to the question contains API entities and the answer's score should be positive.

The APIs were extracted from the code snippets in markdown scripts of the accepted answers in SO. In a markdown script, code snippets are wrapped by $\langle \text{code} \rangle$ tags. One can use regular expressions to localize the code snippets and further extract the APIs. BIKER also provided a test dataset for evaluating its performance. The test data contains 413 questions along with their ground-truth APIs. We use the title of these 413 questions as the query for evaluation. Since the ground-truth APIs only have a single API, we refer it as a best API.

Note that, BIKER's test dataset mainly contains SO posts with high quality, which cannot reflect the overall quality of SO posts. Thus, we have also created a random test datasets which contain randomly selected SO posts for removing human bias (the same dataset creat manner has also been used in the comparison of CLEAR and BIKER in CLEAR's paper [23]).

^{6.} https://docs.oracle.com/javase/8/docs/api

^{7.} https://neo4j.com/

4.3.2 Dataset with Multiple ground-truth API

For dataset with Multiple ground-truth API, we manually collect 60 API-related questions with multiple ground-truth APIs from 60 SO posts following BIKER's criteria [8].

The questions posted on Stack Overflow typically receive answers and comments. To assess KAHAID's knowledge expansion ability, ground-truth APIs must include the best API and extended APIs with seven semantic relations to best API. These relations, Function Similarity, Function Opposite, Function Replace, Function Collaboration, Logic Constraint, Behavior Difference, and Efficiency Comparison, are described in Section 3.1.3. Unlike BIKER, we consider APIs from all positively-scored answers and comments, not only from accepted answers. This is because the accepted answer often only includes a single API, while non-accepted answers and comments often provide more extended APIs.

To obtain the best API and extended APIs, we asked an expert and two external annotators (two master students familiar with Java but unaffiliated with this work) to review all answers and comments in specific question posts. After a 30-minute training on classifying non-best API methods, the two annotators reviewed the API methods separately. The expert made the final decision if there was disagreement. Finally, we obtained 60 queries and 298 ground-truth API methods, including 60 best API methods and 238 extended API methods with varying semantic relations.

4.3.3 Evaluation Datasets

In summary, we adopt three test datasets covering three different scenarios, i.e., high-quality dataset with single ground-truth API (i.e., BIKER's test dataset), real-wold random dataset with single ground-truth API, and 60 manual SO dataset with multiple ground-truth API.

- BIKER test dataset: is the evaluation dataset of BIKER, which contains 413 manually selected and verified SO queries with API answers.
- Random test dataset: contains 1K random selected SO queries with API answers from BIKER's training dataset.
- Multi-API SO test dataset: contains 60 manually selected SO queries with multiple relevant APIs.

4.4 Baselines

We compared KAHAID with BIKER [8], CLEAR [23], ZaCQ [18], which are three state-of-the-art API recommendation techniques.

Baseline1 (BIKER) [8]: is an API search approach that uses a word-embedding model to calculate the semantic similarity between a query and each API description, returning the top-N API methods with similarity scores. Additionally, BIKER calculates the similarity score between the given query and other queries on Stack Overflow, enabling it to return similar queries and extend the given query with other related queries. By doing so, BIKER is capable of returning a broad range of API methods. Its source code can be downloaded on Github⁸.

Baseline2 (CLEAR) [23]: is a API search approach based on BERT sentence embedding and contrastive learning. CLEAR selects a set of candidate Stack Overflow posts based on BERT sentence embedding similarity and reranks them using a BERT-based classification model to recommend the top-N APIs. Two different models were trained in CLEAR for recommending class-level APIs and methodlevel APIs. Our evaluation in this work focuses solely on the performance of the API recommendation at the methodlevel model. Therefore, we solely utilize the method-level model for our experiments. Its data and source code can be downloaded on⁹.

Baseline3 (ZaCQ) [18]: is a source code search method that re-ranks code snippets by interactively refining queries with multiple rounds of clarifying questions. Following ref [18], we select ZaCQ's Top-3 code snippets after the first round of question clarification, in which we collect all API methods as the result API methods. Its data and source code can be downloaded on Github¹⁰.

4.5 Performance Measures

Following existing studies [23], we use Mean reciprocal rank (MRR) [24], Mean average precision (MAP) [25], Precision, and Recall to evaluate the performance of API recommendation approaches. MRR and MAP are the widely accepted measurements for information retrieval. MRR measures the effort needed to find the first correct answer in the recommended list and MAP considers the ranks of all correct answers. We also evaluate the performance with Precision and Recall, which can be defined as follows:

$$Precision = \frac{recommendedAPI \cap ground - truthAPI}{recommendedAPI}$$
(4)

$$Recall = \frac{recommendedAPI \cap ground - truthAPI}{ground - truthAPI}$$
(5)

5 RESULT ANALYSIS

5.1 RQ1: How Performance is KAHAID compare with the Existing API Search Approaches [8], [23]?

5.1.1 Method

We compared KAHAID with two existing API search baselines, BIKER [8] and CLEAR [23], on three different test datasets (see section 4.3.3). As the authors of BIKER and CLEAR have provided replication packages, we used these to conduct experiments and compare the results.

