Improving API Knowledge Comprehensibility: A Context-Dependent Entity Detection and Context Completion Approach using LLM

Zhang Zhang, Xinjun Mao*, Shangwen Wang*, Kang Yang, Tanghaoran Zhang, Fei Gao, Xunhui Zhang
College of Computer Science and Technology
National University of Defense Technology
Changsha, China
{zhangzhang14, xjmao, wangshangwen13, yangkang, zhangthr, gaofei, zhangxunhui}@nudt.edu.cn

***Abstract*—Extracting API knowledge from Stack Overflow has become a crucial way to assist developers in using APIs. Existing research has primarily focused on extracting relevant API-related knowledge at the sentence level to enhance API documentation. However, this level of extraction can lead to a loss of crucial context, especially when sentences contain context-dependent entities (i.e., whose understanding requires reference to the surrounding context) that may hinder developers' understanding. To investigate this issue, we conducted an empirical study of 384 Stack Overflow posts and found that (1) approximately one-third of API functionality sentences contain context-dependent entities, and (2) these entities fall into two categories: Referential Context-Dependent Entities and Local Variable Context-Dependent Entities. In response, we developed a novel method, CEDCC, which combines an entity filtering strategy informed by insights from our empirical study, with a large language model (LLM) to construct coreference chains for detecting context-dependent entities. Additionally, it employs a step-by-step approach with the LLM to complete the necessary context for understanding these entities. To evaluate CEDCC, we constructed a dataset of 1,023 API knowledge sentences, including 567 context-dependent entities and their required contexts. The results demonstrate the effectiveness of CEDCC in accurately detecting context-dependent entities and completing context tasks, achieving an F1-score of 0.865 and a BERTScore of 0.373, significantly surpassing the baseline methods. Human evaluations further confirmed that CEDCC effectively improves the comprehensibility of API knowledge sentences.**

***Index Terms*—API Knowledge, Stack Overflow, Context-dependent, large language model**

I. Introduction

In modern software development, Application Programming Interfaces (APIs) play a critical role in enabling software reuse. However, API documentation often lacks practical examples and comprehensive explanations [1], [2]. Consequently, developers frequently turn to online communities such as Stack Overflow (SO) to obtain API-related knowledge, including information on API functionality and usage examples [3], [4]. An empirical study on developers' API knowledge needs in SO indicates that knowledge describing API functionality is the most in-demand knowledge for developers and rich API functionality knowledge has been accumulated [5]. Building on these insights, this paper focuses on API functionality knowledge. For clarity, "API knowledge" will hereafter refer to API functionality knowledge unless otherwise specified.

*Corresponding author.

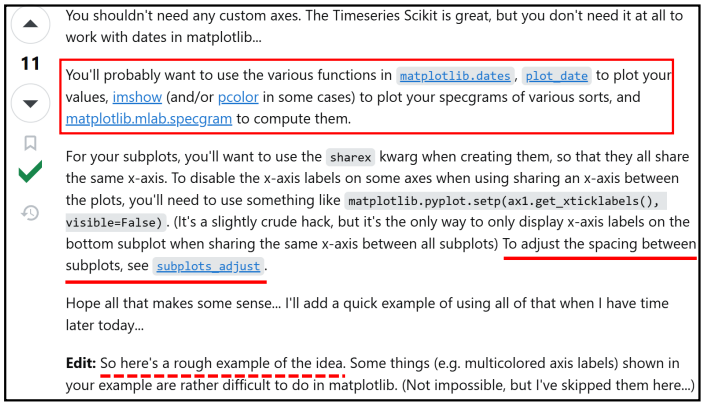

Fig. 1. An example from Stack Overflow illustrating API knowledge extraction at the post and sentence levels.

Existing research has aimed to extract API knowledge from SO to improve API documentation at various levels of granularity [6], [7], including post-level [8] (extracting entire questions, answers, or comments) and sentence-level [9] (isolating individual sentences that specifically contain API-related information). For example, consider the SO post shown in Fig. 1, which mentions the API *matplotlib.pyplot.subplots_adjust*. In post-level extraction, all sentences within the post are considered relevant to the API, resulting in the inclusion of both sentences that describe the API's functionality (highlighted with red underlines) and those unrelated to the API, such as the sentence "So here's a rough example of the idea" (highlighted with red dotted lines). These unrelated sentences introduce unnecessary information, making it more difficult for developers to focus on the key details. In contrast, sentence-level extraction targets only the sentence marked with red underlines, reducing redundancy and emphasizing the most relevant information. As a result, sentence-level extraction has become the preferred approach [10], [11].

Although sentence-level extraction has many advantages,

it also leads to a loss of context, which may hinder the comprehension of the API [12]. Take another sentence in Fig. 1 for example: "... *plot_date* to plot your values" (highlighted with a red box). This sentence describes the functionality of the *matplotlib.pyplot.plot_date* API by indicating its use—"plot your values." However, fully understanding what "your values" refers to requires additional context (i.e., other sentences that were not extracted during the sentence-level extraction). Therefore, we refer to entities like "your values" as **context-dependent entities**, and extracting this sentence alone as a functional description of *matplotlib.pyplot.plot_date* could undermine developers' understanding and reduce its practicality. In contrast, the sentence describing the functionality of *matplotlib.pyplot.subplots_adjust* marked with red underlines in Fig. 1 can be understood without additional context, indicating that it does not contain context-dependent entities. Thus, detecting context-dependent entities within API knowledge sentences and providing the necessary context are crucial for improving the comprehensibility of the extracted sentence-level API knowledge.

Motivated by this, we conducted an empirical study of 384 SO posts to explore the characteristics of context-dependent entities within API knowledge sentences. Our findings revealed that: (1) approximately 1/3 of API knowledge sentences contain context-dependent entities; (2) there are two types of context-dependent entities: Referential Context-Dependent Entities and Local Variable Context-Dependent Entities; and (3) the context required to assist in understanding these entities is often distributed both within the post containing the API sentence and across other posts. These results underscore the need to detect context-dependent entities and complete the necessary context, which has inspired the design of our correspondence detection and completion approach.

Based on these findings, we developed CEDCC (**C**ontext-dependent **E**ntity **D**etection and **C**ontext **C**ompletion), an approach designed to improve the comprehensibility of API knowledge. This method takes an API knowledge sentence as input and identifies both the context-dependent entities within it and the context necessary for understanding them. Specifically, in the detection stage, we adopt corresponding filtering strategies to filter candidate entities for each type of context-dependent entity identified in our empirical study, leveraging syntactic and code analysis. Next, we utilize a large language model (LLM) to construct coreference chains (i.e., a sequence of phrases in a text that refer to the same entity [13]) that include the candidate entities. A rule-based analysis is then performed on these chains to detect context-dependent entities. In the completion stage, we implement a step-by-step context completion process using the LLM. The LLM first extracts context related to the context-dependent entity from SO posts. Drawing on our empirical analysis of the distribution of required context, we limit the scope of SO posts to the SO thread where the entity appears, including the question, accepted answer, other answers, and their comments. Finally, the LLM uses the selected relevant context to generate the necessary context to aid in comprehending the entities.

