Appendix

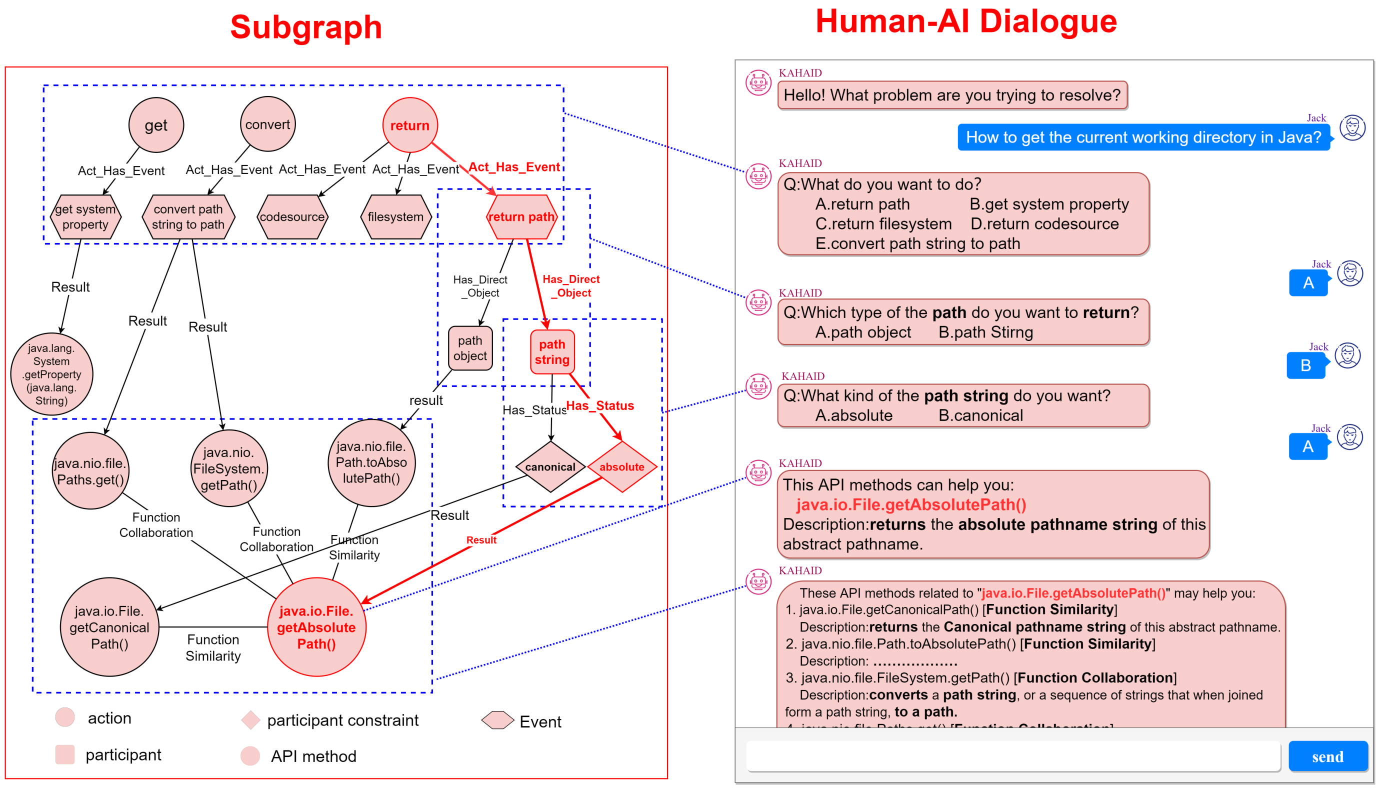
In this document you can view the following content:

- MOTIVATING EXAMPLE
- Knowledge Extraction
- Semantic Roles Annotation Details
- Syntactic Roles Annotation Details
- Template that generated according to the Aspects
- RQ: What is the Accuracy of Entity and Relation Extraction for Constructing API Behavior KG?
- User Study

I. MOTIVATING EXAMPLE

A. 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.



KAHAID
Hello! What problem are you trying to resolve?

