KG4CraSolver: Recommending Crash Solutions via Knowledge Graph

Xueying Du*
Fudan University
Shanghai, China

Yiling Lou*
Fudan University
Shanghai, China

Mingwei Liu*†
Fudan University
Shanghai, China

Xin Peng*
Fudan University
Shanghai, China

Tianyong Yang*
Fudan University
Shanghai, China

ABSTRACT

Fixing crashes is challenging, and developers often discuss their encountered crashes and refer to similar crashes and solutions on online Q&A forums (e.g., Stack Overflow). However, a crash often involves very complex contexts, which includes different contextual elements, e.g., purposes, environments, code, and crash traces. Existing crash solution recommendation or general solution recommendation techniques only use an incomplete context or treat the entire context as pure texts to search relevant solutions for a given crash, resulting in inaccurate recommendation results.

In this work, we propose a novel crash solution knowledge graph (KG) to summarize the complete crash context and its solution with a graph-structured representation. To construct the crash solution KG automatically, we propose to leverage prompt learning to construct the KG from SO threads with a small set of labeled data. Based on the constructed KG, we further propose a novel KG-based crash solution recommendation technique KG4CraSolver, which precisely finds the relevant SO thread for an encountered crash by finely analyzing and matching the complete crash context based on the crash solution KG. The evaluation results show that the constructed KG is of high quality and KG4CraSolver outperforms baselines in terms of all metrics (e.g., 13.4%-113.4% MRR improvements). Moreover, we perform a user study and find that KG4CraSolver helps participants find crash solutions 34.4% faster and 63.3% more accurately.

CCS CONCEPTS

• **Software and its engineering** → **Maintaining software**; **Software evolution**; **Software testing and debugging**.

KEYWORDS

Crash Solution Recommendation, Knowledge Graph, Stack Overflow

*X. Du, Y. Lou, M. Liu, X. Peng, and T. Yang are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China.
†M. Liu is the corresponding author (liumingwei@fudan.edu.cn).

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616317

ACM Reference Format:
Xueying Du, Yiling Lou, Mingwei Liu, Xin Peng, and Tianyong Yang. 2023. KG4CraSolver: Recommending Crash Solutions via Knowledge Graph. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616317

1 INTRODUCTION

Software crashes have been widely recognized as a type of serious bug and should be fixed in a high priority. Resolving crashes is challenging and time-consuming [22, 23, 66], and developers often turn to online Q&A forums for help (e.g., Stack Overflow) by discussing their encountered crashes or referring to other similar crashes and solutions. As shown by previous work [44], crash bugs have been actively discussed on Stack Overflow (SO), i.e., 7% out of 2.65 million Java-related SO threads are about crashes, which provide a large amount of crash solution knowledge for developers.

However, describing and understanding a crash often involve very complex contexts. In addition to the code and the reported crash traces, precisely diagnosing the root cause of a crash also relies on the environment that the project is configured with, the purpose of the developers writing the code, and the symptom of the crash. Therefore, developers often write very lengthy posts to describe their encountered crashes. Based on our statistics, the average length of the Java exception-related SO threads is around 320 words. As a result, it takes developers a lot of time to search and read many relevant SO threads, among which they further find the one that shares the most similar crash context as theirs and then fix their own crash based on its solution.

To alleviate the manual efforts in navigating through so many lengthy SO threads and to help developers quickly find the solution for their encountered crash, researchers have proposed to automatically recommend crash solutions by finding the SO thread that discusses the same/similar crash context as the encountered one [23, 44, 45, 66]. Existing crash solution recommendation work identifies relevant SO threads for a given crash by only searching with the code or the crash trace, which have not considered other important elements in contexts (e.g., environment, symptom, or the purposes of the project). However, these contextual elements are also essential for developers to diagnose the crash. For example, two crashes with the same code snippets and the same crash traces could be caused by different reasons if they involve different environments (e.g., using different versions of the library). As a

result, using such an incomplete context to identify relevant SO threads could be inaccurate. In addition, for existing techniques that recommend solutions for general software engineering problems by mining online Q&A forums [20, 73], they actually have limited effectiveness in crash solution recommendation, since they concatenate different crash contextual elements (e.g., code and natural language descriptions) into a long textual query and such a purely text matching would also lead to inaccurate recommendation results.

To address these limitations, we propose a novel crash solution knowledge graph (KG), which summarizes the complete crash context and its solution with a graph-structured representation. In our crash solution KG, the nodes represent different elements in the crash context while the edges represent the relationships between elements. To construct the crash solution KG automatically, we propose to leverage prompt learning to construct the KG from SO threads with a small set of labeled data. Based on the constructed KG, we further propose a novel KG-based crash solution recommendation technique KG4CraSolver, which precisely finds the relevant SO thread for an encountered crash by finely analyzing and matching the complete crash context based on the crash solution KG. The benefits of recommending crash solutions based on the KG are as follows. First, compared to existing crash solution recommendation techniques that only use code or crash trace in the crash context, our crash solution KG is able to represent a complete crash context with all the different elements. Second, compared to existing general solution recommendation techniques that simply concatenate different contextual elements into a long textual query, our crash solution KG represents the crash context in a more structured way by representing different elements and their relationships with nodes and edges. Therefore, compared to these techniques, our structured and comprehensive representation of crash contexts enables fine-grained and precise context matching between a given crash context and candidate SO threads. In addition, with such a KG, KG4CraSolver could further explain each recommended solution with matching details (e.g., the matching scores in different elements) and summarize the solution in a more concise way, which increases the usability and readability of the recommended solution for developers.

We construct a crash solution KG with 963,334 nodes and 1,626,101 edges for 245 common Java exceptions from 71,592 SO threads and further implement KG4CraSolver as an automatic tool. We first evaluate the effectiveness of our KG construction, and find that each key step achieves high precision. For example, our construction approach classifies different sentences (e.g., symptoms or reasons) in the crash context with 91.6% precision and 91.2% recall, and it further extracts fine-grained phrases (e.g., environment) from classified sentences with 0.855 BLEU and 0.843 EM (Exact Match). We then evaluate the effectiveness of solution recommendation on a newly-constructed benchmark of 855 crash bugs from SO duplicate question records. The results show that KG4CraSolver outperforms five baselines on the MRR (Mean reciprocal rank) and Hit@10. Furthermore, to evaluate the practical usefulness of KG4CraSolver for developers, we then conduct a user study by asking 10 participants to find solutions for crash bugs with KG4CraSolver. The results show that compared to using baselines, the participants could find solutions more accurately (+63.3%) and faster (+34.4%) with the help of KG4CraSolver. Moreover, we survey the participants and their feedback shows that they consider the solution summary generated by KG4CraSolver as complete, concise, easy to read, and useful.

In summary, this paper makes the following contributions.

- **A new knowledge graph for crash solutions** that summarizes the complete crash contexts and solutions in a structured and comprehensive way;
- **A novel approach for crash solution KG construction** that leverages prompt learning to automatically construct the crash solution KG with a very small set of labeled data;
- **A novel crash solution recommendation technique KG4CraSolver** that analyzes the complete context of an encountered crash and finds the relevant SO thread in a fine-grained manner based on the crash solution KG;
- **An extensive evaluation** that demonstrates the effectiveness of our KG construction and crash solution recommendation, and also shows the practical usefulness of KG4CraSolver with a user study.

2 CRASH SOLUTION KG DEFINITION

In this section, we introduce how we define our crash solution KG. In particular, we first perform a pilot study on a small dataset of SO threads to understand what kind of information is commonly included in crash-related SO discussions; and then we design our KG for crash solution based on the results.

Pilot study. We first collect a dataset of crash-related SO threads. A thread includes a question with the corresponding answers. In particular, from SO data dumps [2], we randomly sample 100 threads that (i) are related to crash solution with one specific exception type in their question titles or tags, and (ii) have an accepted answer.

Similar to previous work [37], we annotate all information units in the threads (i.e., the title, tags, and all sentences in the question and answer) based on what kind of information it could provide to describe the crash context and solutions (i.e., crash descriptive elements and solution descriptive elements). Our annotation follows an open coding procedure [27], which involves three of the authors in discussion. We start with three codes from previous work [37], i.e., erroneous implementation, error type, and error occasion. During the annotation process, if none of the existing codes is applicable, we create new codes or refine the names and definitions of existing codes after discussions.

In this way, we summarize nine elements that cover the essential crash descriptive information and solution descriptive information in crash-related threads. Table 1 further shows the detailed definitions and examples of each element. An element of a text type is described by natural language, and an element of a non-text type is described in structured domain-specific language (e.g., code). In particular, *Purpose*, *Symptom*, *Environment*, *Erroneous Code*, *Crash Trace*, and *Exception Type* are elements extracted from the question in SO threads, and the others are extracted from the answer.