We also constructed a specialized dataset to evaluate the performance of CEDCC. This dataset includes 1,023 API knowledge sentences from five commonly used Python libraries: matplotlib, pandas, numpy, scipy, and sklearn. It contains 567 context-dependent entities, along with the corresponding contexts needed for their comprehension. Since no existing methods directly address our task, we selected two methods to adapt as baselines. The first baseline is based on LLM, which leverages the LLM's strong performance across various natural language processing (NLP) tasks to detect context-dependent entities and complete the necessary context. The second baseline is based on AllenNLP[1], which excels in coreference resolution, a task similar to ours that involves identifying and linking pronouns or noun phrases to their referents within a text [14]. The results show that CEDCC significantly outperforms both baselines in detecting context-dependent entities, achieving a precision of 0.860, a recall of 0.869, and an F1 score of 0.865, as well as in completing the required context, with a ROUGE-1 score of 0.395, a ROUGE-L score of 0.369, and a BERTScore of 0.373. We also conducted ablation experiments to assess the contribution of different components within CEDCC. The results showed that these components all positively contributed to CEDCC's overall performance. Additionally, a human evaluation of CEDCC's performance confirmed that it effectively improves the comprehensibility of API knowledge sentences.

In summary, the contributions of this paper are:

- We conducted an empirical study of 384 SO posts and identified two types of context-dependent entities in API knowledge sentences, which hinder developers' understanding of API knowledge within these sentences.
- We developed a novel method, CEDCC, that can detect context-dependent entities in API knowledge sentences and complete the necessary context. This method can be used independently or alongside existing API knowledge extraction techniques to enhance the comprehensibility of the extracted API knowledge.
- We constructed a specialized dataset[2] consisting of 1,023 API knowledge sentences and 567 context-dependent entities, which can be used to evaluate both the detection of context-dependent entities and the completion of the context that assists in understanding these entities.

II. Empirical Study

To understand the challenges and potential solution space for detecting context-dependent entities in API knowledge sentences within SO posts, as well as to complete their necessary context, we conducted an empirical study aimed at answering the following research questions:

- RQ1: How prevalent are context-dependent entities in API functionality sentences?
- RQ2: What types of context-dependent entities are in API functionality sentences?

[1]https://github.com/allenai/allennlp

[2]https://github.com/ZHANGDOUBLE96/CEDCC

- RQ3: What is the distribution of the context on which the entity depends?

A. Data Collection

First, we downloaded the official SO data as of December 2023[3]. We then selected posts related to five popular Python libraries, namely NumPy, Pandas, Matplotlib, SciPy, and Scikit-learn, as our study subjects, given the widespread use of these libraries in the developer community and their extensive APIs[4]. Additionally, there are a substantial number of discussions on SO related to these libraries, providing a rich dataset for analyzing context dependency in API sentences.

To ensure the reliability of the extracted API sentences, developers typically extract them from accepted answers in SO [15], [16], which are the answers deemed by the questioner to best solve their problem. Following this practice, we selected accepted answers as our knowledge source. Specifically, we collected all accepted answers from SO that were tagged with at least one of the five Python libraries mentioned earlier, resulting in a total of 290,316 accepted answers.

Second, due to the large volume of answers, conducting a manual analysis of each answer was impractical. Therefore, we employed a sampling method to ensure both the feasibility of the analysis and the representativeness of the results. We randomly sampled 384 answers from this dataset, with a 95% confidence level and a 5% confidence interval [17]. Since context-dependent entities needed to be annotated at the sentence level, we used the NLTK sentence parser [18], ultimately obtaining 1,981 sentences from these 384 answers.

B. RQ1:Prevalence of Context-Dependent Entities

Approach. To answer RQ1, we employed two master's students proficient in Python to manually classify 1,981 sentences in the dataset, determining whether each sentence describes API functionality. In cases where their classifications differed, one of the authors facilitated a discussion with them to reach a consensus. Once the API function sentences were identified, the next step was to determine whether these sentences contained entities that required external context for comprehension, followed by annotating those entities.

Results. As a result, we identified 373 API functionality sentences, achieving a Cohen's Kappa coefficient of 0.81 [19]. Among these, 112 sentences contained 141 context-dependent entities, all of which were noun phrases (i.e., groups of words that function as nouns in a sentence, typically consisting of a noun or pronoun as the head and any associated modifiers [20]), with a Cohen's Kappa coefficient of 0.84. In other words, approximately one-third of the sentences describing API functionality require additional context beyond the sentence itself for complete understanding. This finding highlights the importance of detecting context-dependent entities and completing the necessary context in API sentences.

[3]https://archive.org/details/stackexchange

[4]https://lp.jetbrains.com/python-developers-survey-2021/

C. RQ2:Types of Context-Dependent Entities

Approach. To answer RQ2, the two master's students, along with one of the authors, classified the context-dependent entities identified in RQ1. The classification process followed an iterative approach similar to the methods used by Liu *et al.* [5] and Snow *et al.* [21]. Specifically, we began by randomly sampling 10 sentences from the 112 sentences identified in RQ1 as containing context-dependent entities for exploratory annotation. Through independent annotations and subsequent discussions among annotators, we established an initial type called "Referential Phrases", referring to phrases that contain demonstrative pronouns, such as "this data". Following this, we annotated the remaining sentences in the dataset. If an entity was identified as context-dependent but did not match any of the existing types, we revised the definitions of current context-dependent entity types or created a new type based on discussions. If any changes to the entity types occurred, all sentences were re-annotated accordingly. This iterative approach ensured comprehensive classification.

It is important to note that, during the annotation process, some entities were recognized as domain-specific terms, such as "Mersenne-Twister PRNG" in the sentence "Random numbers are created by the Mersenne-Twister PRNG in numpy.random." Since our focus was on entities requiring additional context beyond the API sentence for understanding, and the comprehension of domain-specific terms relies on the annotators' expertise in their respective fields, we chose not to classify these entities as context-dependent, despite their potential to hinder users' understanding of API knowledge.

To further validate the correctness and completeness of our classification of context-dependent entity types, we employed another two master's students proficient in Python (who had not participated in the previous annotation) to annotate a subset of the API functionality sentences using our coding protocol. Specifically, we randomly selected 20 API sentences from 373 API functionality sentences, and the two students independently annotated the context-dependent entities based on our definitions. If none of the existing definitions were applied, they labeled the sentence as "New Type." The Cohen's Kappa coefficient [19] for this annotation process was 1. The results of this round were consistent with our previous annotations, and no new entity types were identified.