Jack
How to get the current working directory in Java?

KAHAID
Q:What do you want to do?
A.return path B.get system property
C.return filesystem D.return codesource
E.convert path string to path

Jack
A

KAHAID
Q:Which type of the **path** do you want to **return**?
A.path object B.path Stirng

Jack
B

KAHAID
Q:What kind of the **path string** do you want?
A.absolute B.canonical

Jack
A

KAHAID
This API methods can help you:
java.io.File.getAbsolutePath()
Description:**returns** the **absolute pathname string** of this abstract pathname.

KAHAID
These API methods related to "java.io.File.getAbsolutePath()" may help you:
1. java.io.File.getCanonicalPath() [**Function Similarity**]
Description:**returns** the **Canonical pathname string** of this abstract pathname.
2. java.nio.file.Path.toAbsolutePath() [**Function Similarity**]
Description: ....
3. java.nio.file.FileSystem.getPath() [**Function Collaboration**]
Description:**converts** a **path string**, or a sequence of strings that when joined form a path string, **to a path.**

Fig. 1: An Example of API Search with the help of Query-Related API Behavior Subgraph and Human-AI Dialogue

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 [1] (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.

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 *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 *getCanonicalPath()* is only ranked seventh.

B. 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 Java[1]" 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".

This question receives very diverse answers, including not only directly relevant APIs like *java.io.File.getAbsolutePath()* and *java.io.File.getCanonicalPath()* but also many extended APIs, for example *java.nio.File.toAbsolutePath()* which offers similar functionality to *java.io.File.getAbsolutePath()*, and *java.nio.File.Paths.get()* (a cooperative API) which can converts the path string returned by *java.io.File.getAbsolutePath()* to a path object. Meanwhile, these answers and comments on Stack Overflow also provide explanation of the recommmended 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.

C. 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 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 [1]) 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()*.

In addition to *getAbsolutionPath()*, based on the API semantic relations in the knowledge graph, KAHAID also locates some extended APIs, including two functionally similar APIs (*java.io.File.getCanonicalPath()* and *java.nio.file.Path.toAbsolutePath()*) and two functionally cooperative APIs (*java.nio.file.Paths.get()* and *java.nio.file.FileSystem.getPath()*). It presents all the four APIs to Jack. 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.

Now, Jack get to know not only some Java IO APIs but also Java NIO APIs he does not know before, and how these APIs are related. He has many options to satisfy his need. He can directly use *java.io.file.getAbsolutePath()*. Or he may decide to use *java.io.file.getAbsolutePath()* to get a path string first, and then use *java.nio.file.Paths.get()* or *java.nio.file.FileSystem.getPath()*

[1]https://stackoverflow.com/questions/4871051/

to convert the path string to a path object, which is more convenient for later processing. Seeing *java.io.file.getCanonicalPath()* may make him realize an alternative to absolute path.

In fact, Jack has been made aware of these options even before he sees the recommended APIs during the human-AI dialogue process. For example, in the second round of dialogue, the option path object may make Jack realize a better option than accessing the path as a string. If he replies "path object" in that round, KAHAID will recommend *java.nio.file.Path.toAbsolutePath()* and other related APIs. Being exposed to the concept of path object and relevant Java NIO APIs would be serendipitous which cannot be supported by current API search methods. Furthermore, the dialogue process may help Jack determine what he really wants. For example, seeing the options in the clarification questions, Jack may realize that his initial question is very vague. He may rewrite the initial question more specifically with the knowledge obtained from the options. This knowledge-inspired query rewriting would be more effective than the query expansion based on mechanical word co-occurrence. More specific query allows KAHAID to locate the relevant APIs with fewer round of dialogue. Unlike current API search which cannot find extended APIs for specific query, KAHAID can recommend extended APIs for specific query based on the rich API relations in the knowledge graph.

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.