KAHAID generates CQs with options for each dialogue round, which are utilized to recommend different API sequences depending on the selected options. To simulate user behavior, we device an option selection strategy based on the assumption that the user always select the option that leads to the best API for their query. This strategy selects an option only if the resulting API sequence includes the ground-truth's best API; otherwise, KAHAID's recommended API sequence is considered empty. To conduct a fair and objective comparison between KAHAID and the baselines, we assess their top-5 API in recommended API sequence for each query. We consider a API is correct if it is the best API in the ground-truth APIs.

^{9.} https://github.com/Moshiii/CLEAR-replication 10. https://github.com/Zeberhart/ZaCQ

5.1.2 Result

TABLE 1: Performance Comparison of API Search Approach.

Dataset	Metric	BIKER	CLEAR	KAHAID
	MRR	0.614	0.755	0.610
BIKER test dataset	MAP	0.143	0.765	0.503
	Precision	0.249	0.550	0.693
	Recall	0.714	0.764	0.853
Random test dataset	MRR	0.253	0.413	0.428
	MAP	0.096	0.366	0.365
	Precision	0.110	0.293	0.639
	Recall	0.301	0.397	0.516
	MRR	0.359	0.523	0.769
Multi-API SO	MAP	0.219	0.243	0.794
test dataset	Precision	0.152	0.353	0.839
	Recall	0.733	0.251	0.867

Table 1 shows the result of KAHAID compared with the other baselines. For dataset with single ground-truth API (BIKER test dataset and Random test dataset), overall KAHAID outperforms both BIKER and CLEAR. Especially on the BIKER test data, KAHAID achieved a high recall of 0.853, which is an improvement of 19.5% and 11.6% over BIKER and CLEAR, respectively. This indicates that KAHAID can accurately recommend the most appropriate API for 85.3% of query. Note that, Both BIKER and CLEAR have the same performance reported in this work and CLEAR's paper. However, different from the comparison reported in BIKER's paper, the performance of BIKER and CLEAR was worse, especially on the Random test data. The four metrics (MRR, MAP, Precision, Recall) of BIKER are all below 0.301, and the four metrics of CLEAR are all below 0.413. This is because we used the same random method to get dataset as the CLEAR paper, but it may not coincide with the random test dataset in CLEAR's paper. Additionally, BIKER and CLEAR focus on accepted answers, which are not always the best APIs. On the contrary, KAHAID guides users to clarify their intentions through dialogue and accurately recommend APIs that match their intentions, not just limited to accepted answers. For example, given the question "Getting the name of t he currently executing method ¹¹", KAHAID recommends the method java.lang.Class.getEnclosingMethod() as the best API, which is present in the comment section of the highestscoring answer rather than the accepted answer. Instead, both BIKER and CLEAR recommend the accepted answer, i.e., java.lang.Thread.getStackTrace(), which has a flaw as it may occasionally return a zero-length array, as described in the API documentation¹².

For the Muti-API SO test data, KAHAID outperforms BIKER (by 114.2%, 262.6%, 452%, and 18.3% improvement) and CLEAR (by 47%, 226.7%, 137.7%, and 245.4% improvement) on all four evaluation metrics (MRR, MAP, Precision, Recall). This indicates that compared to existing API search approaches, KAHAID has strong knowledge expansion ability. Meanwhile, we noticed that BIKER achieves a recall of 0.733, 192% higher than CLEAR and only 15.5% lower than KAHAID. This is because BIKER's method of calculating similarity between SO posts and APIs is more

11. https://stackoverflow.com/questions/442747/

12. https://docs.oracle.com/en/java/javase/11/docs/api/java.base-/java/lang/Thread.html#getStackTrace() tolerant to variations in questions compared to CLEAR. However, we do not consider this as a form of knowledge expansion, as the API sequence recommended by BIKER contains many APIs that are not related to the query, which is also the reason for its low precision.

In comparison to existing API search methods BIKER and CLEAR, the human-AI dialogue method KAHAID has a strong ability for API recommendation and knowledge extension.

5.2 RQ2: How Performance is KAHAID compare with Existing Human-AI Dialogue Approach [23]

5.2.1 Method

Real-world queries often have multiple correct answers that are semantically related, such as through Function Similarity or Function Collaboration. To address this, human-AI dialogue approaches recommend different APIs from various aspects, while also using knowledge expansion to identify APIs with semantic relations. To compare KAHAID with the existing human-AI dialogue approach, ZaCQ, we utilize the Multi-API SO test dataset described in section 4.3.2. This dataset includes both the best API and other APIs with semantic relations to the best API. We employ the same option selection strategy as mentioned in section 5.1.1 to simulate user behavior for both KAHAID and ZaCQ. Then, we calculate metrics (MRR, MAP, Precision, Recall) for the API sequences recommended by KAHAID and ZaCQ from the first round of dialogue to the third. This enables us to evaluate and compare the performance of both human-AI dialogue approaches in recommending semantically related APIs.

5.2.2 Result

TABLE 2: Performance Comparison of query clarification Approach.