KG schema. Based on these elements, we further design the KG for crash solutions. Fig. 1 shows the conceptual schema of our KG. In particular, the KG consists of two parts: the crash scenario knowledge (i.e., the orange part) and crash solution knowledge (i.e., the blue part), with the elements extracted from the question and answer, respectively. In terms of relation connections, the elements

Table 1: Definitions of Elements in Crash Solution KG

Name	Type	Definition	Example
Purpose	Text	Describe the task that the developer wants to complete, serving as the background of the crash bug.	I am trying to implement a chat application in Java using UDP for multiple clients.
Symptom	Text	Describe the situation when the crash occurs, which includes the operation before the crash occurs, program running status, and unexpected program outputs.	When I run antlr TestParser.g4 && javac *.java the parser code gets generated and compiled. When I run grun TestParser testRule -gui I get the error.
Environment	Text	Describe the environment where the crash bug occurs, such as programming languages, operating systems, and libraries.	java, spring, spring-security, spring-boot, spring boot version 1.3.7.RELEASE
Reason	Text	Describe the root cause of the crash.	In your class User, you don't have a name property.
Solution Step	Text	Describe the solution steps of the crash.	The problem is solved by just undeploying and redeploying the respective portlet in liferay.
Erroneous Code	Non-text	Buggy code snippets that trigger the crash.	public class Cloning { Cloning c=new Cloning(); ...
Solution Code	Non-text	Correct code snippets that fixes the crash.	static Cloning c=new Cloning();
Crash Trace	Non-text	Error messages and stack traces that are reported with the crash.	Exception in thread "main" java.sql.SQLException: Can't create table 'Sensors_db.one' (errno: 121) at com.mysql.jdbc.SQLError...
Exception Type	Non-text	Exception type of the crash.	NullPointerException

related to the crash scenario are centered around the concept node [Crash Bug], and the elements related to the crash solution are all connected to the concept node [Solution]. The "solved by" relation connects [Crash Bug] to [Solution]. Moreover, the sentences describing the causes of the exception [Reason] or the sentences describing the symptoms of the exception [Symptom] are connected by "succession" relations; similarly, the sentences describing the [Solution Step] are connected by "followed by" relations.

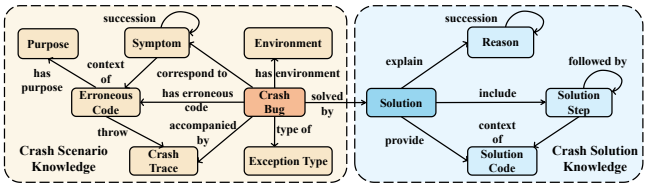

Figure 1: Conceptual Schema of Crash Solution KG

3 CRASH SOLUTION KG CONSTRUCTION

In this section, we propose an automated approach to construct the crash solution KG from SO threads. The key challenge in constructing such a KG from the massive online corpus is to precisely identify different elements in each SO thread. Existing sentence classification and phrase extraction techniques are all learning-based approaches and require a large amount of labeled data for model training. However, in our task, there is no high-quality and ready-made labeled data, and it is time-consuming to manually annotate such a large training dataset. To this end, we propose a few-shot learning-based approach for crash solution KG construction, which leverages prompt learning to precisely identify different elements in SO threads with a small set of labeled data. Fig. 2 shows the overview of our KG construction approach.

Step 1 (Section 3.2): Given a large number of SO threads, we first identify high-quality crash-related threads, which serve as the input of the KG construction.

Step 2 (Section 3.3): We then leverage template-based rules to identify non-text elements (i.e., crash trace, erroneous code, and solution code) from the question and the answer of each thread.

Step 3 (Section 3.4): We then identify *crash descriptive sentence* from the question paragraphs and identify *solution descriptive sentence* from the answer paragraphs by utilizing prompt learning for sentence classification.

Step 4 (Section 3.5): We further extract crash descriptive phrases (i.e., purposes and environments) from crash descriptive sentences identified in Step 3.

After all the elements have been extracted from SO threads, we add them to the crash solution KG and establish the relevant relationships according to the schema in Fig. 1.

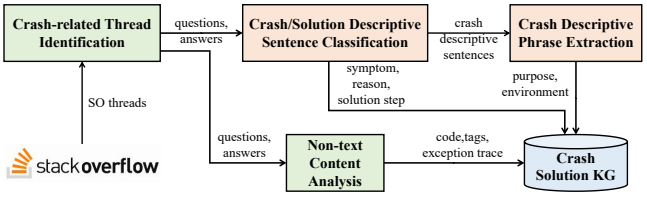

Figure 2: Overview of Crash Solution KG Construction

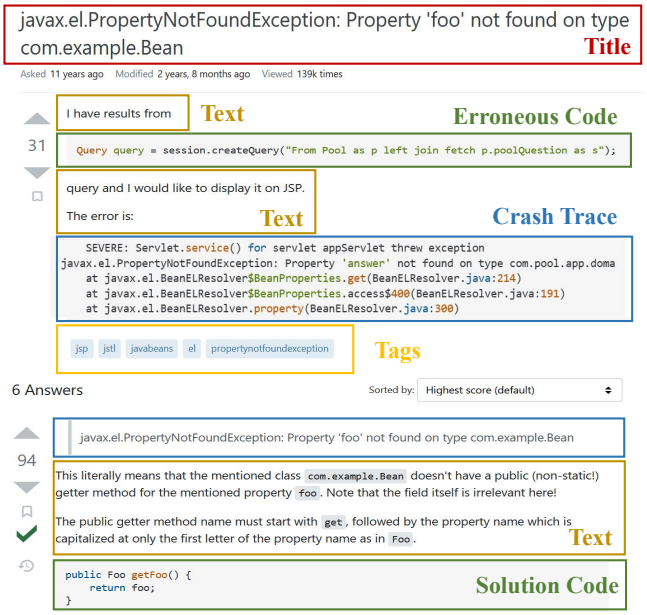

Figure 3: Example of A Crash-related Thread

3.1 Running Example

In this section, we illustrate how we construct the KG with a running example. Fig. 3 shows a SO thread [16], which discusses how to

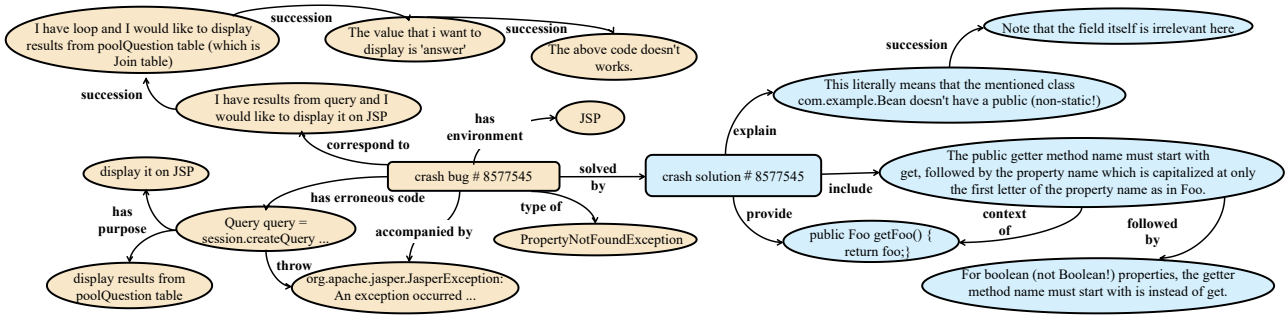

Figure 4: An Example of Crash Solution KG

fix a PropertyNotFoundException exception. The accepted answer shows that this exception occurs as the program accesses a property without public getter methods. Fig. 4 shows part of the crash solution KG constructed for this example, where orange ellipses, and blue ellipses denote crash descriptive elements, and solution descriptive elements, respectively.

After identifying the example in Fig. 3 as a crash-related thread in Step 1, we then extract the non-text descriptive elements from the thread in Step 2. For example, the crash trace *"org.apache.jasper.JasperException: An exception occurred processing JSP page /WEB-INF/views/home.jsp at line 21"*, the erroneous code *"<c:forEach items ="$pools" var="pool">..."*, and the solution code *"public Foo getFoo() {return foo;}"* are extracted as non-text descriptive elements in Fig. 4. In Step 3, the descriptive sentences are classified into the symptoms (e.g., *"The above code doesn't works."*), reasons (e.g., *"Note that the field itself is irrelevant here..."*), and solution steps (e.g., *"The public getter method name must start with get..."*) with our prompt learning-based sentence classification. In Step 4, the key phrases in the purpose (e.g., *"display it on JSP"*) and environments (e.g., *"JSP"*) are further extracted from crash descriptive sentences with our learning-based phrase extraction. In this way, we extract fine-grained crash scenario knowledge and crash solution knowledge to construct the KG, which then can be used to support KG-based crash solution recommendation (in Section 4).

3.2 Crash-related Thread Identification

This step extracts high-quality crash-related threads from SO data dumps [2], as the input for constructing the crash solution KG. In this work, we focus on crash bugs in Java programs given their prevalence [44, 66]. With the following criteria, we select the threads that: (1) have "java" in the title or tags, (2) have "exception" or "error" in the title or tags, (3) have an accepted answer, (4) have a positive vote for its question, and (5) contain at least one specific exception type (e.g., *NullPointerException*) in the given exception type list. To build a pool of common Java exception types, we systematically parse 35,773 Java libraries from Maven Central [5] according to the Libraries.io dataset [4] and JDK 1.8 [13] and extract the names of all classes that are a subclass of *java.lang.Exception* or *java.lang.Error*. To guarantee the quality of the solutions, we only keep the accepted answers in threads. We further group the crash-related threads of the same exception type together.