II. Knowledge extraction

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

A. 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[2], 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 part-of-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.

B. 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.

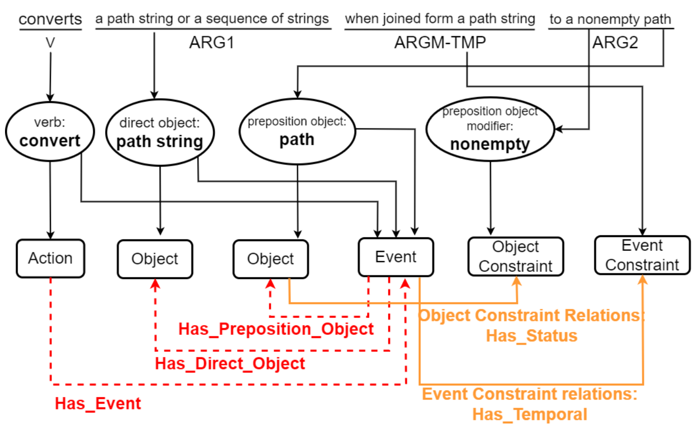
(6) The Object Constraint entity is the grammatical part with the syntactic role "direct object's modifier" or "preposition object's modifier"; its object modifier determines the type of object constraint relation which it forms with the Object entity. If the object's modifier is an adjective (ADJ), 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. We store these functional relations in the API entity's FUNCTION attribute to provide users with multifaceted clarifications during question answering because they can be used to characterize the API's functionality from fourteen aspects.

C. 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.[3], 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


converts
a path string or a sequence of strings
when joined form a path string
to a nonempty path
V
ARG1
ARGM-TMP
ARG2
verb: convert
direct object: path string
preposition object: path
preposition object modifier: nonempty
Action
Object
Object
Event
Object Constraint
Event Constraint
Has_Preposition_Object
Has_Direct_Object
Has_Event
Object Constraint Relations: Has_Status
Event Constraint relations: Has_Temporal

Note: The red text and lines indicate the event functional relations and the yellow text and lines indicate the constraint functional relations.

Fig. 2: An Example of Extracting Entities and Relations

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()"*, *"InputStream.read()"*.

(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, *doFinal()*⟩, 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.doFinal()*⟩.

Fig. 2 shows a practical example of how to extract entities and relations in three steps.

Step 1: Given an API "*java.nio.file.Paths.get(String, String ...)*" and its description "convert a path string or a sequence of strings when joined form a path string to a nonempty path.", we annotate it with semantic roles using AllenNLP, and obtain the functional parts with functional semantic roles and the constraint part with the constraint semantic role. The former includes "convert" with "V", "a path string or a sequence of strings" with "ARG1", "to a nonempty path" with "ARG2"; the latter refers to "when joined form a path string" with "ARGM-TMP". As for the former, they are assembled into a functional statement in the correct order ("convert a path string or a sequence of strings to a nonempty path"), and it is part-of-speech tagged to yield grammatical parts with six different syntactic roles, namely, "convert" with "verb", "path string" with "direct object", "path" with "preposition object", and "nonempty" with "preposition object's modifier".

Step 2: We organize these grammatical parts and constraint parts into entities and functional relations: The API entity is "*java.nio.file.Paths.get(String, String ...)*". The Event entity is "convert a path string to a path" in the form of "verb + preposition + preposition object". The Action entity is "convert". Two Object entities are "path string" and "path". Along with the formation of four entities (API, Event, Action, Object), four event relations (API Has Event, Act Has Event, Has Direct Object, Has Preposition Object) are formed. The Object Constraint entity is "nonempty", and since its preposition object's modifier is adjectives (ADJ), the object constraint relation is the Has Status relation. The Event Constraint entity is "when joined form a path string", and since its semantic role is "ARGM-TMP", the event constraint relation is the Has Temporal relation.

Step 3: As for the API semantic relations of the API entity "*java.nio.file.Paths.get(String, String ...)*", we obtain the relations *Function Similarity, Function Collaborate, Function reverse* by matching "*java.nio.file.Paths.get*" with the triples in the API-Task Knowledge graph.