Metrics	Approaches	round 1	round 2	round 3
MRR	KAHAID	0.506	0.769	0.815
	ZaCQ	0.524	0.542	0.574
MAP	KAHAID	0.508	0.794	0.845
	ZaCQ	0.521	0.576	0.582
Precision	KAHAID	0.654	0.839	0.842
	ZaCQ	0.500	0.615	0.615
Recall	KAHAID	0.721	0.867	0.875
	ZaCQ	0.633	0.628	0.638

Table 1 shows the result of KAHAID compared with ZaCQ. KAHAID demonstrated a significant improvement from the first to the second round of dialogue, with increases of 52%, 56.3%, 28.2%, and 21.8% in MRR, MAP, Precision, and Recall, respectively. The third round showed the highest values with MRR, MAP, Precision, and Recall of 0.815, 0.845, 0.842, and 0.875, which ourperfors the ZaCQ by 42%, 45.2%, 36.9%, and 37.1%. This demonstrates that KAHAID has a strong capability for knowledge extension and can meet the diverse needs of developers. To our surprise, ZaCQ has a poor knowledge extension, as indicated by its low MRR and MAP scores, which did not exceed 0.582. Additionally, its Precision and Recall scores were no higher than 0.638. This is because ZaCQ is merely a tool for re-ranking initial search results through multiple rounds of clarifying conversations. If the best API is not found in the initial search results, ZaCQ fails to obtain the best API no matter how many rounds of re-ranking are performed. KAHAID, on the other hand, can obtain the best API via semantic relation extension even if it is not found in the initial search results. For example, Given a query "How do I remove repeated elements from ArrayList?", ZaCQ cannot recommend the best API "*java.util.stream.Stream.distinct*", either in the initial results or in any subsequent rounds of re-ranked results. Given the same query, KAHAID could not find the best API in the initial results either. However, by extending the API "*java.util.stream.Stream.collect*" in the initial results, the best API can be found in the extended APIs.

In comparison to existing human-AI dialogue baseline ZaCQ, KAHAID has a strong ability for knowledge extension; it works well even when faced with an uncertain and under-specified API query.

5.3 RQ3: How Semantically Diverse are the Question Options Generated during the Human-Al Dialogue?

A variety of options can provide users with additional references and inspiration. In this section, we measure the diversity of CQ options in each human-AI dialogue round.

5.3.1 Method

For 60 experimental queries in the test set, we use KAHAID to generate multiple rounds of human-AI dialogue for each query. Each round of dialogue includes a clarification question and question options. We select every option and allow KAHAID to generate different clarification questions and question options for the next round of dialogue depending on the option selected. In this way, we can explore all possible selection path.

We invite two participants (two master students who are unaffiliated with this work but are familiar with Java) to rate the semantic diversity between two options from 0 to 1. Score 0 indicates that there is no diversity and that the semantics of the two options are extremely similar. Score 0.5 indicates that the two options have moderate diversity and roughly similar semantics, for example, the semantics of option B is just one of many semantics of option A. Score 1 denotes a high degree of diversity, indicating that the semantics of the two options are completely different.

There are four types of options available: Event, Object, Object Constraint, and Event Constraint. The types of options for the same clarification question are the same. We score the Object, Object Constraint, or Event Constraint options according to the semantic diversity scoring criteria described above. We don't judge diversity solely on lexical differences, as some options have different words but have same semantics (e.g., "path" and "path object"). That is also why we manually assess diversity.

The Event option consists of verbs and objects. The diversity score between two such options is rated based on two conditions: the verbs in options are diverse, and the two objects in options are diverse. If two conditions are met, the diversity score is 1; if neither condition is met, the score is 0; and if only one condition is met, the score is 0.5. We score semantic diversity of objects in the same way that we score Object options. To score the diversity of verbs, we refer to the 87 verb classes summarized by Xie et al [10] who group verbs that have the same function into a single class. Two



Fig. 4: Diversity of Question Options in Each Round

verbs are diverse if they do not belong to the same class in function.

We calculate the semantic diversity of a related option in a clarification question by averaging the Semantic diversity of each related option and the other options in the clarification question. Then, for each clarification question in a dialogue round, we average the semantic diversity of question options to obtain the Semantic diversity of question options in this dialogue round. To obtain the diversity of question options in the entire query dialogue, we average the diversity of question options across multiple rounds of human-AI dialogue. Finally, we compute the overall diversity of question options in human-AI dialogue by averaging the diversity of question options across 60 query dialogues.

5.3.2 Result

Fig. 4 depicts, around the 60 questions, the diversity of question options generated in each round of human-AI dialogue (such as 0.751 in the first round, 0.719 in the second round, and 0.761 in the third round), as well as the overall diversity of question options for the 60 questions (0.749). Note that not every human-AI dialogue includes three dialogue rounds; some have fewer than three rounds, while others have more than three rounds. It is only counted to the rounds that the dialogue should have for less than three rounds, and it does not participate in the average calculation in the subsequent round. When there are more than three rounds, we only count up to the third round.