3.3 Non-text Content Analysis

For a crash-related thread, we first extract its non-textual elements (i.e., crash traces, erroneous code, and solution code) from the non-textual content. First, we use BeautifulSoup [3] to parse the question and the answer from HTML format to clean text and extract the non-text content wrapped by *<pre><code></code></pre>* or *<blockquote></blockquote>*. We then classify the extracted content into one of three categories, namely code snippets, crash traces, and others. The classification is based on the list of regular expressions proposed by Liu et al. [37] in their previous work on SO content analysis. The code snippets identified from the question are erroneous code, while the code snippets identified from the answer are solution code as shown in Figure 3. From the identified crash traces, we further identify the exception message by matching with regular expressions. For example, from the crash trace shown in Figure 3, we identify the exception message *"javax.el.PropertyNotFoundException: Property 'answer' not found on type com.pool.app.domain.Pool"*.

We replace the recognized non-text content with a placeholder *-CODE-* and then split the question and the answer into sentences using spaCy [7] for the following sentence classification (Section 3.4). Where necessary, a "." was added after *-CODE-* to ensure that the following sentence splitting is correct.

3.4 Crash/Solution Descriptive Sentence Classification

In this step, we identify crash/solution descriptive sentences from the remaining textual content in the thread via a prompt-based text classification model. We then introduce the definition of the task (Section 3.4.1), the design of the prompt-based learning model used (Section 3.4.2), and the concrete implementations (Section 3.4.3), respectively.

3.4.1 Task Definition. For sentences in crash-related threads, we classify them into one or more of the following five categories (i.e., *Purpose*, *Symptom*, *Reason*, *Solution Step*, and *Others*). The definitions for the first four categories are shown in Table 1, and the sentences (e.g., *"Thanks."* or *"I was finally able to solve this."*) without any concrete information on crashes and solutions are categorized as *Others*. In particular, a long sentence might contain information of multiple categories. Therefore, we formulate our sentence identification problem as multiple binary classification tasks, which separately train a binary text classifier for each of the four categories (except *Others*) and yield binary outputs $\boldsymbol{y}$ of positive or

negative. Specifically, we apply the text classifiers on sentences in the question to identify *Purpose* and *Symptom* categories and apply the text classifiers on sentences in the answer to identify *Reason* and *Solution Step* categories. Sentences that do not fall into any of these four categories would be regarded as in *Others* category and would be filtered out then.

3.4.2 Model Design. Fine-tuning a pre-trained language model (PLM) for downstream tasks (e.g., text classification [25, 34, 71], machine translation [60], named-entity detection [57]) has achieved great success and been widely adopted in various domains [32, 65]. However, to achieve good results on downstream tasks, it still requires enough labeled data to fine-tune the PLMs in a supervised way.

Prompt-based learning is a new paradigm in the NLP field for using knowledge in PLMs [42]. Its idea is to narrow the gap between the downstream tasks and the pre-training task by converting the training objective of downstream tasks into a similar form as the pre-training stage, i.e., the MLM objective [54]. As shown in Fig. 5, a natural language prompt (i.e., *[X] I think it [Z] the reason for this the problem.*) is added to the input sentence to make the input format identical to the pre-training stage and then the PLM will predict the mask token as the pre-training task. Prompt-based learning methods generally require much less training data compared to traditional supervised learning and have been widely used for few-shot learning or even zero-shot learning [24]. Therefore, we build our text classifiers based on prompt-based learning, which only requires a small set of labeled data for fine-tuning.

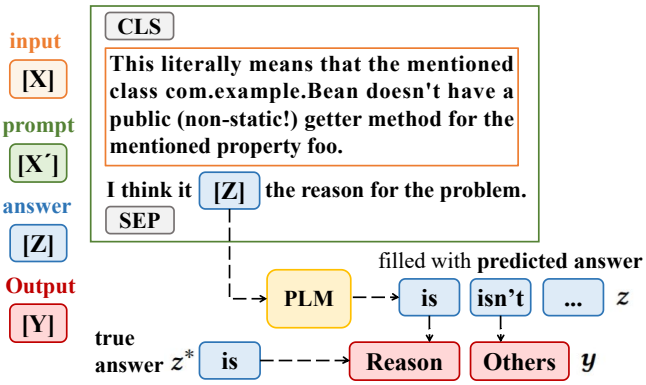

Figure 5: An Example of Prompt-based Text Classification

We then detail how our prompt-based sentence classifiers work with the example in Fig. 5. The original input $\boldsymbol{x}$ is modified with a template into a textual string prompt $\boldsymbol{x}'$ that has some unfilled slots, and then the PLM probabilistically fills the unfilled information to obtain a predicted answer $\boldsymbol{z}$, from which the final output $\boldsymbol{y}$ can be derived. In Fig. 5, the input $\boldsymbol{x}$ is the sentence to be classified, and the output $\boldsymbol{y}$ is whether the sentence is *Reason* or not. The prompt function $f_{\text{prompt}}(\boldsymbol{x})$ is a function that converts the input into a specific form by inserting the input $\boldsymbol{x}$ and adding a slot $[Z]$ where answer $\boldsymbol{z}$ may be predicted and filled in by the pre-trained model. The predicted answer $\boldsymbol{z}$ will later be mapped into $\boldsymbol{y}$, corresponding to different class labels. For example, when the predicted answer is a word expressing affirmation, e.g., "is", the sentence $\boldsymbol{x}$ will be classified as *Reason*, and when the predicted answer is a word expressing negation, e.g., "isn't," the sentence will be classified as *Others*.

For each category of classifier, we define the corresponding $f_{\text{prompt}}(\boldsymbol{x})$ to modify the input text x into a prompt, i.e., $\boldsymbol{x}' = f_{\text{prompt}}(\boldsymbol{x})$. It contains three parts: [X] for inserting the input x, a textual string for the prompt, and a slot [Z] for the PLM to fill later.

- *Purpose*: [X] I think that [Z] is the reason for this problem.
- *Symptom*: [X] I think this [Z] the situation.
- *Reason*: [X] I think it [Z] the reason for this the problem.
- *Solution Step*: [X] I think it [Z] a solution for exception repair.

We map each sentence class label we defined to a set of label words or phrases. The true answer $\boldsymbol{z}^*$ is defined as "is" for the positive class label and "is not" or "isn't" for the negative class label. For the example in Fig. 5, the true answer for the prompt is "is", and the sentence is classified into *Reason*.

3.4.3 Model Implementation and Dataset Construction. We implement the prompt-based text classifier by using OpenPrompt [6], an open-Source framework for prompt learning with two thousand stars on GitHub. We build our approach based on the BERT base model (uncased) [21], one of the most representative PLMs. We fine-tune the classifiers on our training dataset with the following hyperparameters: CrossEntropyLoss as loss function, AdamW as optimizer, learning rate 0.0001, and 10 training epochs.

To construct the training dataset, we randomly select 50 Java crash-related threads (from Section 3.2) and manually annotate the sentences in these threads into the five categories. Note that a sentence may be classified into multiple categories if it contains diverse information. As a result, we obtain 30, 133, 55, 87, and 103 sentences for *Purpose*, *Symptom*, *Reason*, *Solution Step*, and *Others* respectively. We separate the labeled sentences obtained from the question (i.e., *Purpose*, *Symptom*, *Others*) and those obtained from the answer (i.e., *Reason*, *Solution Step*, *Others*). For each classifier, its positive samples consist of annotated data for the corresponding category, while the negative samples consist of all remaining sentences in the question or answer. The whole labeling process takes about 4 man-hours. Such manual costs are much less than traditional supervised learning and fine-tuning methods, e.g., previous work takes 175 man-hours to label 2,278 SO threads for the SOSum dataset construction [28].

3.5 Crash Descriptive Phrase Extraction

In the last step, we extract the crash descriptive sentences (i.e., *Purpose* and *Symptom*) from the question. In fact, some crash descriptive sentences contain fine-grained descriptive phrases that summarize the purpose (verb-object phrase) and environment (noun phrase) of a crash. For example, the phrases *"JSP"* and *"display it on JSP"* in the sentence *"I have results from query and I would like to display it on JSP"* summarize the environment and purpose, respectively. Compared with the whole sentence, these descriptive phrases often contain fewer noise words, which is beneficial for more accurate matching in the subsequent solution recommendation. Therefore, in this step, we further extract crash descriptive phrases (i.e., purposes and environments) from the crash descriptive sentences. We then introduce the task definition (Section 3.5.1), the design of the

EQA-based phrase extraction model (Section 3.5.2), and the concrete implementations (Section 3.5.3), respectively.

3.5.1 Task Definition. In this step, we focus on extracting the descriptive phrases that are related to purposes and environment from the crash descriptive sentences. We currently do not extract phrases for other elements such as symptoms and reasons, given their complexity and diversity in expression. In summary, the task in this step is to extract continuous spans that represent purposes and environment from a given sentence.

3.5.2 Model Design. Our phrase extraction task is actually a named entity recognition problem, which is often tackled by a sequence tagging model based on a PLM model (such as BERT) [35, 41, 55]. However, it requires a large amount of labeled data to train a sequence tagging model since the model needs to predict the tag for each token in the sentence [17]. Our idea is to convert the phrase extraction task into an extractive question-answering task, which aims finding an answer (i.e., a segment of text, or span) in a given context paragraph for a specific question [61]. In this way, the extractive question answering task could be done by designing appropriate questions as prompts for pre-trained models. Some researchers have designed a BERT-based EQA model [18, 49], which combines the question and the context paragraphs as the input and predicts the start token and end token of the answer span with a confidence score as the output. These EQA models have been pre-trained in large-scale question-answering datasets, e.g., Stanford Question Answering Dataset (SQuAD) [51] with 107,785 question-answer pairs on 536 articles, which have good performance and generalization in extractive question answering. As shown in Figure 6, based on the trained EQA model, we combine the crash descriptive sentence (as the context) and some prompts designed specifically for the descriptive phrases (as the question) as the input and take the output of the EQA model as the phrase extraction results. In the example illustrated in Figure 4, we performed phrase extraction on the sentence *"I have loop and I would like to display results from poolQuestion table (which is Join table)"*, yielding the extracted purpose phrase *"display results from poolQuestion table"*. Note that the extraction result of an EQA model is the special token [CLS] (the start of a sentence) representing that the model cannot find any answer span from the given context. In this way, we can obtain more flexible extraction results without a large amount of annotated training data.