III. RQ: What is the Accuracy of Entity and Relation Extraction for Constructing API Behavior KG?

Because we rely on the knowledge graph's functional relations to generate clarification questions during the user dialogue, our evaluation of KG quality focuses on the accuracy of the extracted functional relations.

In this work, we do not assess the accuracy of the extracted entities separately for two reasons. First, relations, not entities, play an important role in generating conversations. Second, entities are contained within relations. The entity is correct if the relation is correct. Furthermore, we only assess the accuracy of the 15 types of extracted functional relations rather than the 7 types of extracted semantic relations.

IV. Semantic roles annotation details

The semantic roles annotation is based on the propbank[4] standard to annotate five types of functional semantic roles (V, ARG1, ARG2, ARG3, ARG4) and nine types of constraint semantic roles (ARG-LOC, ARG-DIR, ARG-DIR, ARG-MNR, ARG-EXT, ARG-TMP, ARG-GOL, ARG-PRP, ARG-PRD, ARG-ADV). For human annotation, we show each annotator with a detailed annotation guideline complete with definitions and examples. Here we provide some brief explanations for each semantic role annotation:

- *V*: The V role is the first action in a API description.
- *ARG1*: The Arg1 role is typically assigned to the patient argument, i.e. the argument that changes state or is affected by the V.
- *ARG2*: The ARG2 role is the instrument, benefactive and attribute of the V in a API description.
- *ARG3*: The ARG3 role is the starting point of the V.
- *ARG4*: The ARG4 role is the ending point of the V.
- *ARGM-LOC*: ARGM-LOC is the Locative modifiers of the V that indicate where an action take place. The concept of a Locative is not limited to physical locations; abstract locations are also marked as ARGM-LOC. For example, "in this applet context" is a ARGM-LOC role int the description"finds all the keys of the streams *in this applet context*."
- *ARGM-DIR*: ARGM-DIR is the directional modifiers that show motion along some path. It includes phrases that begin with a direction word and clauses that are led by a direction word. For example, "up one focus traversal cycle" is a ARGM-DIR role in the description "moves the focus *up one focus traversal cycle*."
- *ARGM-MNR*: ARGM-MNR specify how the V is performed. It takes the form of phrases and clauses that begin with the words "using", "via", and "by". For example, "via this source data line" is a ARGM-MNR role in the description "writes audio data to the mixer *via this source data line*."
- *ARGM-EXT*: ARGM-EXT indicates the amount of change occurring from an action, and are used mostly for the following: 1) numerical adjuncts like "(raised prices) by 15%", 2) quantifiers such as "a lot", "fully" and "partially". For example, "partially" is a ARGM-EXT role in the description "*partially* resolves a name."
- *ARGM-TMP*: ARGM-TMP indicates when an action accurred, such as "in 1987", "last Wednesday", "soon" or "immediately". This role also includes duration adverbs (for a year/in an year) and time adverbial clauses introduced by "when" and "util". For example, in the description "repaints the component *when the image has changed*", the clause "when the image has changed" should be annotated to ARGM-TMP.
- *ARGM-GOL*: This role is the goal of the V. It includes the final destination of the V, or modifiers that indicate that the action was done for something. For example, "for the editor" shoule be annotated to a ARGM-GOL role in the description "fetches the command list *for the editor*."
- *ARGM-PRP*: ARGM-PRP includes purpose clauses, which are used to demonstrate the motivation for some action. Clauses beginning with "to", "in order to" and "so that" are canonical purpose clauses. For example, "so that its resources can be reclaimed" is a ARGM-PRP role in the description "destroys the orb *so that its resources can be reclaimed*."
- *ARGM-PRD*: ARGM-PRD specifies the true result of V. For example, in a description "gets a representation of the current choice *as a string*", "as a string" is a ARGM-PRD role. It specifies that the true result of "gets a representation of the current choice" is "a string".
- *ARGM-ADV*: ARG-ADV includes the conditional adverbial clauses that modify the V. For example, in a description "returns the window object representing the full-screen window *if the device is in full-screen mode*.", "if the device is in full-screen mode" is a ARGM-ADV role.