Diverse question options may encourage divergent thinking and help developers clarify what they really want.. As shown in Fig. 1, in the second CQ round, two different options (path string, path object) are generated, causing the user to realize that the initial question "How to get the current working directory?" is very vague and that he should rewrite the initial question more specifically into "How to get the path object of current working directory?" using the knowledge "path object" obtained from the options.

During the dialogue process, KAHAID can generate a wide variety of options for each clarification question, encouraging divergent thinking and assisting developers in clarifying what he truly desires.

5.4 RQ4: What Factors can Improve Human-AI Dialogue Efficiency?

Human-AI dialogue efficiency is determined by the number of its rounds, which is affected by both external and internal factors.

5.4.1 Method

The uncertainty of the API query is the external factor. The more uncertain the query, the more rounds of dialogue are required to gradually clarify the query. To confirm this, we create three types of uncertain and under-specified queries. These queries are made up of four syntactic roles (verb, direct object, preposition, and preposition object) that are derived from the 25,020 API descriptions mentioned in the KG construction process.

These three types of queries are listed as follows:

(1) 18,683 "V-DO" queries, each with a verb (V) and a direct object (DO). For example, "obtain (V) data time (DO)." (2) 6,543 "V-PO" queries, each with a verb (V), a preposition and a preposition object (PO). For example, "obtain (V) in chronology (PO)." (3) 6,132 "V-DO-PO" queries, each with a verb, a direct object, a preposition and a preposition object. For example, "obtain (V) date time (DO) in chronology (PO)." This type of queries are less uncertain than the "V-DO" queries and "V-PO" queries.

In addition, we also adopt the real-world queries, i.e., 60 queries collect from SO as depict in section 4.3.2.

The decision tree strategy is the internal factor, because the underlying decision tree determines each round of questions and options, as well as what the next round of questions ask. We employ these two decision tree strategies (ID3 and C4.5) to support human-AI dialogue around those three types of queries and compute the average rounds of human-AI dialogue (HAR) by Eq. 6.

$$HAR = \frac{\sum_{i=0}^{n} |Leaf_i| \times Depth_i}{|API|} \tag{6}$$

where $|Leaf_i|$ refers to the number of APIs contained in the i-th leaf node of this decision tree; $Depth_i$ refers to the depth of the i-th leaf node; and |API| refers to the number of APIs contained in all leaf nodes of a decision tree. In the decision tree, because the dialogue ends when receiving any API from the i-th leaf node, the dialogue clarification questions and options are generated based on non-leaf nodes (and their edges) from the root node to the leaf node. As a result, the depth of a leaf node $Depth_i$ in the decision tree can be used to represent the number of dialogue rounds required to obtain any API in the leaf node. Specifically, we use $\sum_{i=0}^{n} |Leaf_i| \times Depth_i$ to compute the sum of the number of human-AI dialogue rounds required to obtain each API in a decision tree, and then make it divide the number of all APIs in the decision tree to calculate HAR. The fewer rounds of dialogue there are on average, the more efficient the human-AI dialogue.

5.4.2 Result

As shown in Fig. 5-a, regardless of the query, ID3's HAR is always lower than C4.5's and is no more than 3, which suggests that ID3 is more efficient than C4.5. This is because C4.5 calculates an information gain ratio based on ID3 and then selects the aspect dialogue round. This type of CQ has fewer options, which reduces the difficulty of making decisions per dialogue round. However, it increases the average number of dialogue rounds, which is not conducive to guiding users to quickly clarify their intentions. As a result, we do not use C4.5. Furthermore, when compared with "V-DO-PO" queries, "V-DO" queries and "V-PO" queries have



Fig. 5: Internal and External Factors Influencing the Human-AI dialogue Efficiency

greater uncertainty, resulting in a much larger proportion of HAR greater than 0, as shown in Fig. 5-b. This shows that the more uncertain the API query, the more dialogue rounds are required. It also implies that our method can assist users in clarifying questions through multiple dialogue rounds.

The ID3 decision tree strategy improves the efficiency of question clarification and reduces the number of human-AI dialogue rounds.

6 USER STUDY 6.1 Study Design

6.1.1 Test Set

8 questions and their ground-truth API methods are chosen at random from the test set described in Section 4.3.2 for the user study, as shown in Table 3.

6.1.2 Participants

We recruited 12 participants from both university and IT companies. Six of them (6 graduate students) are from the first author's university, and six of them are from two IT companies. All of them have Java developing experience in either commercial or open source projects, and the years of their developing experience vary from 1 year to 5 years, with an average of 2.9 years.

Through a pre-study survey, we ensure that none of these students had encountered the experimental questions before. We divided the 12 participants into two groups, with each group containing three graduate students and three participants from IT companies. *GroupZ* used ZaCQ and *GroupK* used KAHAID. Each group member was asked to answer 8 questions using the specific tool.

6.1.3 Study Setup

We gave all group members a 20-minute training session to teach them how to use the specific tool because they were expected to use it to answer 8 questions.