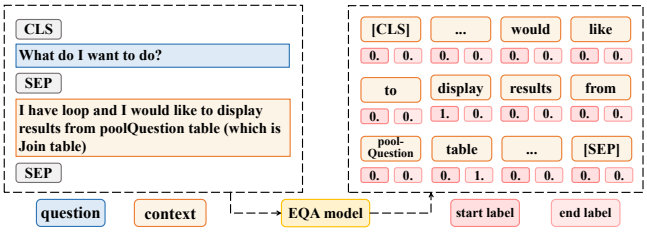

Figure 6: An Example of EQA-based Phrase Extraction

In particular, we design the following question prompts.

- *Environment*:
  - What are the tools and their versions?
  - What are the environment, using tools and their versions?
  - What are the environment?
  - What are the using tools?
- *Purpose*:
  - What am I trying to do?
  - What do I want to do?
  - What is the purpose of the project?
  - What is the goal of the project?

To ensure the quality of extracted phrases, we filter out: (1) the phrases with low confidence score (i.e., less than 0.1); (2) the phrases without alphabetical characters; (3) the environment phrases with more than five words; and (4) the phrases with the placeholder token *-CODE-* or the words "Exception", "Error", or "Code".

3.5.3 Model Implementation and Dataset Construction. To achieve better performance, we propose to fine-tune the EQA model with a few data samples (i.e., few-shot learning) instead of directly using EQA model with the designed question prompts (i.e., zero-shot learning). To this end, we build a training dataset for the phrase extraction task by further annotating the sentence classification dataset (constructed in Section 3.4.3). Each sample should include three aspects of information: the context (original sentence to be extracted), the question prompts, and the extracted answer span. To enhance the data and allow the model to fully learn different question prompts, we apply different question prompts to the same sentence to get different training data. As a result, the dataset includes 120 and 616 samples for purpose phrases and environment phrases, respectively.

We implement the phrase extraction model based on Transformer [12], a Python library providing APIs to easily download and to train state-of-the-art pre-trained models. We choose the fine-tuned checkpoint of the pre-trained DistilBERT (i.e., distilbert-base-uncased-distilled-squad [1]) as the base EQA model, which has been trained on SQuAD dataset [51]. Based on it, we use the labeled samples to further fine-tune it to obtain the final phrase extraction model. To fine-tune the model, we use the Adam optimizer with a learning rate of 0.0001 for a total of 10 training epochs.

4 KG-BASED CRASH SOLUTION RECOMMENDATION

Based on the crash solution KG, we further design a crash solution recommendation approach KG4CraSolver, which precisely identifies the relevant solutions for a given crash by analyzing and matching the complete crash context. Fig. 7 presents the overview of KG4CraSolver, which includes four steps, i.e., crash descriptive element extraction, candidate solution retrieval, candidate solution re-ranking, and solution summary. Given a crash and its corresponding detailed crash description as input, KG4CraSolver first extracts each crash descriptive element (e.g., purposes, environments, and crash traces) in the same way as the KG construction; Second, for each crash descriptive element, KG4CraSolver leverages BM25-based lexical similarity or BERT-based semantic similarity strategies to retrieve candidate solutions; Third, KG4CraSolver further re-ranks candidate solutions based on the similarity scores of all crash descriptive elements. Lastly, KG4CraSolver generates

a comprehensive summary of the recommended solutions by including the solution, crash description, and other matching details.

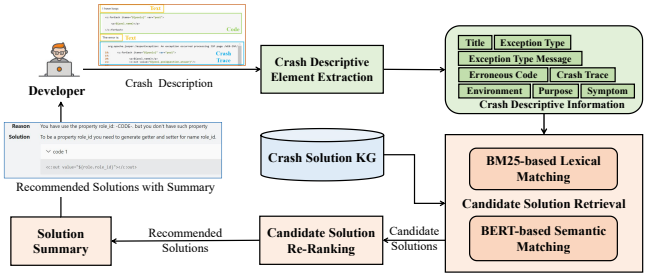

Figure 7: Overview of KG4CraSolver

4.1 Crash Descriptive Element Extraction

Given a crash and its crash description (e.g., example in Fig. 3) as input, this step extracts all the crash descriptive elements via the same method as Section 3.3 to Section 3.5. In particular, we extract the following crash descriptive elements, including exception type, title, purpose, symptom, environment, erroneous code, exception type message, and crash trace, which are then used in the subsequent candidate solution retrieval and re-ranking.

4.2 Candidate Solution Retrieval

In this step, KG4CraSolver leverages each extracted crash descriptive element to retrieve relevant crashes from our constructed KG and regards their solutions as candidate solutions. First, KG4CraSolver selects out all the crashes that have the same exception type as the given crash. Among these crashes, we then retrieve the top-k ($k = 30$ in our implementation) crashes with the highest similarity with each crash descriptive element, respectively. We then consider the solutions of all these retrieved crashes as our candidate solutions. As for the similarity calculation, we leverage two strategies for different crash descriptive elements given their different characteristics, i.e., BM25-based lexical matching for environments, erroneous code, crash trace, title, exception type message, and BERT-based semantic matching for symptom and purpose. We then detail the two matching methods in Section 4.2.1 and 4.2.2.

4.2.1 BM25-based Lexical Matching. BM25 [53] has been extensively used in many search engines, and it is an efficient information retrieval method that mainly matches based on keywords. The code, crash trace, and exception type message are often in a structured format, and environments and title are also short phrases or sentences with concise keywords. Therefore, BM25-based lexical matching is inherently more suitable for these crash descriptive elements. Specifically, the BM25 similarity score between the query q and the d is computed as Equation 1, where $f(w_i, q)$ is the word w_i's term frequency in query q, $IDF(w_i)$ is the inverse document frequency of word w_i. k and b are two free parameters, which are used to normalize the range of term frequencies and control the influence of document length.

$$Sim_{BM25}(q,d) = \sum_{i=1}^{n} \frac{\mathrm{IDF}(\mathrm{w_i}) \times \mathrm{f}(\mathrm{w_i}, \mathrm{q}) \times (\mathrm{k}+1)}{\mathrm{f}(\mathrm{w_i}, \mathrm{q}) + \mathrm{k} \times \left(1\text{-}\mathrm{b} + \mathrm{b} \times \frac{|\mathrm{q}|}{\mathrm{avgdl}}\right)} \quad (1)$$

We implement BM25-based lexical matching based on Elasticsearch [10], a search engine based on the Lucene library using BM25 as the default score function. Each element is cleaned with standard preprocessing procedures (i.e., tokenization, lemmatization, and stop word removal) before matching.

4.2.2 BERT-based Semantic Matching. Recently, BERT-based semantic matching methods have become popular, which can preserve the semantic-related sequential information [68]. The method includes a bi-encoder-based retrieval step and a cross-encoder-based re-ranking step. In the bi-encoder-based retrieval step, it uses a BERT-based sentence embedding model to map the query and the documents to a high-dimensional vector space, where sentences with similar semantics are close, and then select candidate documents based on the similarities of their vector representations. In the cross-encoder-based re-ranking step, it uses a BERT-based sentence relevance prediction model to predict the relevance between the candidate documents and the query and re-ranks the candidates based on their relevance. There may be a large lexical gap in the expression of purposes and symptoms with similar semantics, such as *"display the result"* and *"show the answers"*. Therefore, BERT-based semantic matching is more suitable for these descriptive elements.

We implement the BERT-based semantic matching based on Haystack [11], which is an open-source framework for building search systems based on novel NLP models (e.g., BERT). We use the base DistillBERT (uncased) [56] in bi-encoder-based retrieval step and cross-encoder-based re-ranking step, and fine-tune them with 21,172 duplicate question title pairs from SO data dumps [2].

4.3 Candidate Solution Re-ranking

We re-rank candidate solutions by combining the matching scores of all crash descriptive elements. For the input crash description c, we calculate its final matching score with each candidate solution s by summing up the weighted matching scores of all crash descriptive elements according to Equation 2. E denotes the set of all the crash descriptive elements (i.e., title ttl, purpose pur, symptom sym, environment env, erroneous code ec, exception type message em, and crash trace ct). In Equation 3, $RankScore$ normalizes the similarity scores of s on each descriptive element t based on its ranking i.e., $Rank_t(c,s)$. α is a constant hyper-parameter set to 3.

$$Score(c,s) = \sum_{t \in E} W_t \times RankScore_t(c,s) \quad (2)$$

$$RankScore_t(c,s) = 100 - \alpha \times (Rank_t(c,s) - 1) \quad (3)$$

As different descriptive elements could have different importance during matching (e.g., environment and exception type message), we introduce weights W_t in Equation 2. To avoid overfitting, we tune these weights with an hyperparameters optimization framework optuna [14] on a small validation set constructed by all SO duplicate question pairs (up to 50 pairs) of 10 randomly-selected exception types. The reranking of all candidate posts is performed according to the final score as shown in Equation 2, and the MR metric of the correct solution served as the objective function for tuning. Each hyper-parameter is constrained within a range of (0,1) with a step size of 0.01. $W_{ttl} = 0.98$, $W_{pur} = 0.20$, $W_{sym} = 0.29$, $W_{env} = 0.95$, $W_{ec} = 0.58$, $W_{em} = 0.48$, and $W_{ct} = 0.16$. Based on the $Score(c,s)$ of each candidate solutions, we return the top-n (n is set to 10

in our implementation) candidates with the highest scores as the recommended solutions.

