CKTyper: Enhancing Type Inference for Java Code Snippets by Leveraging Crowdsourcing Knowledge in Stack Overflow

ANJI LI, Sun Yat-sen University, China
NENG ZHANG*, Central China Normal University, China
YING ZOU, Queen's University, Canada
ZHIXIANG CHEN, Sun Yat-sen University, China
JIAN WANG, Wuhan University, China
ZIBIN ZHENG, Sun Yat-sen University, China

Code snippets are widely used in technical forums to demonstrate solutions to programming problems. They can be leveraged by developers to accelerate problem-solving. However, code snippets often lack concrete types of the APIs used in them, which impedes their understanding and resue. To enhance the description of a code snippet, a number of approaches are proposed to infer the types of APIs. Although existing approaches can achieve good performance, their performance is limited by ignoring other information outside the input code snippet (e.g., the descriptions of similar code snippets) that could potentially improve the performance.

In this paper, we propose a novel type inference approach, named CKTyper, by leveraging crowdsourcing knowledge in technical posts. The key idea is to generate a relevant context for a target code snippet from the posts containing similar code snippets and then employ the context to promote the type inference with large language models (e.g., ChatGPT). More specifically, we build a crowdsourcing knowledge base (CKB) by extracting code snippets from a large set of posts and index the CKB using Lucene. An API type dictionary is also built from a set of API libraries. Given a code snippet to be inferred, we first retrieve a list of similar code snippets from the indexed CKB. Then, we generate a crowdsourcing knowledge context (CKC) by extracting and summarizing useful content (e.g., API-related sentences) in the posts that contain the similar code snippets. The CKC is subsequently used to improve the type inference of ChatGPT on the input code snippet. The hallucination of ChatGPT is eliminated by employing the API type dictionary. Evaluation results on two open-source datasets demonstrate the effectiveness and efficiency of CKTyper. CKTyper achieves the optimal precision/recall of 97.80% and 95.54% on both datasets, respectively, significantly outperforming three state-of-the-art baselines and ChatGPT.

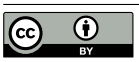
CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Additional Key Words and Phrases: Type Inference, Crowdsourcing Knowledge, ChatGPT, Stack Overflow

ACM Reference Format:
Anji Li, Neng Zhang, Ying Zou, Zhixiang Chen, Jian Wang, and Zibin Zheng. 2025. CKTyper: Enhancing Type Inference for Java Code Snippets by Leveraging Crowdsourcing Knowledge in Stack Overflow. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE009 (July 2025), 21 pages. https://doi.org/10.1145/3715724

*Corresponding author.

Authors' Contact Information: Anji Li, Sun Yat-sen University, Zhuhai, China, lianj8@mail2.sysu.edu.cn; Neng Zhang, Central China Normal University, Wuhan, China, nengzhang@ccnu.edu.cn; Ying Zou, Queen's University, Kingston, Canada, ying.zou@queensu.ca; Zhixiang Chen, Sun Yat-sen University, Zhuhai, China, chenzhx69@mail2.sysu.edu.cn; Jian Wang, Wuhan University, Wuhan, China, jianwang@whu.edu.cn; Zibin Zheng, Sun Yat-sen University, Wuhan, China, zhzibin@mail.sysu.edu.cn.



ACM 2994-970X/2025/7-ARTFSE009
https://doi.org/10.1145/3715724

1 Introduction

In technical forums, e.g., Stack Overflow[1] (SO) and Quora[2], code snippets (CSs) are widely used to illustrate programming problems or solutions [20]. In practice, developers often search for CSs to find solutions to their problems [1, 2, 19], which can save valuable time and efforts from learning extensive knowledge they have not yet grasped. Unfortunately, CSs generally do not provide sufficient code dependency information to determine the concrete types (i.e., fully qualified names (FQNs)) of APIs used in them [5, 25]. As a result, CSs usually cannot be compiled and reused directly [10, 29]. As reported by Terragni et al. [25], over 91% CSs on SO cannot be compiled due to various errors, among which the missing type declarations of APIs is the most common error that accounts for 38%. Therefore, inferring the types of APIs in CSs is a prerequisite step required in order to successfully use the CSs [7, 11, 21].