Results. After the above process, we identified two types of context-dependent entities: Referential Context-Dependent Entities (**RCDE**) and Local Variable Context-Dependent Entities (**LVCDE**). Their definitions are as follows:

- RCDE: If an entity includes referencing terms, such as demonstrative pronouns (e.g., "this") or spatial references (e.g., "above"), and understanding it requires additional context beyond the API sentence, it is classified as an RCDE.
- LVCDE: If an entity is a local variable defined in a code snippet (e.g., *df* in the *CODE* section of Fig. 2) or displayed in a data snippet (i.e., a small, structured extract of a dataset, typically shown in tabular formats, such as

TABLE I
THE DETAILS OF CONTEXT-DEPENDENT ENTITY TYPES

Type	Example	Count
RCDE	you can convert **it** to a dict using pandas.DataFrame.to_dict method	114
LVCDE	use numpy.setdiff1d to find all the rows of **df_a** that not in the inner join	27

Sell in the *DATA* section of Fig. 2), and understanding it requires additional context beyond the API sentence, it is classified as an LVCDE.

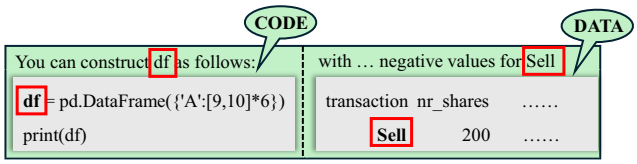

Fig. 2. Example of two types of local variables.

Table I provides examples and the count of occurrences for RCDE and LVCDE. In the examples, the context-dependent entities are highlighted in bold. Among these, RCDE is the most frequently occurring type.

D. RQ3:Distribution of Context Dependencies

Approach. To answer RQ3, we first collected the SO posts associated with each context-dependent entity. This included the accepted answer containing the entity, the corresponding question, other answers to the same question, and their comments. The two master's students involved in the annotations for RQ1 and RQ2 continued their work by reviewing the SO posts to identify the specific context needed to understand each context-dependent entity. For each entity, they recorded the location in the SO post where the necessary context was found. It is important to note that if the required context appeared in a comment, the location was recorded as the corresponding question or answer to which the comment belonged. If the context appeared in multiple locations, the first occurrence (ordered by time) was recorded as the reference. In cases where the annotators disagreed, one of the authors facilitated a discussion to resolve the conflicts.

Results. Table II presents the distribution of required contexts for each type of context-dependent entity. We found that, regardless of the type of entity, the context needed for understanding could be distributed across various parts of the SO posts (e.g., questions, accepted answers, and other answers), without being limited to a specific part. This finding suggests that when designing methods to complete context, it is essential to consider all parts of the SO posts.

III. OUR PROPOSED APPROACH

The overall framework of our approach is shown in Fig. 3. Generally, CEDCC consists of three stages: candidate entity selection, context-dependent entity confirmation, and step-by-step context completion. The first two stages focus on detecting context-dependent entities within a sentence, while the final stage completes the required context needed to understand the detected entities.

TABLE II
DISTRIBUTION OF CONTEXTS ON WHICH ENTITIES DEPEND

Entity Type	Question	Accepted Answer	Other Answer
RCDE	69	44	1
LVCDE	7	20	0

TABLE III
CEDCC CANDIDATE ENTITY SELECTION KEYWORD LIST

Type	Keywords
Personal Pronouns	he, she, it, they, him, her, them
Demonstrative Pronouns	this, that, these, those
Possessive Pronouns	my, your, his, her, its, our, their, mine, yours, hers, ours, theirs
Interrogative Pronouns	what
Indefinite Pronouns	such
Spatial Reference	above, below

A. Stage 1: Candidate Entity Selection

Based on the characteristics of the two types of context-dependent entities identified in Section II, we designed corresponding screening strategies to select candidate entities.

Extraction of candidate RCDE. For extracting candidate RCDEs, we utilized spaCy, a library known for its strong performance in NLP tasks [22]. First, we extracted all noun phrases from the API sentences using spaCy's syntactic parsing capabilities, following the observation from RQ1 that all context-dependent entities are noun phrases. Then, we filter the extracted noun phrases based on a predefined keyword list, selecting a noun phrase as a candidate RCDE if it contains any of the keywords from the list. This keyword list was carefully designed to capture phrases that refer to other entities. The details of the keyword list are provided in Table III, and its construction is based on the findings of our empirical study and relevant linguistics knowledge [23], [24].

Extraction of candidate LVCDE. According to our empirical study results, LVCDEs are local variables defined in code snippets or data snippets in SO posts, as illustrated in Fig. 2. This insight led us to select candidate LVCDE from local variables. Given that code and data snippets are common sources of local variable definitions and assignments, and they are often enclosed in $< Code >$ tags in SO, we first extract the snippets marked by the $< Code >$ tag from the posts where the given API sentence is located.

After extracting them, we differentiated them and applied specific processing strategies for each type to extract local variables. Specifically, we use Tree-sitter[5], a robust parsing

[5]https://github.com/tree-sitter/tree-sitter

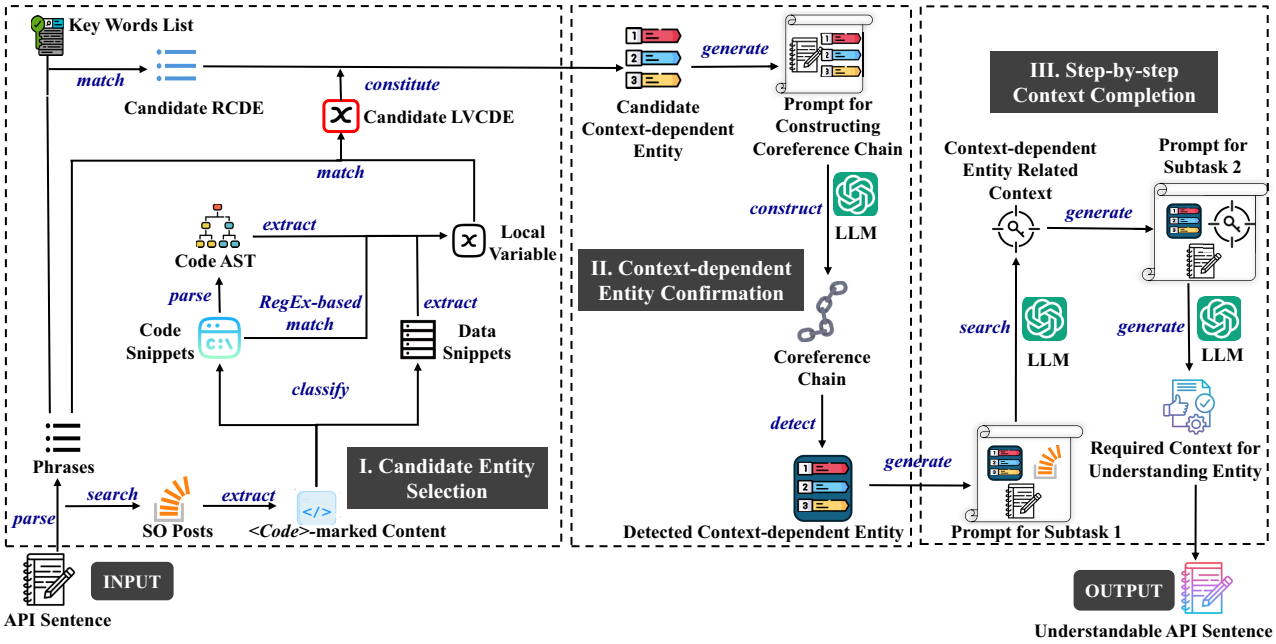

Fig. 3. Framework of our approach.