4.4 Solution Summary

For each recommended solution, KG4CraSolver generates a comprehensive summary of the recommended solution by including the solution, crash description, and the matching details, to help developers better understand the recommended solution and quickly judge the relevance of the recommended solution to the given crash description. Fig. 8 shows an example of the final summary of the recommended solution. In particular, the resolution summary includes basic information of a relevant post (i.e., post ID, post title, and extra information such as the view count), crash descriptive elements (i.e., the environment, exception type message, purpose, and symptom), solutions (i.e., the reason and solution steps), and the matching degree (e.g., "Environment Score: 91") that shows the detailed matching scores of each crash descriptive element during solution retrieval. In this way, developers could have a clear picture of the crash context and solutions, as well as the relevant crash descriptive elements that match the given crash.

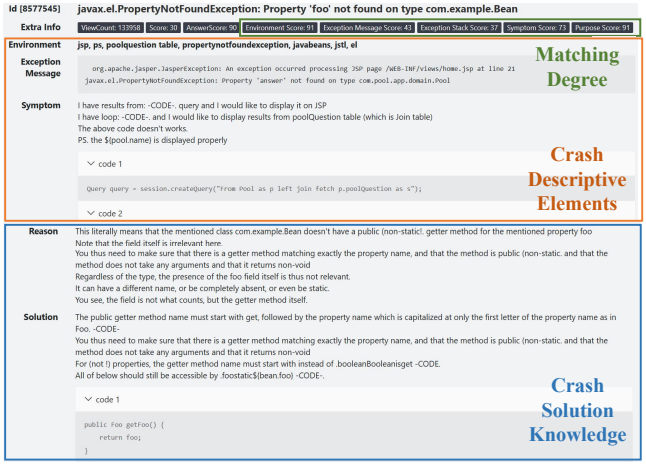

Figure 8: An Example of the Solution Summary

5 EVALUATION

We implement KG4CraSolver based on the crash solution KG constructed from 71,592 sampled crash-related threads across 245 types of exceptions. The resulting KG consists of 963,334 nodes and 1,626,101 edges, including 70,474 *Purpose*, 321,535 *Symptom*, 107,082 *Reason*, 143,528 *Solution*, and 130,191 *Environment*. We then extensively evaluate the effectiveness of KG construction, the effectiveness of crash solution recommendation, and the practical usefulness of recommended solutions for developers, by answering the following research questions.

RQ1 (Effectiveness of KG construction): What is the intrinsic quality of the critical steps in the KG construction?

RQ2 (Effectiveness of solution recommendation): How effective is KG4CraSolver in crash solution recommendation?

RQ3 (Usefulness of recommended solution): How useful of KG4CraSolver in helping developers solve crash bugs?

5.1 RQ1: Effectiveness of KG Construction

In this RQ, we evaluate the effectiveness of two major steps in KG construction, i.e., crash/solution descriptive sentence classification (Step 3) and crash descriptive phrase extraction (Step 4) in Section 5.1.1 and 5.1.2, respectively.

5.1.1 Crash/Solution Descriptive Sentence Classification. We first introduce the benchmark, baseline, and metrics used in this evaluation, and then present the results.

Baseline. Our sentence classifier leverages prompt learning to fine-tune BERT with a small set of training samples. To investigate the contribution of our prompt design, we compare our classifier with a baseline that directly fine-tunes BERT with the same dataset as ours (mentioned in Section 3.4.3). In particular, the baseline appends an additional Softmax layer after BERT, which is a common practice for building a classifier on BERT [26].

Benchmark. We manually construct a benchmark of 100 crash-related threads as the testing set for evaluation. In particular, we first randomly sample 100 crash-related threads that are not overlapped with the training dataset, and then involve two MS students with 2 years Java development experience to annotate the sentences in these threads.

Metrics. We use four commonly used evaluation metrics in sentence classification tasks, i.e., accuracy, precision, recall, and F1. Accuracy is the proportion of correctly classified sentences; precision is the proportion of true positive predictions among all positive predictions; recall is the proportion of true positive predictions among all instances; and F1-score is the harmonic mean of the precision and recall, which balances both values.

Results. As shown in Table 2, our sentence classifier outperforms the baseline in all metrics, i.e., 6.7%, 9.2%, 7.8%, and 8.6% improvements in terms of accuracy, precision, recall, and F1-score, respectively. The results demonstrate the effectiveness of our prompt design and also indicate the superiority of prompt learning over traditional fine-tuning when the size of training data is small.

Table 2: Effectiveness on Sentence Classification

Sentence Type	KG4CraSolver				Baseline			
	Acc.	F1	Prec.	Recall	Acc.	F1	Prec.	Recall
Purpose	0.971	0.947	0.939	0.956	0.898	0.826	0.801	0.861
Symptom	0.975	0.971	0.978	0.965	0.910	0.894	0.898	0.891
Reason	0.883	0.874	0.886	0.867	0.798	0.776	0.783	0.770
Solution	0.860	0.860	0.862	0.861	0.814	0.813	0.815	0.812
Average	0.922	0.913	0.916	0.912	0.855	0.827	0.824	0.834

5.1.2 Crash Descriptive Phrase Extraction. We first introduce the benchmark, baseline, and metrics used in this evaluation, and then discuss the results.

Baseline. Similarly, we adopt a BERT-based sequence tagging model as the baseline to study the effectiveness of our EQA model. In particular, the baseline fine-tunes BERT with an additional Softmax layer, which predicts the tag for each token in the input sequence. In our task, we use the BIO tag schema and include the following 5 tags, i.e., B-Purpose, I-Purpose, B-Environment, I-Environment, O, which represent the start token of a purpose phrase, the inside token of a purpose phrase, the start token of an environment phrase, the inside token of an environment phrase, and not an entity token, respectively. By interpreting the tagging results, the baseline is able

to extract phrases in sentences. The baseline is fine-tuned with the same training dataset as ours (mentioned in Section 3.5.3).

Benchmark. We manually construct a benchmark of 100 crash-related threads as the testing set for evaluation. In particular, we first randomly sample 100 crash-related threads that are not overlapped with the training dataset, and then involve two MS students with 2 years Java development experience to annotate the phrases of purpose and environment in the sentences of the purpose and symptom categories.

Metrics. We use two commonly-used evaluation metrics in information extraction tasks, i.e., BLEU (Bilingual Evaluation Understudy) [48] and EM (Exact Match). BLEU calculates the n-gram precision between the predicted and the reference answers; and EM is a binary classification metric that calculates the percentage of exact matches between the predicted and the reference answers.

Results. As shown in Table 3, our phrase extraction model substantially outperforms the baseline in both BLEU and EM metrics, with 45.8% and 44.2% improvements, respectively. The poor performance of the baseline indicates that it heavily relies on a large training dataset, whereas our prompt designs help better utilize the pre-trained model when the number of training samples is small.

Table 3: Effectiveness on Crash Descriptive Phrase Extraction

Phrase Type	KG4CraSolver		Baseline	
	BLEU	EM	BLEU	EM
Purpose	0.860	0.827	0.573	0.564
Environment	0.850	0.840	0.221	0.219
Average	0.855	0.834	0.397	0.392

5.1.3 Summary. The results demonstrate the effectiveness of two major steps in our KG construction, i.e., crash/solution descriptive sentence classification and crash descriptive phrase extraction. Additionally, the prompt learning is helpful in our scenario when there is only a small number of training samples.

5.2 RQ2: Effectiveness of Solution Recommendation

In this RQ, we evaluated the effectiveness of KG4CraSolver in recommending crash solutions.

5.2.1 Benchmark. SO data dumps [2] mark the duplicate relationships between questions and some of them are between crash-related questions, e.g., *"Stackoverflow error in class constructor"* [8] and *"Why am I getting a StackOverflowError exception in my constructor"* [9] are duplicate questions. To construct the benchmark, we first randomly select 50 exception types involved in our crash solution KG and then collect pairs of crash-related questions that 1) belong to these selected exception types; and 2) have duplicate relationships. For each exception type, we select at most 50 pairs of duplicate questions, leading to 855 pairs of duplicate questions. In this way, our benchmark can cover crash bugs related to diverse exception types, e.g., *SQLException* and *NoSuchMethodException*. For each pair of duplicate questions, we use one of it as the query expressing the crash scenario and the accepted answer of the other one as the ground truth for the crash solution recommendation.

5.2.2 Baselines. We compare KG4CraSolver with baselines of two categories, i.e., existing crash solution recommendation techniques, and existing retrieval methods that recommend solutions for general questions. In particular, for the former, we include the state-of-the-art technique CraSolver [66]; for the latter, we include two representative retrieval methods (AnswerBot [73] and CLEAR [68]), which represent word-embedding-based and sentence-embedding-based information retrieval methods, respectively.