Given the goal of this user study is to investigate the user experience with the tool rather than whether the user can use the tool to obtain the ground-truth API method, we set up the following scenario: If the result API is the ground-truth API method, the tool tells the user that he succeeded and allows the user to answer the next question; otherwise, the tool encourages the user to interact with the tool again until the ground-truth API method is found. If the user cannot find the ground-truth API method within 5 minutes, he will be informed that he has failed and will be permitted to answer the next question.

PID	StackOverflow ID	Query	Best API
Q_1	153724	How to round a number to n decimal places in Java?	java.text.DecimalFormat.format
Q_2	4871051	How to get the current working directory in Java?	java.io.File.getAbsolutePath
Q_3	5868369	How can I read a large text file line by line using Java?	java.io.BufferedReader.readLine
Q_4	2860943	How can I hash a password in Java?	javax.crypto.SecretKeyFactory.generateSecret
Q_5	1069066	Convert Date to String?	java.lang.Thread.getStackTrace
Q_6	428918	How can I increment a date by one day in Java?	java.time.LocalDate.plusDays
Q_7	35842	How can a Java program get its own process ID?	java.lang.management.ManagementFactory.getRuntimeMXBean
Q_8	9481865	Getting the IP address of the current machine using Java?	java.net.InetAddress.getLocalHost



Note that the percentages of people who scored 2, 1, and 0 are represented by yellow, blue, and green, respectively. G_k refers to the group using KAHAID; G_z refers to the group using ZaCQ.

Fig. 6: The Proportion of Ratings Given to Each of the Six Indicators.

While answering 8 questions with the tool, we ask each group member to rate the process of answering each question in terms of the following 6 indicators:

- *Syntax,* which measures the syntactic correctness of the generated clarification questions.
- *Logic*, which assesses the logical correctness of the generated clarification questions.
- *Relevance,* which measures the relationship between the generated clarification questions and the query.
- *Inspiration,* which assesses how well the clarification question options generated enlighten the user.
- *Extensibility*, which measures how well the extended API meets the query.
- *Interpretability*, which measures how well the resulting API is understood in relation to each other and to the query based on API descriptions and highlighted keywords in these descriptions.Note that only *GroupK* are required to score this indicator because only KAHAID can explain why the resulting API was found.

Each of these indicators has a score between 0 and 2, with 0 indicating dissatisfaction, 1 indicating satisfaction, and 2 indicating extreme satisfaction. Finally, we ask each group member to write down their feedback, which include both advantages and disadvantages.

6.2 Participants' Feedback

Both positive and negative feedback about KAHAID are posted, respectively.

- Advantages:
 - The majority of clarification questions were syntactically and logically correct, as well as closely related to queries. They were extremely helpful in clarifying my query requirements.
- I particularly enjoyed the extended APIs because they could teach me new things. The API with the efficiency comparison, in particular, surprised me by responding to the query more quickly.
- Various options presented during the QA process deeply inspired me to solve problems in previously unknown ways.
- These API descriptions and highlighted API keywords were useful. They explained how APIs related to queries and what APIs could do, giving me more confidence in my final decision.

Disadvantages:

- Some clarifying questions, while grammatically and logically correct, are not expressed naturally. This problem adds some comprehension time to the questions.
- The presentation of result APIs is a little perplexing. The layout is clumsy, especially when showing multiple result APIs at once, which makes my experience unpleasant.

6.3 Indicator Analysis

We collected each group member's rating results on different indicator, and plotted them in Fig 6. This figure shows that G_k outperforms G_Z across the indicators. Although more than 90% of G_k and G_Z gave Syntax, Logic, and Relevance scores greater than 0, the percentage of G_k with a score of 2 was higher than G_Z (56.3% vs. 41.67%, 54.17% vs. 47.92%, 50% vs. 33.33%). In terms of Inspiration and Extensibility, the G_k and G_Z score results are significantly different. Compared to ZaCQ, on scores of 1 and 2, KAHAID improves by 20.83% Inspiration and by 12.5% Extensibility. Through analyzing the options participants selected during human-AI dialogue process as well as the result API they obtained, we also have the following findings:

First, participants prefer options that inspire them to think of new ways to answer the questions. For example, G_k assigned a higher Inspiration score to Q2. This illustrates that the options generated by KAHAID are more illuminating than those generated by ZaCQ. Specifically, ZaCQ generated a clarification question for O2: "Are you interested in getting the current working directory for the Default File System?" with only two options "Yes" or "No." As a result, G_z were not inspired beyond the "default file system" strategy. In contrast, G_k were inspired by the diverse options generated by KAHAID in the first round ("return filesystem", "return path" and "converting path string to path Object"). G_k discovered that, in addition to "return filesystem", they could get the current working directory by "returning path" and "converting path string to path Object". KAHAID, in particular, employs the "path object" option to encourage G_k to obtain the current working path via the Path Object. Finally, 83.33% of Gk participants chose this option and assigned a score of 2 to Q2.