library capable of handling complex programming language grammars, to parse each line of the snippet. If a line was successfully parsed, it was classified as a code snippet. For lines that Tree-sitter could not parse, we applied heuristic analysis to detect typical code features. We searched for programming constructs such as keywords (e.g., `for`, `while`, `return`), assignment operator (=), and other syntax elements indicative of code. Lines meeting these criteria were also classified as code snippets. Finally, lines that did not meet the code criteria were classified as data snippets. Then we processed code snippets and data snippets as follows:

- **Code snippets.** For successfully parsed code snippets, we used Tree-sitter to build an abstract syntax tree (AST) to extract code identifiers, including variable names and function names. For code snippets that fail to parse, we applied regular expression matching, such as extracting the string before "=". Since programming language keywords and built-in functions do not typically require additional context for understanding, we excluded these from our identifier list. Specifically, we excluded Python keywords, built-in functions, and the APIs of the five Python libraries used in our study (i.e., the libraries mentioned in Section II), including functions, methods, and classes. These exclusions were determined based on the official API documentation. This refinement allowed us to focus our analysis on user-defined identifiers, which are more likely to be context-dependent.
- **Data snippets.** For data snippets, we separated the text by spaces and newlines to extract all row names, column names, and values. We then filtered out pure numbers and dates and removed duplicates to obtain the local variables from the data snippets.

Through these steps, we generated a list of local variables defined in the SO posts containing the API sentence. A noun phrase extracted from the API sentence was selected as a candidate LVCDE if it appeared in the local variables list.

B. Stage 2: Context-dependent Entity Confirmation

After identifying candidate context-dependent entities, we need to determine whether these entities are truly context-dependent, i.e., whether understanding them requires additional context beyond the sentence itself. To achieve this, we aimed to assess whether there are other entities within the API sentence that refer to the candidate entities and are not local variables or phrases containing referencing terms. The underlying intuition is that if such entities exist, the API sentence itself would provide sufficient information to understand the candidate entities, meaning the referred candidates would not be context-dependent. If no such entity exists, the candidate is classified as context-dependent.

Motivated by this, we construct coreference chains within API sentences and analyze the chain to which each candidate belongs to determine whether the aforementioned entities exist, and thus, whether the candidate is context-dependent. A coreference chain is a sequence of phrases in a text that refer to the same entity [13]. Some API libraries, such as AllenNLP, provide models for constructing coreference chains by analyzing entity references in text. However, certain context-dependent entities refer to concrete objects (e.g., "*it*" in Case 2 of the example in Fig. 4), while others refer to abstract processes (e.g., "*this*" in Case 1 of the example in Fig. 4), which do not have a concrete object. This poses a challenge for existing coreference resolution models, which often struggle to handle such abstract references [25], [26]. Recently, LLMs

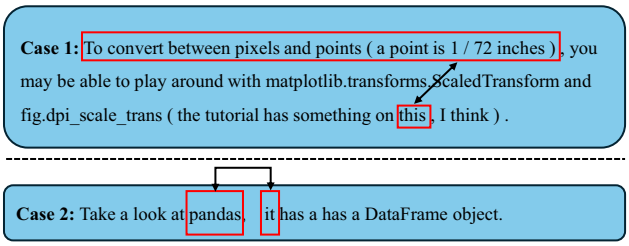

Fig. 4. The example for two types of reference.

have demonstrated strong information extraction capability and have been effective in processing abstract references [27], [28]. Therefore, we consider LLMs an ideal choice for constructing coreference chains.

To reduce the complexity of inferring coreference chains and allow LLMs to focus on constructing coreference chains involving candidate entities, we designed the prompt shown in Fig. 5. Specifically, we marked the context-dependent candidate entities in the sentence using special symbols to facilitate their identification by the LLM during analysis. We then instructed the LLM to only generate coreference chains including these marked entities, rather than inferring coreference for all entities within the API knowledge sentence.

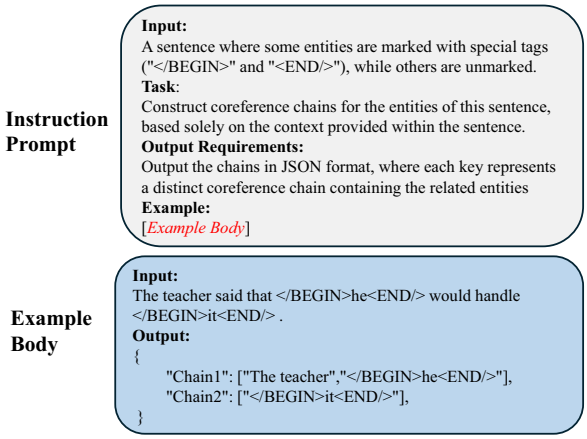

Fig. 5. The prompt for constructing coreference chain.

After generating the coreference chains, we analyze each chain by traversing through it. Specifically, if all entities in a coreference chain are candidate entities, we conclude that understanding them requires context beyond the sentence, classifying them as context-dependent entities. Conversely, if the coreference chain includes non-candidate entities, the candidate entities in that chain are not considered context-dependent. Using the *Example Body* in Fig. 5 as an example, "he" and "it" are both identified as candidate context-dependent entities through filtering. In the coreference chain where "he" appears (Chain1), there is a non-candidate entity, "the teacher." In contrast, all entities in the coreference chain for "it" (Chain2) are candidates, consisting solely of "it." Therefore, we classify "he" as a non-context-dependent entity, while "it" is a context-dependent entity.

C. Stage 3: Step-by-step Context Completion

In this stage, we employ a step-by-step strategy to guide the LLM in completing the required context for understanding context-dependent entities. This approach is based on the idea that breaking tasks into multiple sub-tasks, rather than directly using the LLM for inference, helps guide the model incrementally. This ensures a more thorough understanding of the context, reduces ambiguity in interpreting complex semantic relationships, and improves the quality of inference [27], [29], [30].

As shown in the third stage of Fig. 3, the context completion task is divided into two subtasks. In the first subtask, we input the API sentences, the detected context-dependent entities, and the candidate context required for understanding these entities into the LLM. The candidate context is defined as the SO thread containing the API sentence, which includes the question, all answers, and comments. This scope was determined by the findings of RQ3. The LLM is then instructed to extract the portions of the candidate context that are relevant to the detected context-dependent entities (e.g., the sections highlighted by red boxes in Fig. 6). In the second subtask, the outputs from the first subtask, along with the API sentences and the detected context-dependent entities, are fed into the LLM. The LLM is tasked with generating the necessary context to improve comprehension of the context-dependent entities (e.g., the natural language explanation of "your features" shown in Fig. 6). The prompts used for these two subtasks are available online[6].