- **CraSolver** [66]. A crash solution recommendation method that uses BM25 to retrieve relevant questions from SO by matching the input crash trace with crash traces contained in the questions.
- **AnswerBot** [73]. AnswerBot includes a module for relevant question retrieval, which combines the Word2Vec model [47] and IDF metric to measure the relevance between the input query and the questions in the corpus.
- **CLEAR** [68]. CLEAR is an automated API recommendation approach with a BERT-based relevant question retrieval step and re-ranking step.

We enhance these baselines by limiting their candidates within crash-related threads that contain the same exception type as the query, which is consistent with our approach (Section 4.2) for a fair comparison.

5.2.3 Metrics. Following previous work [73], we use the widely-used information retrieval metrics, i.e., MRR (Mean Reciprocal Rank) and Hit@k ($k = 1, 5, 10$) for evaluation. MRR calculates the average ranking of the correct solution in the ranked list, and Hit@k computes the ratio of queries that the correct solution ranked with Top-N positions in the ranked list to the total queries (i.e., 855 queries). For each query, we focus on the Top-100 results in the ranked list returned by each technique.

5.2.4 Results. Table 4 presents the effectiveness of KG4CraSolver and all baselines. AnswerBot-full or AnswerBot-title denotes the baseline AnswerBot that takes the complete crash description (body + title) or the summarized description (only title) as inputs. Same as it is for CLEAR-full and CLEAR-title. Overall, KG4CraSolver outperforms all the baselines on all metrics by achieving 13.4%-113.4%, 4.0%-77.6%, 5.0%-149.6%, and 26.9%-160.3% improvements in terms of MRR, Hit@1, Hit@5, and Hit@10, respectively. The results indicate that the fine-grained matching of different descriptive elements in KG4CraSolver is indeed more effective than the whole text matching in baselines (e.g., KG4CraSolver vs. AnswerBot-full and KG4CraSolver vs. CLEAR-full). Moreover, the gap between CLEAR-full and CLEAR-title and the gap between KG4CraSolver and CraSolver further demonstrate the superiority of using a complete crash context over using a partial context.

Table 4: Effectiveness on Solution Recommendation

Approaches	MRR	H@1	H@5	H@10
CraSolver	0.095	0.058	0.115	0.156
AnswerBot-full	0.151	0.084	0.199	0.264
AnswerBot-title	0.179	0.099	0.240	0.320
CLEAR-full	0.162	0.085	0.218	0.283
CLEAR-title	0.146	0.078	0.198	0.278
KG4CraSolver	**0.203**	**0.103**	**0.287**	**0.406**

5.2.5 Summary. The results show that KG4CraSolver substantially outperforms solution recommendation baselines, indicating the effectiveness of our KG-based solution recommendation approach and our fine-grained utilization of complete crash contexts.

5.3 RQ3: User Study on Usefulness

In this RQ, we conduct a user study to evaluate the practical usefulness of KG4CraSolver in helping developers solve crash bugs.

5.3.1 Study Design. The details of our study design are as follows.

Participants. We invite 10 Master students with 1-4 years Java programming experience for this user study. We conduct a pre-experiment survey on their Java programming experience and divide them into two roughly equal participant groups (G_A and G_B) based on the survey.

Crash Bugs. We select 12 questions describing crash bugs from the benchmark constructed in RQ2 (Section 5.2). We classify the exception types into three categories based on the number of identified crash-related threads (Section 3.2), i.e., popular (more than 500 threads), normal (200-500 threads), and unpopular (less than 200 threads). To ensure the diversity of selected crash bugs, we sample four questions for each category respectively and all questions selected belong to different exception types. We randomly divide 12 crash bugs into two equal groups (T_A and T_B), each with six crash bugs (two crash bugs for each exception type category).

Procedure. The task in this user study is to ask participants to find solutions (i.e., SO threads) for a given crash bug with the aid of KG4CraSolver or with the baseline (i.e., using the default SO search engine). In particular, the participants in G_A are assigned to solve crashes in T_A with KG4CraSolver and to solve crashes in T_B with the baseline; the participants in G_B are assigned to solve crashes in T_B with KG4CraSolver and to solve crashes in T_A with the baseline. For each task, we provide the original SO question (including title and question body) as the input crash context for the participants. When participants are working on their tasks with the baseline, they are allowed to search with any keywords on SO with unlimited trials until they find solutions that they consider as correct. When the participants are working on their tasks with KG4CraSolver, they make the decision based on only the top-10 recommendation SO threads returned by KG4CraSolver. Each task has a time limit of 10 minutes, and if the task is not completed within the time limit, an empty solution will be submitted. For each task, we record the relevant threads they submit, their completion time, and the time used in finding the first relevant thread.

After all tasks are finished, we further conduct a survey to collect user feedback. Participants are asked to evaluate KG4CraSolver in terms of readability, conciseness, completeness, and usefulness on a 4-points Likert scale [30](1-disagree; 2-somewhat disagree; 3-somewhat agree; 4-agree) by the following statements:

- **Readability**. KG4CraSolver can provide a well-organized and easy-to-understand solution summary.
- **Conciseness**. KG4CraSolver can provide a solution summary containing little redundant information.
- **Completeness**. KG4CraSolver can provide a solution summary containing all necessary information for solving crash bugs.
- **Usefulness**. KG4CraSolver is useful in helping participants solve crash bugs.

5.3.2 Results. For the 12 tasks, the participants submit 55 non-empty results (4 empty results with KG4CraSolver and 1 empty result with the baseline). We then invite two extra participants (who are not involved in previous experiments) with more than 4 years experience of in Java development to judge whether the returned threads provide the correct solution for the given crash. For each submitted thread, if it is judged differently by the two participants, a third participant is assigned to give an additional assessment to resolve the conflict by a majority-win strategy. The agreement between the judgments is a substantial agreement (i.e., Cohen's Kappa coefficient [46] of 0.678).

Table 5: Results of the User Study

	Accuracy	First Thread Time	Completion Time
Baseline	21.7%	92s	275s
KG4CraSolver	85.0%	61s	233s
Improvement	+63.3%	-31s (34.4%)	-42s (15.2%)

Table 5 presents the results of our user study. Overall, with KG4CraSolver participants can find solutions more accurately and more efficiently. In particular, compared to using baselines, participants find correct solutions for more crashes (63.3% more) within less time (31 seconds less). We further perform Welch's t-test [69] to assess the statistical significance of the differences. The *p-value* shows that all the differences are statistically significant ($p \ll 0.05$).

As for readability, completeness, conciseness, and usefulness of KG4CraSolver, all participants rate them as either 4 (agree) or 3 (somewhat agree). Specifically, with 80%, 50%, 30% and 50% of the ratings for each category respectively are 4. The results, as well as the informal feedback from participants, indicate that our solution summary is well-organized and easy to read. In particular, they consider that the environment, exception type message, and the retrieval matching scores of each descriptive element are helpful for them to quickly determine the relevance of the recommended thread. Compared to directly searching on SO, KG4CraSolver significantly improves their search efficiency and accuracy.

5.3.3 Summary. KG4CraSolver helps developers find the solutions more accurately and more efficiently and the participants consider the solution summary generated by KG4CraSolver as complete, concise, easy to read, and useful.

5.4 Threats to Validity

A major threat to the internal validity of our studies lies in the subjective judgment in human annotations in RQ3. To mitigate the threat, we follow commonly used data analysis principles such as assigning multiple annotators, conflict resolution, and reporting agreement coefficients. Additionally, the participants' professional background may also impact the results of RQ3. To mitigate this threat, we collected statistics regarding the participants' years of experience and proficiency in Java development during the recruitment process, and ensured that the average level of each group of participants was as close as possible during the grouping process. A common threat is that the baselines we used in RQ1 and RQ2 are implemented by ourselves because of no publicly available implementations. However, we carefully reproduced and tested the baselines to avoid introducing errors. A threat to the external validity is that our experiments are only for Java crash bugs. Thus, the findings of our studies may not be generalized to other programming languages in practice. However, the design of our KG is not specific to Java and the implementation could be easily extended to support libraries of other object-oriented languages.

6 RELATED WORK

Crash Solution Recommendation. Existing crash solution recommendation techniques [44, 45, 66] mainly use code or crash traces as the input and find relevant SO threads based on code matching [44, 45] or crash trace matching [66]. For example, MAESTRO [44] uses the buggy code to search relevant SO threads, and Mahajan et al. [45] further extend MAESTRO to extract the patch from the SO thread to help developers fix Java crashes. CraSolver [66] uses the structural information in the crash traces to search relevant SO threads. However, these techniques only utilize a part of the crash context (i.e., code or crash traces) to identify the relevant solution, without using the other important contextual elements (e.g., environment or symptom) for crash diagnosis. Our work is different from these techniques by analyzing the crash context in a comprehensive and structured way. To this end, we first construct a novel knowledge graph to represent different crash/solution descriptive elements and then perform fine-grained matching based on KG to enable more accurate solution recommendation.

In addition, there is a series of techniques that recommend solutions for general software engineering problems by mining online Q&A forums [19, 20, 33, 37]. For instance, AnswerBot [19] generates a query-focused multi-answer-posts summary for a given technical question, and CROKAGE [20] provides the solution for a programming task based on the natural language description. These techniques search relevant posts with pure natural language queries, while crash context contains different descriptive elements in different structures (e.g., code and natural language descriptions). Thus simply concatenating them as a long textual query would lead to an inaccurate recommendation. Our work is different from these techniques by representing the crash context in a more structured way instead of a long textual sequence, which further supports fine-grained matching rather than pure text matching.