Annotators are then asked to annotate semantic roles from 378 API descriptions using the explanations provided. It is worth noting that annotators should annotate at least one semantic role of the V and one semantic role of the ARG1 for each API description. Other functional semantic roles (ARG2, ARG3, ARG4) can only have one of each type annotated in an API description, whereas constraint semantic roles can have multiple.

V. syntactic roles annotation details

For each API description, we sequentially combine the annotated functional semantic roles annotated into a functional description sentence. For example, for the description "*finds all the keys of the streams* in this applet context", it annotated three functional semantic roles of a V (finds), a ARG1 (all the keys) and a ARG2 (of the streams). The these three semantic roles are then combined to form a functional description sentence, "finds all the keys of the streams."

Annotators are asked to annotate six types of syntactic roles (verb, direct object, direct object modify, preposition, preposition object, preposition object modify) from all functional description sentences. The following are explanations of each syntactic role for annotators:

- *verb*: The first predicate in a functional description sentence.
- *direct object*: The noun phrase which is the (accusative) object of the verb.

- *direct object modifier*: Any word that modifies the direct object, including adjectives, quantifiers, adverbs, nouns, and Java built-in data types (byte, int/integer, float, char, boolean, double, long, shrot). These words come before the direct object and after the verb.
- *preposition*: The first preposition follows the direct object.
- *preposition object*: The noun phrase which is the (accusative) preposition object of the verb.
- *preposition object modifier*: Any word that modifies the preposition object. These words come before the preposition object and after the preposition. It has the same form as direct object modifier.

Annotators should keep the following two points in mind when annotating syntactic roles: 1) Annotators should annotate each functional description with a unique verb and direct object. 2) In a functional description, preposition and preposition object can only have one per type, whereas direct object modify and preposition object modify can have multiple.

VI. Template that generated according to the aspects

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}?

VII. RQ: What is the Accuracy of Entity and Relation Extraction for Constructing API Behavior KG?

A. Method

Functional relations are extracted from API descriptions. Given the inability to manually label 30,200 API descriptions, we use the sampling method [5] to select the minimum number (MIN) API descriptions which could ensure that the estimated population is within a certain confidence interval at a certain confidence level. This MIN can be calculated using Eq.1, where n_0 depends on the confidence level chosen and the desired error margin (see Eq.2), where z is the z-score of the confidence level and e is the error margin. The error margin e is 0.05 and the z-score is 1.96 when the confidence level is at 95%. With a population size of 30,200, the MIN number of API description instances randomly sampled is calculated to be 378.

$$MIN = n_0/(1 + (n_0 - 1)/populationsize) \tag{1}$$

$$n_0 = (z^2 * 0.25)/e^2 \tag{2}$$

We use the KG construction method described in Section II to extract 2,510 functional relations for these 378 API descriptions, which are stored in the set $S = \{s_i | 1 \leq i \leq 2510\}$, where s_i is one of the 2,510 extracted functional relations.

We invite two annotators (two master students familiar with Java who are unaffiliated with this work) to label the accuracy of the selected samples independently. As defined in Section II, a functional relation is a triplet of head-entity, relation-type, and tail-entity. A functional relation is correct only if the head-entity, relation-type, and tail-entity are all correct. Following the KG construction method mentioned in Section II, we provide annotators with definitions and examples of 6 types of entities and 15 types of functional relations so that they can label the accuracy of the extracted relations. In addition, we use Cohen's kappa [6] to assess the inter-rater agreement. If a sample is labeled differently than the others, a third annotator is assigned to give an additional label to resolve the conflict using a majority-win strategy. Based on the final labels, we compute the data accuracy. Note that we do not evaluate the 7 types of semantic relations because they are from the API-Task KG [3], which Yuan et al. have built and verified. Specifically, only if the head-entity and tail-entity of a semantic relation in the API-Task KG can be fully matched with two entities in our KG do we acknowledge and establish this semantic relation.