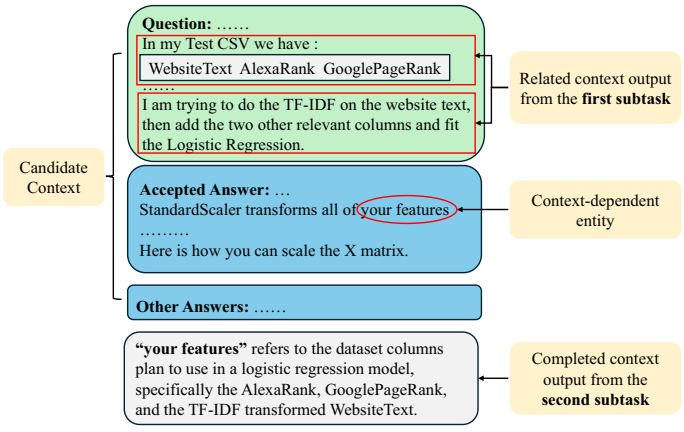

Fig. 6. An example: completing the context required for understanding entity "your features".

IV. Experiment Settings

In the experiment, we want to answer the following three research questions (RQs):

- RQ4. How well does our approach perform in detecting context-dependent entities and completing the required context?

[6]https://github.com/ZHANGDOUBLE96/CEDCC

- RQ5. What is the contribution of different components to the performance of our approach?
- RQ6. Can our approach improve developers' understanding of API sentences?

A. Datasets

We expanded the dataset constructed in the empirical study by conducting additional sampling. Specifically, we randomly sampled 616 additional accepted answers from the 290,316 answers obtained in the empirical study and combined them with the previously analyzed 384 answers. This increased the total number of answers in the dataset to 1,000, ensuring a 99% confidence level with a 5% confidence interval. Following the annotation methodology used in the empirical study, we identified API functionality sentences and annotated context-dependent entities, achieving Cohen's Kappa coefficients of 0.87 and 0.84, respectively.

Next, we collected the questions corresponding to the accepted answers, as well as other answers and their comments. In total, we gathered 1,000 questions, 1000 accepted answers, 966 other answers, and 4,703 comments. We analyzed these SO posts to identify and complete the necessary context for understanding the context-dependent entities, using an iterative process of independent annotation followed by discussions to resolve discrepancies. In total, we obtained 1,023 API sentences containing 567 context-dependent entities, comprising 490 RCDEs and 77 LVCDEs.

It is important to note that the detection task is evaluated at the entity level, meaning the number of test samples corresponds to the total number of entities, not just the context-dependent ones. According to the principle that a sample size of around 384 is sufficient to achieve a 95% confidence level with a 5% confidence interval as the total sample size increases [17], evaluating 567 context-dependent entities for the context completion task provides a reliable reflection of `CEDCC`'s performance on a larger-scale test set.

B. Performance Measures

In our study, we aim to show the competitiveness of `CEDCC` through both automatic and human evaluation.

Automatic Evaluation. The detection of context-dependent entities can be considered a named entity recognition (NER) task. Therefore, we use commonly adopted evaluation metrics in NER tasks, including precision, recall, and F1 score [31]. Precision measures the proportion of detected context-dependent entities that are correctly identified, while recall measures the proportion of true context-dependent entities that are correctly identified. The F1 score is the harmonic mean of precision and recall, providing a balanced evaluation.

Since the expressions of context needed to understand the entities can vary, the context completion task is framed as a generative task. Accordingly, we employ two commonly used evaluation metrics for this task: ROUGE [32] and BERTScore [33]. Specifically, we use ROUGE-1, which measures unigram overlap, and ROUGE-L, which captures the longest common subsequence between the generated and reference texts. These metrics evaluate the quality of the generated context completions in terms of both lexical similarity and sequence structure. Additionally, we use BERTScore to provide a semantic-level evaluation by comparing the contextual embeddings of the generated and reference context completions, thus assessing how well the meaning is preserved.

Human Evaluation. Our human evaluation primarily follows the methodology of Lin *et al.* [34]. We employed six Ph.D. students, none of whom are co-authors of this paper. Each participant has over five years of programming experience and is familiar with Python API libraries. To avoid bias, where participants may have prior assumptions about the method used to generate API sentences that could influence their evaluation, the processing method used for each API sentence was anonymized in the questionnaire, and each participant completed the questionnaire independently. Before the evaluation, we provided detailed guidelines on detecting context-dependent entities and completing their required contexts. Each participant was asked to score each API sentence based on the following four aspects:

- **Accuracy**, which reflects how correctly the completed context explains the context-dependent entities.
- **Independence**, which measures the extent to which the processed API sentences can be understood without additional context.
- **Comprehensibility**, which reflects how readable and understandable the processed API sentences are.
- **Practicality**, which evaluates the usefulness of the processed API sentences in aiding software development.

All scores are given on a scale from 1 to 5, with 1 indicating poor, 2 marginal, 3 acceptable, 4 good, and 5 excellent.

C. Baselines

Since no existing methods directly address our task, we selected two methods to adapt as baselines:

- `LLM-baseline`: Directly and solely uses LLM (i.e., *gpt-4-0613*) to detect context-dependent entities in sentences and complete the necessary context.
- `AllenNLP-baseline`: Use AllenNLP's coreference resolution model to build and analyze coreference chains for detecting context-dependent entities and completing the necessary context.

D. Implementation Details

All experiments were conducted on a computer equipped with an Intel(R) Core(TM) i9-10900X CPU and a GeForce RTX 4090 GPU with 24 GB of memory, running Ubuntu 20.04 as the operating system. The version of LLM used is *gpt-4-0613*, with a temperature setting of 0.5. For spaCy, we used version 3.7.3, with the *en_core_web_sm* model as the NLP parser. For the code parser Tree-sitter, we used version 0.23.2. For the baselines, we used AllenNLP version 2.10.0, with the coreference resolution model *coref-spanbert-large-2021.03.10*.

TABLE IV
THE OVERALL PERFORMANCE OF CEDCC VS BASELINE IN DETECTING CONTEXT-DEPENDENT ENTITIES AND COMPLETING CONTEXT

Method	Detect			Complete in $Scope_{All}$			Complete in $Scope_{Identified}$		
	P	R	F1	ROUGE-1	ROUGE-L	BERTScore	ROUGE-1	ROUGE-L	BERTScore
CEDCC	**0.860**	**0.869**	**0.865**	**0.395**	**0.369**	**0.373**	**0.453**	**0.423**	**0.428**
LLM-baseline	0.267	0.638	0.377	0.224	0.208	0.192	0.363	0.337	0.310
AllenNLP-baseline	0.167	0.843	0.279	0.050	0.049	0.034	0.122	0.120	0.084

V. RESULTS ANALYSIS

A. RQ4:The Overall Performance

Approach. In this RQ, we compare the performance of CEDCC and two baselines in detecting and completing phases.

Results. The comparison results are presented in Table IV. Notably, the evaluation of context completion was divided into two scopes: (1) the evaluation of the completion for all context-dependent entities, referred to as $Scope_{All}$, where an empty string was used as the completion result if a context-dependent entity was not correctly identified; and (2) the evaluation of the completion for only those context-dependent entities that were correctly identified, referred to as $Scope_{Identified}$. The best value in each column is highlighted.