Knowledge Graphs in the Software Engineering Domain. Researchers in the software engineering domain have constructed knowledge graphs for different kinds of knowledge to support software development tasks, such as API KG [29, 38, 39, 43, 50, 52], software development concept KG [40, 62–64, 72, 74], programming task KG [36, 59], code KG [31, 70], ML/DL model KG [41], and bug KG [58, 67]. To the best of our knowledge, our work constructs the first knowledge graph for crash solutions, based on which we further automatically recommend solutions for a given crash description. Moreover, our work is the first one that uses prompt-based learning in KG construction, requiring much less labeled data than previous work (e.g., [38, 59]).

7 CONCLUSIONS

This work proposes a novel crash solution knowledge graph (KG) to summarize the complete crash context and its solution with a graph-structured representation. We leverage prompt learning to automatically construct the KG from SO threads with a small set of labeled data. Based on the constructed KG, we further propose a novel KG-based crash solution recommendation technique KG4CraSolver by finely analyzing and matching the complete crash context based on the crash solution KG. Our results show that the constructed KG is of high quality and KG4CraSolver outperforms baselines in all metrics. Moreover, we perform a user study to show the practical usefulness of KG4CraSolver.

8 DATA AVAILABILITY

All data is included in our replication package [15].

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China (2021ZD0112903) and the National Natural Science Foundation of China under Grant No. 61972098.

REFERENCES

[1] 2019. *Distilbert-ase-uncased-distilled-squad.* https://huggingface.co/distilbert-base-uncased-distilled-squad
[2] 2021. *Stack Overflow data dump version from September 4, 2021.* Retrieved May 4, 2022 from https://archive.org/download/stackexchange/
[3] 2022. *BeautifulSoup.* Retrieved January 20, 2023 from https://www.crummy.com/software/BeautifulSoup/bs4/doc/
[4] 2022. *Libaries.io open data.* Retrieved January 20, 2023 from https://libraries.io/data
[5] 2022. *Maven Central Repository.* Retrieved January 20, 2023 from https://mvnrepository.com
[6] 2022. *Openprompt.* https://github.com/thunlp/OpenPrompt
[7] 2022. *spaCy.* Retrieved January 20, 2023 from https://spacy.io
[8] 2023. *Duplicate question "Stackoverflow error in class constructor".* https://stackoverflow.com/questions/18421891/stackoverflow-error-in-class-constructor
[9] 2023. *Duplicate question "Why am I getting a StackOverflowError exception in my constructor".* https://stackoverflow.com/questions/35844801/why-am-i-getting-a-stackoverflowerror-exception-in-my-constructor
[10] 2023. *ElasticSearch.* https://github.com/elastic/elasticsearch
[11] 2023. *Haystack.* https://github.com/deepset-ai/haystack
[12] 2023. *Hugging Face Transformer Api.* https://huggingface.co/docs/transformers/index
[13] 2023. *JDK 1.8.* https://docs.oracle.com/javase/8/docs/api/overview-summary.html/
[14] 2023. *Optuna.* https://github.com/optuna/optuna
[15] 2023. *Replication Package.* Retrieved February 2, 2023 from https://github.com/FudanSELab/KG4CraSolver
[16] 2023. *Stack Overflow Example Thread.* https://stackoverflow.com/questions/8577545/
[17] Gizem Aras, Didem Makaroglu, Seniz Demir, and Altan Cakir. 2021. An evaluation of recent neural sequence tagging models in Turkish named entity recognition. *Expert Syst. Appl.* 182 (2021), 115049. https://doi.org/10.1016/j.eswa.2021.115049
[18] Sabur Butt, Noman Ashraf, Muhammad Hammad Fahim Siddiqui, Grigori Sidorov, and Alexander F. Gelbukh. 2021. Transformer-Based Extractive Social Media Question Answering on TweetQA. *Computación y Sistemas* 25, 1 (2021). arXiv:2110.03142 https://arxiv.org/abs/2110.03142
[19] Liang Cai, Haoye Wang, Bowen Xu, Qiao Huang, Xin Xia, David Lo, and Zhenchang Xing. 2019. AnswerBot: an answer summary generation tool based on stack overflow. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019.* ACM, 1134–1138. https://doi.org/10.1145/3338906.3341186
[20] Rodrigo Fernandes Gomes da Silva, Chanchal K. Roy, Mohammad Masudur Rahman, Kevin A. Schneider, Klérisson V. R. Paixão, Carlos Eduardo de Carvalho Dantas, and Marcelo de Almeida Maia. 2020. CROKAGE: effective solution recommendation for programming tasks by leveraging crowd knowledge. *Empir. Softw. Eng.* 25, 6 (2020), 4707–4758. https://doi.org/10.1007/s10664-020-09863-2
[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2019). https://doi.org/10.18653/v1/n19-1423
[22] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou. 2011. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011.* IEEE Computer Society, 333–342. https://doi.org/10.1109/ICSM.2011.6080800
[23] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. 2015. Fixing Recurring Crash Bugs via Analyzing Q&A Sites (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015,* Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 307–318. https://doi.org/10.1109/ASE.2015.81

[24] Yuxian Gu, Xu Han, Zhiyuan Liu, and Minlie Huang. 2022. PPT: Pre-trained Prompt Tuning for Few-shot Learning. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*. Association for Computational Linguistics, 8410–8423. https://doi.org/10.18653/v1/2022.acl-long.576

[25] Jeremy Howard and Sebastian Ruder. 2018. Universal Language Model Fine-tuning for Text Classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*. Association for Computational Linguistics, 328–339. https://doi.org/10.18653/v1/P18-1031

[26] Chanwoo Jeong, Sion Jang, Eunjeong L. Park, and Sungchul Choi. 2020. A context-aware citation recommendation model with BERT and graph convolutional networks. *Scientometrics* 124, 3 (2020), 1907–1922. https://doi.org/10.1007/s11192-020-03561-y

[27] Shahedul Huq Khandkar. 2009. Open coding. *University of Calgary* 23 (2009), 2009.

[28] Bonan Kou, Yifeng Di, Muhao Chen, and Tianyi Zhang. 2022. SOSum: A Dataset of Stack Overflow Post Summaries. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 247–251. https://doi.org/10.1145/3524842.3528487

[29] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. 2018. Improving API Caveats Accessibility by Mining API Caveats Knowledge Graph. In *34th IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, September 23-29, 2018, Madrid, Spain*. IEEE Computer Society, 183–193. https://doi.org/10.1109/ICSME.2018.00028

[30] Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology* (1932).

[31] Zeqi Lin, Yanzhen Zou, Junfeng Zhao, and Bing Xie. 2017. Improving software text retrieval using conceptual knowledge in source code. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, 123–134. https://doi.org/10.1109/ASE.2017.8115625

[32] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task Learning based Pre-trained Language Model for Code Completion. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 473–485. https://doi.org/10.1145/3324884.3416591

[33] Mingwei Liu, Xin Peng, Qingtao Jiang, Andrian Marcus, Junwen Yang, and Wenyun Zhao. 2018. Searching stackoverflow questions with multi-faceted categorization. In *Proceedings of the 10th Asia-Pacific Symposium on Internetware*. ACM, 10:1–10:10. https://doi.org/10.1145/3275219.3275227

[34] Mingwei Liu, Xin Peng, Andrian Marcus, Christoph Treude, Xuefang Bai, Gang Lyu, Jiazhan Xie, and Xiaoxin Zhang. 2021. Learning-based extraction of first-order logic representations of API directives. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 491–502. https://doi.org/10.1145/3468264.3468618

[35] Mingwei Liu, Xin Peng, Andrian Marcus, Christoph Treude, Xuefang Bai, Gang Lyu, Jiazhan Xie, and Xiaoxin Zhang. 2021. Learning-based extraction of first-order logic representations of API directives. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 491–502. https://doi.org/10.1145/3468264.3468618

[36] Mingwei Liu, Xin Peng, Andrian Marcus, Christoph Treude, Jiazhan Xie, Huanjun Xu, and Yanjun Yang. 2022. How to Formulate Specific How-To Questions in Software Development?. In *30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2020, November 14-18, 2022, Virtual Event, Singapore*. ACM, 1015–1026. https://doi.org/10.1145/3540250.3549160

[37] Mingwei Liu, Xin Peng, Andrian Marcus, Shuangshuang Xing, Christoph Treude, and Chengyuan Zhao. 2022. API-Related Developer Information Needs in Stack Overflow. *IEEE Trans. Software Eng.* 48, 11 (2022), 4485–4500. https://doi.org/10.1109/TSE.2021.3120203

[38] Mingwei Liu, Xin Peng, Andrian Marcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. 2019. Generating Query-specific Class API Summaries. In *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, August 26-30, 2019, Tallinn, Estonia*. ACM, 120–130. https://doi.org/10.1145/3338906.3338971

[39] Mingwei Liu, Xin Peng, Xiujie Meng, Huanjun Xu, Shuangshuang Xing, Xin Wang, Yang Liu, and Gang Lv. 2020. Source Code based On-demand Class Documentation Generation. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 864–865. https://doi.org/10.1109/ICSME46990.2020.00114

[40] Mingwei Liu, Simin Yu, Xin Peng, Xueying Du, Tianyong Yang, Huanjun Xu, and Gaoyang Zhang. 2023. Knowledge Graph based Explainable Question Retrieval for Programming Tasks. In *39th IEEE International Conference on Software Maintenance and Evolution, ICSME 2023, Bogotá, Colombia, October 1-6, 2023*. IEEE.