B. Result Analysis

The KG constructed is of high quality, with an average accuracy value of 0.909 for extracting all functional relations as shown in Table I. These functional relations are classified into three types: event relations, object constraint relations, and event constraint relations.

TABLE I: Accuracy of Extracting Functional Relations and Cohen's Kappa Agreement between the Two Annotators.

Types	**Functional Relations**	**Accuracy**	Agreement
event relations	**API Has Event**	0.918	0.919
	Act Has Event	0.918	0.926
	Has Direct Object	0.940	0.956
	Has Preposition Object	0.921	0.915
object constraint relations	**Has Status**	0.898	0.923
	Has Type	0.848	0.898
event constraint relations	**Has Location**	0.920	0.836
	Has Manner	0.907	0.893
	Has Goal	0.946	0.779
	Has Purpose	0.914	0.829
	Has Result	0.947	0.914
	Has Condition	0.931	0.915
	Has Direction	0.851	0.850
	Has Extent	0.714	0.756
	Has Temporal	0.848	0.868
	Average	0.909	0.878

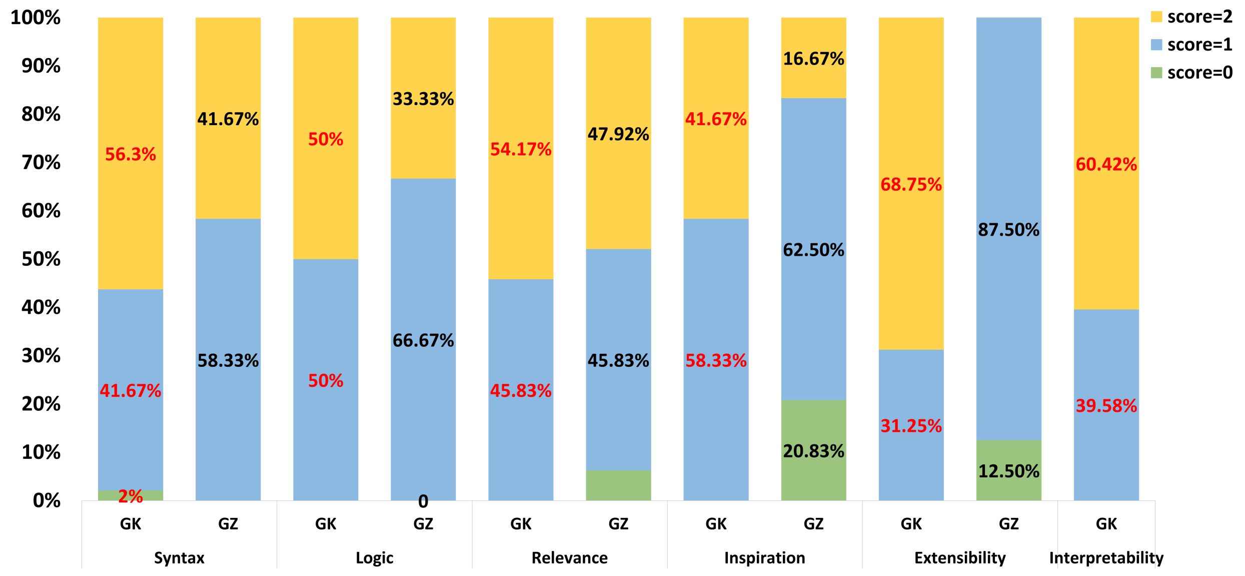
Note: Green-colored relations have an accuracy greater than 0.9; gray-colored relations have an accuracy less than 0.9; and all other types of relations have no color.

The accuracy for extracting event relations are all greater than 0.9. All agreement rates are above 0.915, which indicated almost perfect agreement. This shows that a large number of APIs can be correctly associated with the event, encouraging users to obtain the corresponding API after clarifying the event. For example, given a correct relation ⟨java.nio.file.Paths.get(), Has Event, "convert path string to path"⟩, if the user selects the option "convert path string to path", he will obtain the API *java.nio.file.Paths.get()* directly.