In the detection phase of context-dependent entities, CEDCC outperformed the baseline methods across all metrics, achieving precision, recall, and F1 of 0.860, 0.869, and 0.865, respectively. In contrast, the LLM-baseline showed a low precision of 0.267. This suggests that identification of context-dependent entities requires analytical processes, such as constructing coreference chains, rather than solely relying on a limited understanding of LLM in entity identification tasks.

The AllenNLP-baseline achieved a recall of 0.843. However, its precision was only 0.167, indicating a high false positive rate and resulting in an F1 of just 0.279. This issue may be attributed to its limited capacity to handle abstract references. For example, in Case 1 (shown in Fig. 4), "this" refers to the action "To convert between pixels ..." rather than a specific entity, while AllenNLP-baseline fails in this.

In both evaluation scopes of the context completion task, CEDCC significantly outperformed the two baseline methods in both semantic-based (BERTScore) and lexical-based (ROUGE) evaluations. While the performance of the LLM-baseline improved in $Scope_{Identified}$, it still fell short of CEDCC. Since both methods utilized LLM for context completion, this suggests that a step-by-step approach to context completion yields better lexical and semantic performance compared to a direct approach. The AllenNLP-baseline performed poorly in both scopes, likely due to its relatively weaker ability to handle long texts and abstract references.

Results for RQ4: CEDCC significantly outperformed both baselines in detecting context-dependent entities and completing context via automatic evaluation.

B. RQ5:Ablation Study

Approach. In this RQ, we investigate the contribution of different components of CEDCC to its overall performance through an ablation study. Specifically, we design several variants of CEDCC as follows:

- **CEDCC-RES**: Uses **r**egular **e**xpressions to **s**elect candidate entities by identifying noun phrases that include pronouns or exhibit morphological characteristics typical of code naming conventions (e.g., CamelCase).
- **CEDCC-WCC**: Directly prompts the LLM to determine whether a candidate entity is context-dependent **w**ithout **c**onstructing **c**oreference chains.
- **CEDCC-CCWT**: **C**onstructs **c**oreference chains **w**ithout specifying **t**arget entities, then filters the chains to identify those containing the target entities after construction.
- **CEDCC-DCC**: Prompts the model to **d**irectly **c**omplete the **c**ontext, rather than using a step-by-step strategy.

Results. Table V presents the performance of CEDCC and its variants. It is clear that CEDCC achieves the best performance across all metrics, validating the contribution of each component to the overall system's effectiveness. In the context-dependent entity detection stage, the low performance of **CEDCC-RES** suggests that entities identified through syntactic and code analysis are more accurate than those obtained via regular expression matching. This, in turn, impacts the quality of coreference chains constructed by the LLM. The discrepancy likely stems from the fact that regular expressions capture only morphological features of entities (e.g., CamelCase), without accounting for contextual relationships.

When comparing methods that use the same candidate entity selection process but differ in how they determine whether an entity is context-dependent, **CEDCC-WCC** shows the worst performance. This may be because identifying context dependence requires explicit rules, such as coreference chain construction and analysis. **CEDCC-CCWT** also underperforms compared to CEDCC, likely due to CEDCC's pre-selection of candidate entities, which enables the LLM to focus on specific entities during coreference chain construction. By clearing target entities, CEDCC reduces interference from unrelated entities, resulting in higher-quality coreference chains.

Regarding strategies for completing context, CEDCC's step-by-step approach, which first narrows the context for the LLM to analyze, enables the model to focus more effectively on completing the essential contextual details in the second step. In contrast, **CEDCC-DCC**'s direct context completion

TABLE V
THE ABLATION STUDY OF CEDCC AND ITS VARIANTS

Method	Detect			Complete in $Scope_{All}$			Complete in $Scope_{Identified}$		
	P	R	F1	ROUGE-1	ROUGE-L	BERTScore	ROUGE-1	ROUGE-L	BERTScore
CEDCC	**0.860**	**0.869**	**0.865**	**0.395**	**0.369**	**0.373**	**0.453**	**0.423**	**0.428**
CEDCC-RES	0.271	0.795	0.405	0.311	0.286	0.278	0.393	0.361	0.351
CEDCC-WCC	0.735	0.309	0.435	0.127	0.116	0.111	0.413	0.377	0.360
CEDCC-CCWT	0.756	0.785	0.770	0.291	0.263	0.263	0.369	0.333	0.334
CEDCC-DCC	**0.860**	**0.869**	**0.865**	0.335	0.315	0.305	0.419	0.394	0.381

approach requires the LLM to handle a broader scope of information, increasing the likelihood of being distracted by irrelevant details, thus reducing the overall performance of the completion process. This step-by-step strategy contributes to CEDCC's superior performance over **CEDCC-DCC**.

Results for RQ5: Each component of CEDCC plays a crucial role in its overall performance.

C. RQ6:Human Evaluation

Approach. Following the sample sizes used in similar manual evaluation [35], we randomly selected 50 API sentences containing context-dependent entities from the dataset. These sentences were processed using AllenNLP-baseline, LLM-baseline, and CEDCC, along with the original API sentences, to form the questionnaire. Since the original API sentences lack completed context for the context-dependent entities, *Accuracy* was not evaluated for them. Each participant was required to evaluate 150 sentences for the *Accuracy* dimension and 200 sentences for each of the other three dimensions. After collecting their scores, we used Fleiss' Kappa [36] to measure inter-rater agreement. The Kappa values for *Independence*, *Practicality*, *Comprehensibility*, and *Accuracy* were 0.746, 0.782, 0.764, and 0.800, respectively, indicating substantial agreement among participants.

Results. Table VI shows that CEDCC outperforms the baseline methods across all dimensions, demonstrating its effectiveness in identifying context-dependent entities in API sentences and providing the necessary contextual information to enhance understanding. This leads to significant improvements in the comprehensibility and practicality of API sentences. In terms of *Independence* and *Accuracy*, LLM-baseline performs relatively well, though slightly below CEDCC, while AllenNLP-baseline performs the worst. This observation aligns with our conclusions from RQ4, which evaluated the detection and completion phases using automated metrics. The original sentences received the lowest scores for *Independence* (2.0), which is consistent with the fact that the selected API sentences inherently contain context-dependent entities.

Case Analysis. Fig. 7 presents an example[7] of identifying and completing context-dependent entities in API sentences using CEDCC, compared to LLM-baseline and AllenNLP-baseline. CEDCC produces results closely aligned with the ground truth in both vocabulary and semantics, achieving high scores for *Independence* and *Accuracy* (5 and 4.6, respectively). It accurately describes the data type of *page_id* as a column in a DataFrame (i.e., a Series) and provides additional explanation of the *pandas.factorize* API. This clarity contributes to higher scores in *Practicality* and *Comprehensibility* (both 4). Notably, one participant gave a *Practicality* score of 2 (i.e., marginal) because he had already mastered this development knowledge.