Second, participants prefer the extended APIs that remind them of implicit knowledge they were previously unaware of. The Extensibility score for these extended APIs was frequently higher. For Q5, for example, KAHAID recommended an extended API "*java.text.DateFormat.format*", which has a Efficiency Comparison relation with the best API "*java.lang.String.format*". This extended API provides G_k with a faster response to the query, which most participants were previously unaware of. As a result, 75% of G_k gave Q5 a Extensibility score of 2 and 25% gave it a score of 1. Although the APIs suggested by ZaCQ can also be used to answer this query, they are all well-known to G_Z participants. Finally, all of the Gz gave Q5 Extensibility scores of 1.

Third, if the API method is explainable, participants can find the best API method more easily or with greater confidence. Q7, for example, received a 100% Interpretability score of 2 from G_k . Both KAHAID and ZaCQ can find its ground-truth API method "ManagementFactory.getRuntimeMXBean" for Q7. However, due to the API's lack of interpretability, half of G_z could not associate it with the query simply by its name. In contrast, G_k can has access to the API method interpretation, which includes the API descriptions and highlighted API keywords. Following that, G_k understands how this API method relates to the query through the keyword ("return managed bean" and "runtime system"), and what it is capable of according to the API description. Eventually, G_k found this API method and was convinced that it was the best API method.

KAHAID can provide a good user experience based on user ratings and feedback in six areas: syntax, logic, relevance, inspiration, extensibility, and interpretability.

7 THREATS TO VALIDITY

One intrinsic threat is the use of the AllenNLP, a generalpurpose natural language processing library. AllenNLP is used to parse sentences in both the semantic and syntactic analyses of the API function descriptions. However, none of the tools, including AllenNLP, can perfectly parse large amounts of textual data. Furthermore, AllenNLP is not specifically designed to parse API function descriptions containing code elements, incomplete sentences, or common syntax errors. To mitigate this threat, we devise heuristics for adapting AllenNLP to parsing API function descriptions. AllenNLP, for example, has trouble parsing API descriptions that frequently begin with a verb. As a result, we devise a heuristic rule: such incomplete sentences are prefixed with "This method", which aids AllenNLP in parsing the sentence components. For another example, the content in parentheses is useless, which affects AllenNLP's ability to parse the sentence. As a result, we device a heuristic rule that automatically removes the content in parentheses in the sentence, improving AllenNLP's ability to parse the sentence.

One **external threat** is the generality of our API behavior KG. Because we built it entirely from official Java API documentation, the KG cannot play a role in a broader domain, such as the domain of other languages. In the future, we plan to expand KG with additional programming language documentation, allowing it to handle a broader range of programming issues.

8 RELATED WORK

API search refers to the process of obtaining APIs related to a search query using various modeling techniques [26] such as information retrieval models [27], machine learning models, and deep learning models [28], [8], [29]. If the search query is too broad, these search models are unlikely to find the relevant APIs. To make the query more specific, many query expansion techniques have been proposed, such as expanding query words with relevant software artifacts from official API documents [30], [31], code changes [32], [33], [34], [35], or Stack Overflow posts [31], [36], [37], [38]. While this enables a more specific query and the discovery of the most relevant API, it lacks interpretation and extensibility, making it difficult for developers to interpret search results and preventing them from discovering potentially useful APIs. Furthermore, current API search research is still a long way from social-technical information seeking on online forums [11], where pragmatic API needs suggestive, explainable, and extensible API recommendation, such as exploring alternative or better solutions, and discovering previously unknown API knowledge [1], [2].

To meet practical API needs, we propose KAHAID with intent clarification, result interpretation, and extension capabilities. It differs from current API search research in three ways. 1) In terms of enlightenment, KAHAID interacts with

the developer to clarify actions, objects, or constraints until it finds some APIs. This may motivate him to seek a better solution or to determine his true desires and rewrite the initial vague question, which is not supported by current query expansion methods. Even though ZaCO [18] is a conversational search method, the various aspects of the clarification questions it generates are limited to only the syntactic analysis for the tasks associated with the search results. In contrast, the diverse options for the clarification questions generated by KAHAID are derived from the API behavior knowledge graph, which contains rich API actions, objects, constraints, and various API semantic relations. Furthermore, ZaCQ is merely a tool for re-ranking the initial search results. If the initial search results do not contain the correct API, no matter how many QAs the developer runs, there is no way to find the correct API. Under such conditions, KAHAID, on the other hand, can use KG to extend the correct API by following the relationships between APIs. 2) In terms of interpretation, KAHAID displays the discovered APIs along with a concise explanation of why these APIs are recommended and how they relate to the clarified query. Specifically, KAHAID displays the API's functionality description and highlights the keywords it uses for clarification, as well as the relations between the extend APIs and the most relevant API. 3) In terms of extensibility, KAHAID provides a variety of APIs, including directly relevant APIs and extended APIs, based on the knowledge graph's rich API semantic relations.

9 CONCLUSION AND FUTURE WORK

In this paper, we propose KAHAID as an initial attempt to combine the immediate responsiveness of API search technologies with the interaction, clarification, explanation, and extensibility capability of social-technical information seeking. The systematic evaluation confirms that KAHAID implements an illuminating, interpretable, and extensible exploratory query process.