The accuracy for extracting object constraint relations are less than 0.9 but greater than 0.84. The Cohen's kappa between the two annotators is 0.923 and 0.898 for "Has Status" relation and "Has Type" relation, respectively, which indicated almost perfect agreement. It shows that the majority of the extracted object constraint relations in KG are correct, but 9% of them were extracted incorrectly, and 8% were not extracted at alla. For example, for the description "returns the pixel bounds of this glyphvector", the Has Type relation extracted by our KG construction method between "bounds of this glyphvector" and "pixel" is incorrect, where 'pixel" is not an object constraint. In addition, our KG construction method fails to extract the true Has Statue relation between the object constraint "this" and the object "glyphvector" because it misidentifies "bounds of this glyphvector" as an object rather than "glyphvector" as an object.

The accuracy values for extracting event constraint relations in the green squares are greater than 0.9, whereas they are less than 0.9 for extracting event constraint relations in the gray squares. All agreement rates are above 0.756 (at least substantial agreement). The event constraint relations in the green squares are extracted from descriptions that include obvious keywords that our KG construction method can easily recognize. For example, the description "fetches the command list *for the editor*" has a keyword "for". By identifying this keyword, our KG construction method can recognize that the constraint "for the editor" is the goal of the event "fetches the command list", and build the Has Goal relation between them.

On the other hand, the event constraint relations in the gray squares differ from those in the green squares. In particular, extracting Has Extent relations has the lowest accuracy of 0.714, where the number of such relations is small, only seven in total. This is because the semantic roles for these relations appear at the beginning of sentences as an adjective or adverb, which the KG construction method frequently ignores. For example, there is a keyword "fully" at the beginning of the sentence "fully parses the text producing an object of the specified type.". The keyword "fully" before the verb "parse" will be ignored because our KG construction method usually focuses on the verb "parse" and the content after the verb. As a result, KG construction method fails to extract the true Has Extent relation between the event constraint "fully" and the event "parses the text". However, the Precision of the extracted Has Extent relation is 0.857, indicating that the majority of extracted Has Event relations are correct.



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

Our KG construction method is efficient and extracts a large number of functional relations correctly, including extracted event relations, object constraint relations, and event constraint relations, resulting in a high KG quality.

VIII. USER STUDY

TABLE II: Eight Real Queries for User Study

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

A. Study Design

1) Test Set: 8 questions and their ground-truth API methods are chosen at random from the test set for the user study, as shown in Table II.

2) Participants: We recruited 16 participants from both university and IT companies. Nine of them (3 undergraduate and 6 graduate students) are from the first author's university, and seven 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 0.1 year to 4 years, with an average of 1.7 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.

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.

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.

B. *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.

C. *Indicator Analysis*

We collected each group member's rating results on different indicator, and plotted them in Fig 3. 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 Q2: "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.

References

[1] Qiao Huang, Xin Xia, Zhenchang Xing, D. Lo, and Xinyu Wang. Api method recommendation without worrying about the task-api knowledge gap. *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 293–304, 2018.

[2] Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew E. Peters, Michael Schmitz, and Luke Zettlemoyer. Allennlp: A deep semantic natural language processing platform. *ArXiv*, abs/1803.07640, 2018.

[3] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Zhengkang Zuo, Changjing Wang, and Xin Xia. 1+1¿2: Programming know-what and know-how knowledge fusion, semantic enrichment and coherent application. *ArXiv*, abs/2207.05560, 2022.

[4] Claire Bonial, Olga Babko-Malaya, Jinho D. Choi, and Jena D. Hwang. Propbank annotation guidelines. 2010.

[5] Monika Eisenhower. Elements of survey sampling. 2016.

[6] J Richard Landis and Gary G. Koch. An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers. *Biometrics*, 33 2:363–74, 1977.