[7]https://stackoverflow.com/questions/15829494

TABLE VI
THE RESULTS OF OUR HUMAN EVALUATION

Method	Independence	Practicality	Comprehensibility	Accuracy
Original Sentence	2.0	2.4	2.6	-
LLM-baseline	3.5	4.0	3.8	4.0
AllenNLP-baseline	2.4	3.0	3.3	1.4
CEDCC	**4.0**	**4.4**	**4.1**	**4.6**

In contrast, AllenNLP-baseline correctly identifies *page_id* as a context-dependent entity but fails to provide a meaningful completion, leading to low scores for *Accuracy* and *Practicality* (1.3 and 1.8, respectively). This failure arises because page_id's reference needs to be extracted from the data snippet within the context, a task that AllenNLP-baseline's coreference resolution model is unable to perform, resulting in an empty completion.

The LLM-baseline incorrectly identifies *array* as context-dependent, although it only requires Python knowledge rather than context from the SO post. Moreover, it fails to clearly specify the data structure for *page_id*, likely due to its direct context completion strategy. CEDCC adopts a step-by-step approach: it first identifies the relevant context defining *page_id* (e.g., the data snippet) and highlights it in the LLM prompt, narrowing the model's focus and improving *Accuracy* (4.2) and *Practicality* (3.5).

Results for RQ6: CEDCC enhances the comprehensibility and practicality of API sentences through manual evaluation.

VI. Related Work

API Knowledge Extraction from SO. API documentation plays a crucial role in software development, but its complexity and diversity often result in hard-to-understand content [37]–[39]. To address this, extensive research has focused on extracting various types of API knowledge from community platforms like SO to assist in understanding [5], [40]. Treude *et al.* [1] use a machine learning-based method to extract insightful sentences from SO to augment API documentation. Nam *et al.* [9] introduce SOREL, a machine learning tool that extracts comparative knowledge of APIs scattered in posts, improving developers' understanding by highlighting differences between comparable APIs. Ren *et al.* [3] utilized SO to supplement knowledge about API constraints, developing a text mining approach that extracts API misuse scenarios from SO. Regarding API functionality knowledge, Shen *et al.* [41] perform syntactic analysis on SO posts to obtain verb-object phrases, thereby providing descriptions for API functionality. While these extraction methods effectively retrieve API knowledge, they could lose context in the extraction, making certain entities harder to understand and reducing the usefulness of the extracted knowledge. Our study addresses this issue by detecting and completing the missing context.

Original API sentence:
……Here I use pandas.factorize() to convert the page_id to an array ……

Related Context:
I have a pandas DataFrame …… which is structured like so:

page_id	score
2430	7.632
……	……

Ground truth:
- ***page_id***: a column in the DataFrame, representing the unique identifier of a page.

Allennlp-baseline:
- ***page_id***:

LLM-baseline:
- ***array***: a data structure that stores a collection of items
- ***page_id*** : an identifier for a specific page within the document

CEDCC:
- ***page_id***: a column in the DataFrame that serves as a unique identifier for each page, used to track individual pages across different scores.

Fig. 7. An Example of detecting context-dependent entities and completing required context in API knowledge sentence.

LLM for Software Engineering. LLMs, due to their powerful natural language and programming capabilities, have been widely in software engineering [42]–[46]. Chen *et al.* introduced ChatUniTest [47], an LLM-based framework for automated unit test generation through an adaptive focal context mechanism and a generation-validation-repair process. InferFix [48], an LLM-powered program repair tool, uses a 12-billion-parameter Codex Cushman model fine-tuned on bug-fix data to generate precise fixes for critical security and performance issues, which enhances the LLM's ability to repair bugs. Li *et al.* propose FSATD [49], a fusion approach that combines ChatGPT with smaller models to detect Self-Admitted Technical Debts. Arora *et al.* [50] investigate how LLMs can improve requirements engineering by enhancing the efficiency and accuracy of tasks such as analysis of software requirements. AERJE [51], an LLM-based framework, focuses on identifying API entities from SO using sequence-to-sequence generation through dynamic prompts. Different from them, our work focuses on applying LLMs to assist in API knowledge extraction. Rather than using LLMs to complete tasks in a single step, we decompose the tasks and design targeted prompts for each subtask.

VII. Threats to Validity

The first threat to internal validity relates to the subjective judgment of the annotators during data annotation. To address this, we followed commonly accepted data annotation principles, such as involving multiple annotators, resolving conflicts, and reporting agreement coefficients where applicable. Another threat concerns the performance and potential limitations of our baseline implementations. To mitigate this, we used models known for their outstanding performance in their respective domains and readily available for use. Specifically, we employed the coreference resolution model *coref-spanbert-large-2021.03.10* from the AllenNLP library and LLM *gpt-4-0613*. The threat to external validity concerns the generalizability of our results and findings. To address this, the test sample size was set to achieve a 95% confidence level with a 5% confidence interval, providing a reliable indication of CEDCC's performance on a larger-scale test set. In the future, we plan to build larger datasets covering more programming languages for further evaluation. The threat to construct validity arises from human studies, which may introduce bias. To minimize this, we provided a tutorial to ensure all participants correctly understood the questionnaire before conducting the human study.

VIII. Conclusion

Existing approaches that extract API-related knowledge from SO at the sentence level could lose essential contextual information, which limits the comprehension of extracted knowledge. In this paper, we first conduct an empirical study to explore this phenomenon, identifying two types of context-dependent entities and analyzing the distribution of the context they rely on. Based on the empirical study, we proposed CEDCC, an automated approach for detecting context-dependent entities in API sentences and completing the context needed to understand them. Experiments demonstrate the effectiveness of CEDCC in detecting context-dependent entities and completing context tasks, with an F1-score of 0.865 and a BERTScore of 0.373, significantly surpassing the baselines. Human evaluations further confirmed CEDCC's effectiveness in improving the comprehensibility of API knowledge.

IX. Acknowledgments

We gratefully acknowledge the financial support from the National Key Research and Development Program of China (2023YFB4503802), the National Science Foundation of China (62172426,62332005).

REFERENCES

[1] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 392–403.

[2] Q. Fan, Y. Yu, T. Wang, G. Yin, and H. Wang, "Why api documentation is insufficient for developers: an empirical study," *Science China. Information Sciences*, vol. 64, no. 1, p. 119102, 2021.

[3] X. Ren, J. Sun, Z. Xing, X. Xia, and J. Sun, "Demystify official api usage directives with crowdsourced api misuse scenarios, erroneous code examples and patches," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 925–936.

[4] Z. Zhang, X. Mao, S. Wang, K. Yang, and Y. Lu, "Career: Context-aware api recognition with data augmentation for api knowledge extraction," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 438–449.

[5] M. Liu, X. Peng, A. Marcus, S. Xing, C. Treude, and C. Zhao, "Api-related developer information needs in stack overflow," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4485–4500, 2021.

[6] G. Uddin, F. Khomh, and C. K. Roy, "Mining api usage scenarios from stack overflow," *Information and Software Technology*, vol. 122, p. 106277, 2020.

[7] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, "Focus: A recommender system for mining api function calls and usage patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1050–1060.