In the future, in addition to Java SDK APIs, we will implement KAHAID for more APIs, such as Python APIs, Ruby APIs and Go APIs. Furthermore, if KAHAID can teach a user to search an API via Human-KG dialog, we may be able to teach the LLM (Large pre-trained Language Model) [39], [40] to adapt the specific downstream task via LLM-KG dialog. (*ab*This will make KGQA (Knowledge Graph-based Question Answering) more open and flexible.

ACKNOWLEDGEMENTS

The work is partly supported by the National Natural Science Foundation of China under Grant (Nos. 61902162, 62262031), the Natural Science Foundation of Jiangxi Province (20202BAB202015), and the Graduate Innovative Special Fund Projects of Jiangxi Province(YC2021-S308, YC2022-s258).

REFERENCES

 X. Ren, J. Sun, Z. Xing, X. Xia, and J. Sun, "Demystify official api usage directives with crowdsourced api misuse scenarios, erroneous code examples and patches," 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pp. 925–936, 2020.

- [2] Z. Eberhart and C. McMillan, "Dialogue management for interactive api search," 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 274–285, 2021.
- [3] S. Haiduc and A. Marcus, "On the effect of the query in irbased concept location," 2011 IEEE 19th International Conference on Program Comprehension, pp. 234–237, 2011.
- [4] F. Thung, S. Wang, D. Lo, and J. L. Lawall, "Automatic recommendation of api methods from feature requests," 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 290–300, 2013.
- [5] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowdsourced knowledge," 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 349–359, 2016.
- [6] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS*, 2013.
- [7] X. Ye, H. Shen, X. Ma, R. C. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 404–415, 2016.
- [8] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "Api method recommendation without worrying about the task-api knowledge gap," 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 293–304, 2018.
- [9] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016.
- [10] 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," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.
- [11] H. Zhang, S. Wang, T.-H. P. Chen, and A. E. Hassan, "Reading answers on stack overflow: Not enough!" *IEEE Transactions on Software Engineering*, vol. 47, pp. 2520–2533, 2021.
 [12] X. Ren, Z. Xing, X. Xia, G. Li, and J. Sun, "Discovering, explaining
- [12] X. Ren, Z. Xing, X. Xia, G. Li, and J. Sun, "Discovering, explaining and summarizing controversial discussions in community q&a sites," 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 151–162, 2019.
- [13] M. Liu, X. Peng, A. Marcus, S. Xing, C. Treude, and C. Zhao, "Apirelated developer information needs in stack overflow," *IEEE Transactions on Software Engineering*, 2021.
- [14] Y. Liu, M. Liu, X. Peng, C. Treude, Z. Xing, and X. Zhang, "Generating concept based api element comparison using a knowledge graph," 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 834–845, 2020.
- [15] C. Bonial, O. Babko-Malaya, J. D. Choi, and J. D. Hwang, "Propbank annotation guidelines," 2010.
- [16] Q. Huang, Z. Yuan, Z. Xing, Z. Zuo, C. Wang, and X. Xia, "1+1;2: Programming know-what and know-how knowledge fusion, semantic enrichment and coherent application," *ArXiv*, vol. abs/2207.05560, 2022.
- [17] M. Gardner, J. Grus, M. Neumann, O. Tafjord, P. Dasigi, N. F. Liu, M. E. Peters, M. Schmitz, and L. Zettlemoyer, "Allennlp: A deep semantic natural language processing platform," *ArXiv*, vol. abs/1803.07640, 2018.
- [18] Z. Eberhart and C. McMillan, "Generating clarifying questions for query refinement in source code search," in SANER, 2022.
- [19] T. Castle-Green, S. Reeves, J. E. Fischer, and B. Koleva, "Decision trees as sociotechnical objects in chatbot design," *Proceedings of the* 2nd Conference on Conversational User Interfaces, 2020.
- [20] B. Hssina, A. Merbouha, H. Ezzikouri, and M. Erritali, "A comparative study of decision tree id3 and c4.5," *International Journal* of Advanced Computer Science and Applications, vol. 4, 2014.
- [21] J. R. Quinlan, "Induction of decision trees," Machine Learning, vol. 1, pp. 81–106, 2004.
- [22] J. Ross Quinlan, "C4.5: Programs for machine learning," 1992.
- [23] M. Wei, N. S. Harzevili, Y. Huang, J. Wang, and S. Wang, "Clear: Contrastive learning for api recommendation," 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), pp. 376–387, 2022.
- [24] D. R. Radev, H. Qi, H. Wu, and W. Fan, "Evaluating web-based question answering systems," in *International Conference on Lan*guage Resources and Evaluation, 2002.
- [25] M. Sanderson, "Christopher d. manning, prabhakar raghavan, hinrich schütze, introduction to information retrieval, cambridge university press 2008. isbn-13 978-0-521-86571-5, xxi + 482 pages," *Natural Language Engineering*, vol. 16, pp. 100 – 103, 2010.