[41] Mingwei Liu, Chengyuan Zhao, Xin Peng, Siming Yu, Haofen Wang, and Chaofeng Sha. 2023. Task-Oriented ML/DL Library Recommendation based on a Knowledge Graph. *IEEE Transactions on Software Engineering* (2023).

[42] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35. https://doi.org/10.1145/3560815

[43] Yang Liu, Mingwei Liu, Xin Peng, Christoph Treude, Zhenchang Xing, and Xiaoxin Zhang. 2020. Generating Concept based API Element Comparison Using a Knowledge Graph. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, September 21-25, 2020, Melbourne, Australia*. IEEE, 834–845. https://doi.org/10.1145/3324884.3416628

[44] Sonal Mahajan, Negarsadat Abolhassani, and Mukul R. Prasad. 2020. Recommending stack overflow posts for fixing runtime exceptions using failure scenario matching. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 1052–1064. https://doi.org/10.1145/3368089.3409764

[45] Sonal Mahajan and Mukul R. Prasad. 2022. Providing Real-time Assistance for Repairing Runtime Exceptions using Stack Overflow Posts. In *15th IEEE Conference on Software Testing, Verification and Validation, ICST 2022, Valencia, Spain, April 4-14, 2022*. IEEE, 196–207. https://doi.org/10.1109/ICST53961.2022.00030

[46] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia Medica: Biochemia Medica* 22, 3 (2012), 276–282.

[47] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, Vol. 26. Curran Associates, Inc., 3111–3119. https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html

[48] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 311–318. https://doi.org/10.3115/1073083.1073135

[49] Kate Pearce, Tiffany Zhan, Aneesh Komanduri, and Justin Zhan. 2021. A Comparative Study of Transformer-Based Language Models on Extractive Question Answering. *CoRR* abs/2110.03142 (2021).

[50] Xin Peng, Yifan Zhao, Mingwei Liu, Fengyi Zhang, Yang Liu, Xin Wang, and Zhenchang Xing. 2018. Automatic Generation of API Documentations for Open-Source Projects. In *IEEE Third International Workshop on Dynamic Software Documentation, DySDoc@ICSME 2018, Madrid, Spain, September 25, 2018*. IEEE, 7–8. https://doi.org/10.1109/DySDoc3.2018.00010

[51] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100, 000+ Questions for Machine Comprehension of Text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*. The Association for Computational Linguistics, 2383–2392. https://doi.org/10.18653/v1/d16-1264

[52] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2020. API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 461–472. https://doi.org/10.1145/3324884.3416551

[53] Stephen E. Robertson and Steve Walker. 1988. Some Simple Effective Approximations to the 2-Poisson Model for Probabilistic Weighted Retrieval. In *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval. Dublin, Ireland, 3-6 July 1994 (Special Issue of the SIGIR Forum)*. ACM/Springer, 232–241. https://doi.org/10.1016/0306-4573(88)90021-0

[54] Jaechul Roh, Minhao Cheng, and Yajun Fang. 2022. MSDT: Masked Language Model Scoring Defense in Text Domain. *CoRR* abs/2211.05371 (2022). https://doi.org/10.1109/UV56588.2022.10185524

[55] Dhruva Sahrawat, Debanjan Mahata, Mayank Kulkarni, Haimin Zhang, Rakesh Gosangi, Amanda Stent, Agniv Sharma, Yaman Kumar, Rajiv Ratn Shah, and Roger Zimmermann. 2019. Keyphrase Extraction from Scholarly Articles as Sequence Labeling using Contextualized Embeddings. *CoRR* abs/1910.08840 (2019). arXiv:1910.08840 http://arxiv.org/abs/1910.08840

[56] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR* abs/1910.01108. arXiv:1910.01108 http://arxiv.org/abs/1910.01108

[57] Felix Stollenwerk. 2022. Adaptive Fine-Tuning of Transformer-Based Language Models for Named Entity Recognition. *CoRR* abs/2202.02617 (2022). arXiv:2202.02617 https://arxiv.org/abs/2202.02617

[58] Yanqi Su, Zhenchang Xing, Xin Peng, Xin Xia, Chong Wang, Xiwei Xu, and Liming Zhu. 2021. Reducing bug triaging confusion by learning from mistakes with a bug tossing knowledge graph. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 191–202. https://doi.org/10.1109/ASE51524.2021.9678574

[59] Jiamou Sun, Zhenchang Xing, Rui Chu, Heilai Bai, Jinshui Wang, and Xin Peng. 2019. Know-How in Programming Tasks: From Textual Tutorials to Task-Oriented Knowledge Graph. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, September 29 - October 4, 2019, Cleveland, OH, USA*. IEEE, 257–268. https://doi.org/10.1109/ICSME.2019.00039

[60] Inigo Jauregi Unanue, Jacob Parnell, and Massimo Piccardi. 2021. BERTTune: Fine-Tuning Neural Machine Translation with BERTScore. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 2: Short Papers), Virtual Event, August 1-6, 2021*. Association for Computational Linguistics, 915–924. https://doi.org/10.18653/v1/2021.acl-short.115

[61] Stalin Varanasi, Saadullah Amin, and Guenter Neumann. 2021. AutoEQA: Auto-Encoding Questions for Extractive Question Answering. In *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*. Association for Computational Linguistics, 4706–4712. https://doi.org/10.18653/v1/2021.findings-emnlp.403

[62] Chong Wang, Xin Peng, Mingwei Liu, Zhenchang Xing, Xuefang Bai, Bing Xie, and Tuo Wang. 2019. A Learning-Based Approach for Automatic Construction of Domain Glossary from Source Code and Documentation. In *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, August 26-30, 2019, Tallinn, Estonia*. ACM, 97–108. https://doi.org/10.1145/3338906.3338963

[63] Chong Wang, Xin Peng, Zhenchang Xing, and Xiujie Meng. 2023. Beyond Literal Meaning: Uncover and Explain Implicit Knowledge in Code Through Wikipedia-Based Concept Linking. *IEEE Trans. Software Eng.* 49, 5 (2023), 3226–3240. https://doi.org/10.1109/TSE.2023.3250029

[64] Chong Wang, Xin Peng, Zhenchang Xing, Yue Zhang, Mingwei Liu, Rong Luo, and Xiujie Meng. 2023. XCoS: Explainable Code Search based on Query Scoping and Knowledge Graph. *ACM Transactions on Software Engineering and Methodology* (2023).

[65] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 382–394. https://doi.org/10.1145/3540250.3549113

[66] Haoye Wang, Xin Xia, David Lo, John C. Grundy, and Xinyu Wang. 2021. Automatic Solution Summarization for Crash Bugs. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1286–1297. https://doi.org/10.1109/ICSE43902.2021.00117

[67] Lu Wang, Xiaobing Sun, Jingwei Wang, Yucong Duan, and Bin Li. 2017. Construct Bug Knowledge Graph for Bug Resolution: Poster. In *39th International Conference on Software Engineering, ICSE 2017 - Companion Volume, May 20-28, 2017 , Buenos Aires, Argentina*. IEEE Computer Society, 189–191. https://doi.org/10.1109/ICSE-C.2017.102

[68] Moshi Wei, Nima Shiri Harzevili, Yuchao Huang, Junjie Wang, and Song Wang. 2022. CLEAR: Contrastive Learning for API Recommendation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 376–387. https://doi.org/10.1145/3510003.3510159

[69] Bernard L Welch. 1947. The generalization of Student's problem when several different population variances are involved. *Biometrika* 34, 1/2 (1947), 28–35.

[70] Rui Xie, Long Chen, Wei Ye, Zhiyu Li, Tianxiang Hu, Dongdong Du, and Shikun Zhang. 2019. DeepLink: A Code Knowledge Graph Based Deep Learning Approach for Issue-Commit Link Recovery. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*. IEEE, 434–444. https://doi.org/10.1109/SANER.2019.8667969

[71] Wenkai Xie, Xin Peng, Mingwei Liu, Christoph Treude, Zhenchang Xing, Xiaoxin Zhang, and Wenyun Zhao. 2020. API method recommendation via explicit matching of functionality verb phrases. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 1015–1026. https://doi.org/10.1145/3368089.3409731

[72] Shuangshuang Xing, Mingwei Liu, and Xin Peng. 2021. Automatic Code Semantic Tag Generation Approach Based on Software Knowledge Graph. *Journal of Software* 33, 11 (2021), 4027–4045.

[73] Bowen Xu, Zhenchang Xing, Xin Xia, and David Lo. 2017. AnswerBot: Automated Generation of Answer Summary to Developersź Technical Questions. In *32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, October 30 - November 03, 2017, Urbana, IL, USA*. IEEE Computer Society, 706–716. https://doi.org/10.1109/ASE.2017.8115681

[74] Xuejiao Zhao, Zhenchang Xing, Muhammad Ashad Kabir, Naoya Sawada, Jing Li, and Shang-Wei Lin. 2017. HDSKG: Harvesting domain specific knowledge graph from content of webpages. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. IEEE Computer Society, 56–67. https://doi.org/10.1109/SANER.2017.7884609

Received 2023-02-02; accepted 2023-07-27