[8] K. Luong, M. Hadi, F. Thung, F. Fard, and D. Lo, "Arseek: identifying api resource using code and discussion on stack overflow," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 331–342.

[9] D. Nam, B. Myers, B. Vasilescu, and V. Hellendoorn, "Improving api knowledge discovery with ml: A case study of comparable api methods," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1890–1906.

[10] G. Uddin and F. Khomh, "Automatic mining of opinions expressed about apis in stack overflow," *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 522–559, 2019.

[11] M. Ahasanuzzaman, M. Asaduzzaman, C. K. Roy, and K. A. Schneider, "Classifying stack overflow posts on api issues," in *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2018, pp. 244–254.

[12] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE software*, vol. 26, no. 6, pp. 27–34, 2009.

[13] K. Lee, L. He, M. Lewis, and L. Zettlemoyer, "End-to-end neural coreference resolution," *arXiv preprint arXiv:1707.07045*, 2017.

[14] R. Mitkov, *Anaphora resolution*. Routledge, 2014.

[15] D. Wu, X.-Y. Jing, H. Zhang, Y. Feng, H. Chen, Y. Zhou, and B. Xu, "Retrieving api knowledge from tutorials and stack overflow based on natural language queries," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–36, 2023.

[16] D. Wu, X.-Y. Jing, H. Zhang, Y. Zhou, and B. Xu, "Leveraging stack overflow to detect relevant tutorial fragments of apis," *Empirical Software Engineering*, vol. 28, no. 1, p. 12, 2023.

[17] W. G. Cochran, "Sampling techniques," *John Wiley& Sons*, 1977.

[18] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[19] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[20] M. S. Dryer, "Noun phrase structure," *Language typology and syntactic description*, vol. 2, pp. 151–205, 2007.

[21] R. Snow, B. O'connor, D. Jurafsky, and A. Y. Ng, "Cheap and fast–but is it good? evaluating non-expert annotations for natural language tasks," in *Proceedings of the 2008 conference on empirical methods in natural language processing*, 2008, pp. 254–263.

[22] Y. Vasiliev, *Natural language processing with Python and spaCy: A practical introduction*. No Starch Press, 2020.

[23] K. Chowdhary and K. Chowdhary, "Natural language processing," *Fundamentals of artificial intelligence*, pp. 603–649, 2020.

[24] P. M. Nadkarni, L. Ohno-Machado, and W. W. Chapman, "Natural language processing: an introduction," *Journal of the American Medical Informatics Association*, vol. 18, no. 5, pp. 544–551, 2011.

[25] R. Liu, R. Mao, A. T. Luu, and E. Cambria, "A brief survey on recent advances in coreference resolution," *Artificial Intelligence Review*, vol. 56, no. 12, pp. 14 439–14 481, 2023.

[26] N. Stylianou and I. Vlahavas, "A neural entity coreference resolution review," *Expert Systems with Applications*, vol. 168, p. 114466, 2021.

[27] X. Wei, X. Cui, N. Cheng, X. Wang, X. Zhang, S. Huang, P. Xie, J. Xu, Y. Chen, M. Zhang *et al.*, "Zero-shot information extraction via chatting with chatgpt," *arXiv preprint arXiv:2302.10205*, 2023.

[28] J. Zhao, N. Xue, and B. Min, "Cross-document event coreference resolution: Instruct humans or instruct gpt?" in *Proceedings of the 27th Conference on Computational Natural Language Learning (CoNLL)*, 2023, pp. 561–574.

[29] B. Li, G. Fang, Y. Yang, Q. Wang, W. Ye, W. Zhao, and S. Zhang, "Evaluating chatgpt's information extraction capabilities: An assessment of performance, explainability, calibration, and faithfulness," *arXiv preprint arXiv:2304.11633*, 2023.

[30] S. Ekin, "Prompt engineering for chatgpt: a quick guide to techniques, tips, and best practices," *Authorea Preprints*, 2023.

[31] J. Li, A. Sun, J. Han, and C. Li, "A survey on deep learning for named entity recognition," *IEEE transactions on knowledge and data engineering*, vol. 34, no. 1, pp. 50–70, 2020.

[32] C.-Y. Lin and F. Och, "Looking for a few good metrics: Rouge and its evaluation," in *Ntcir workshop*, 2004.

[33] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "Bertscore: Evaluating text generation with bert," *arXiv preprint arXiv:1904.09675*, 2019.

[34] B. Lin, S. Wang, Z. Liu, Y. Liu, X. Xia, and X. Mao, "Cct5: A code-change-oriented pre-trained model," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1509–1521.

[35] K. Liu, G. Yang, X. Chen, and C. Yu, "Sotitle: A transformer-based post title generation approach for stack overflow," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 577–588.

[36] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.

[37] G. Uddin and M. P. Robillard, "How api documentation fails," *Ieee software*, vol. 32, no. 4, pp. 68–75, 2015.

[38] M. P. Robillard and R. DeLine, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, pp. 703–732, 2011.

[39] W. Maalej and M. P. Robillard, "Patterns of knowledge in api reference documentation," *IEEE Transactions on software Engineering*, vol. 39, no. 9, pp. 1264–1282, 2013.

[40] C. Chen, Z. Xing, and Y. Liu, "What's spain's paris? mining analogical libraries from q&a discussions," *Empirical Software Engineering*, vol. 24, pp. 1155–1194, 2019.

[41] Q. Shen, Y. Qian, Y. Zou, S. Wu, and B. Xie, "Fusing code and documents to mine software functional features," *Journal of Software*, vol. 32, no. 4, pp. 1023–1038, 2021.

[42] R. Khojah, M. Mohamad, P. Leitner, and F. G. de Oliveira Neto, "Beyond code generation: An observational study of chatgpt usage in software engineering practice," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1819–1840, 2024.

[43] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[44] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt," in *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 2023, pp. 23–30.

[45] M. Watanabe, Y. Kashiwa, B. Lin, T. Hirao, K. Yamaguchi, and H. Iida, "On the use of chatgpt for code review: Do developers like reviews by chatgpt?" in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 375–380.

[46] K. Yang, X. Mao, S. Wang, T. Zhang, B. Lin, Y. Wang, Y. Qin, Z. Zhang, and X. Mao, "Enhancing code intelligence tasks with chatgpt," *arXiv preprint arXiv:2312.15202*, 2023.

[47] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "Chatunitest: A framework for llm-based test generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 572–576.

[48] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1646–1656.

[49] J. Li, L. Li, J. Liu, X. Yu, X. Liu, and J. W. Keung, "Large language model chatgpt versus small deep learning models for self-admitted technical debt detection: Why not together?" *Software: Practice and Experience*.

[50] C. Arora, J. Grundy, and M. Abdelrazek, "Advancing requirements engineering through generative ai: Assessing the role of llms," in *Generative AI for Effective Software Development*. Springer, 2024, pp. 129–148.

[51] Q. Huang, Y. Sun, Z. Xing, M. Yu, X. Xu, and Q. Lu, "Api entity and relation joint extraction from text via dynamic prompt-tuned language model," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–25, 2023.