- [26] C. Liu, X. Xia, D. Lo, C. Gao, X. Yang, and J. C. Grundy, "Opportunities and challenges in code search tools," ACM Computing Surveys (CSUR), vol. 54, pp. 1 – 40, 2022.
- [27] H. C. Wu, R. W. P. Luk, K.-F. Wong, and K.-L. Kwok, "Interpreting tf-idf term weights as making relevance decisions," ACM Trans. Inf. Syst., vol. 26, pp. 13:1–13:37, 2008.
- [28] Q. Huang, A. Qiu, M. Zhong, and Y. Wang, "A code-description representation learning model based on attention," 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 447–455, 2020.
- [29] F. Cai, C. Wang, Q. Huang, Z. Zuo, and Y. Liao, "Search for compatible source code," Int. J. Softw. Eng. Knowl. Eng., vol. 31, pp. 477–502, 2021.
- [30] F. Lv, H. Zhang, J.-G. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model (e)," 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 260–270, 2015.
- [31] G. Hu, M. Peng, Y. Zhang, Q. Xie, W. GAO, and M. Yuan, "Unsupervised software repositories mining and its application to code search," *Software: Practice and Experience*, vol. 50, pp. 299 – 322, 2020.
- [32] Q. Huang and G. Wu, "Enhance code search via reformulating queries with evolving contexts," *Automated Software Engineering*, vol. 26, pp. 705 – 732, 2019.
- [33] Q. Huang and H. Wu, "Qe-integrating framework based on github knowledge and svm ranking," *Science China Information Sciences*, vol. 62, pp. 1–16, 2019.
- [34] Q. Huang, Y. Yang, and M. Cheng, "Deep learning the semantics of change sequences for query expansion," *Software: Practice and Experience*, vol. 49, pp. 1600 – 1617, 2019.
- [35] Q. Huang, Y. Yang, X. Zhan, H. Wan, and G. Wu, "Query expansion based on statistical learning from code changes," *Software: Practice and Experience*, vol. 48, pp. 1333 – 1351, 2018.
- [36] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query expansion based on crowd knowledge for code search," *IEEE Transactions on Services Computing*, vol. 9, pp. 771–783, 2016.
- [37] R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, and Y. L. Traon, "Augmenting and structuring user queries to support efficient free-form code search," *Empirical Software Engineering*, vol. 23, pp. 2622–2654, 2018.
- [38] F. Zhang, H. Niu, I. Keivanloo, and Y. Zou, "Expanding queries for code search using semantically related api class-names," *IEEE Transactions on Software Engineering*, vol. 44, pp. 1070–1082, 2018.
- [39] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. E. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. J. Lowe, "Training language models to follow instructions with human feedback," *ArXiv*, vol. abs/2203.02155, 2022.
- [40] I. Solaiman and C. Dennison, "Process for adapting language models to society (palms) with values-targeted datasets," in *NeurIPS*, 2021.



QING HUANG received the M.S degree in computer application and technology from Nanchang University, in 2009, and the PH.D. degree in computer software and theory from Wuhan University, in 2018. He is currently an Assistant Professor with the School of Computer and

Information Engineering, Jiangxi Normal University, China. His research interests include information security, software engineering and knowledge graph.



Zishuai Li is a second-year graduate student in the School of Computer and Information Engineering, Jiangxi Normal University, China. His research interests are software engineering, humancomputer interaction and knowledge graph.



Zhenchang Xing is a Senior Research Scientist with Data61, CSIRO, Eveleigh, NSW, Australia. In addition, he is an Associate Professor in the Research School of Computer Science, Australian National University. Previously, he was an Assistant Professor in the School of

Computer Science and Engineering, Nanyang Technological University, Singapore, from 2012-2016. His main research areas are software engineering, applied data analytics, and human-computer interaction.



ZHENGKANG ZUO received the Ph.D. degree in computer science and technology from the Chinese Academy of Sciences (CAS), Beijing, China. He is currently a Associate Professor and deputy director of the Computer Science and Technology Department of Jiangxi Normal University, Nanchang, China. His main research interests include software

formal methods, generic programming, etc.



Xin Peng is a professor with the School of Computer Science, Fudan University, China. His research interests include data-driven intelligent software development, cloud-native software and AIOps, and software engineering for AI and cyber-physical-social systems. He was the recipient of the ICSM 2011 Best Paper Award, the ACM SIGSOFT Dis-

tinguished Paper Award at ASE 2018, the IEEE TCSE Distinguished Paper Awards at ICSME 2018,2019,2020, and IEEE Transactions on Software Engineering 2018 Best Paper Award.



Xiwei Xu is a Senior Research Scientist with Architecture& Analytics Platforms Team, Data61, CSIRO. She is also a Conjoint Lecturer with UNSW. She started working on blockchain since 2015. She is doing research on blockchain from software architecture perspective, for example, tradeoff analysis, and decision making and evaluation framework. Her main research interest is software ar-

chitecture. She also does research in the areas of service computing, business process, and cloud computing and dependability.



Qinghua Lu is a Senior Research Scientist with Data61, CSIRO, Eveleigh, NSW, Australia. Before joining Data61, she was an Associate professor at China University of Petroleum, and she worked as a researcher at National Information and Communications Technology Australia. She has published more than 100 aca-

demic papers in international journals and conferences. Her research interests include the software architecture, blockchain, software engineering for AI, and AI ethics.