There are typically two categories of approaches proposed for inferring the types of APIs in CSs, including constraint-based [7, 23, 24] and statistically-based [11, 17, 21, 26]. Constraint-based approaches pre-build a knowledge base (KB) that contains potential API types from a set of API libraries. For a CS, they then extract type constraints (e.g., the methods invoked by a class) from the CS and employ heuristic rules to find the types of APIs in the KB that can satisfy the constraints to the maximal degree. Although these approaches could achieve high precision, they suffer from low recall because CSs often contain syntactic errors which may obstruct the extraction of type constraints. To reduce the impact of syntactic errors, statistically-based approaches build a statistical model by learning from a code corpus and then predict the types of APIs in a CS based on the model. As statistically-based approaches are hardly affected by syntactic errors, they could achieve relatively higher recall. However, statistically-based approaches generally do not consider type constraints in CSs, which may lead to low precision. To complement advantages of constraint- and statistically-based approaches, some hybrid type inference approaches [3, 15] are proposed by integrating both categories of approaches. The large language models (LLMs), e.g., ChatGPT, are also applied to type inference [16]. These hybrid and LLM-based approaches have achieved better performance. However, **the prior approaches generally concentrate on mining and employing the information available in the input CS (e.g., type constraints and token occurrences), but ignoring other useful information external to the CS (e.g., the descriptions of similar CSs in technical posts). Therefore, the prior approaches have limited performance in type inference.**

To address the limitation of existing type inference approaches, we propose a novel approach, named CKTyper, to enhance type inference by leveraging crowdsourcing knowledge available in technical posts. Our key idea is to extract relevant contextual information for an input CS from the descriptions of similar CSs in posts and then use the context to improve the type inference of a LLM (e.g., ChatGPT) on the input CS. CKTyper consists of an offline component and an online component. In the offline processing, we build a crowdsourcing knowledge base (CKB) of pairs between posts and CSs (also referred to as '*post-CS pairs*' for short) by collecting a set of technical posts and extracting CSs from each post. The CKB is then indexed using Lucene[3], an efficient index and search engine for text corpus. Moreover, we build an API type dictionary from a set of API libraries. In the online processing, for an input CS, we retrieve a list of similar CSs from the indexed CKB by combining the Lucene search engine and a code similarity measurement method. Then, we generate a crowdsourcing knowledge context (CKC) for the input CS by extracting and summarizing information (e.g., API-related sentences) from the posts that contain the similar CSs.

[1] https://stackoverflow.com/
[2] https://www.quora.com/
[3] https://lucene.apache.org/

```
1. try {
2.     InetAddress server = Inet4Address.getByName("thehost");
3.     if(server.isReachable(5000)){
4.         Log.d(TAG, "Ping!");
5.     }
6.     Socket clientsocket = new Socket(server, 8080);
7. } catch (UnknownHostException e) {
8.     Log.e(TAG, "Server Not Found");
9. } catch (IOException e) {
10.    Log.e(TAG, "Couldn't open socket");
11. }
```

class
method
field
variable

Fig. 1. Example CS with four kinds of APIs: class, method, field, and variable.

The CKC is then used to prompt the type inference of ChatGPT on the input CS. The fictional types generated due to the hallucination of ChatGPT are eliminated using the API type dictionary.

We implement a prototype of CKTyper for Java CSs based on the SO Q&A posts and a set of Java API libraries. Three state-of-the-art (SOTA) type inference approaches, including SnR [7] (a constraint-based method), MLMTyper [11] (a statistically-based method), and iJTyper (a hybrid approach), as well as the original ChatGPT are used as four baselines. We conduct experiments to evaluate CKTyper using two open-source datasets: StatType-SO [17] and Short-SO [11]. The evaluation results show that CKTyper achieves the highest average precision/recall[4] of 97.80% and 95.54% on both datasets, respectively, significantly outperforming the baselines by at least 0.49%-30.46%. Moreover, several ablation experiments validate the effectiveness of multiple mechanisms adopted in CKTyper, such as the CKC generation and fictional type removal. To sum up, the main contributions of this paper are as follows:

- We propose CKTyper, a novel type inference approach by leveraging crowdsourcing knowledge in technical posts. To our best knowledge, we are the first to report and utilize the external information available outside the input CS.
- We propose multiple mechanisms, such as two-phase similar CS retrieval, CKC generation, and fictional type removal, for improving the performance of CKTyper.
- We conduct extensive experiments to evaluate the effectiveness and efficiency of CKTyper in comparison with three SOTA baselines and ChatGPT.

The rest of the paper is organized as follows. Section 2 introduces the background of type inference and the motivation of CKTyper. Section 3 describes the details of CKTyper. Section 4 presents the experimental setup and results. Section 5 discusses the threats to validity of this work. Section 6 reviews the related work. Section 7 concludes the paper and discusses future work. Section 8 provides the release information of our replication package.

2 Preliminary and Motivation

In this section, we introduce the background of type inference. To motivate CKTyper, we demonstrate the efficacy of employing crowdsourcing knowledge for type inference.

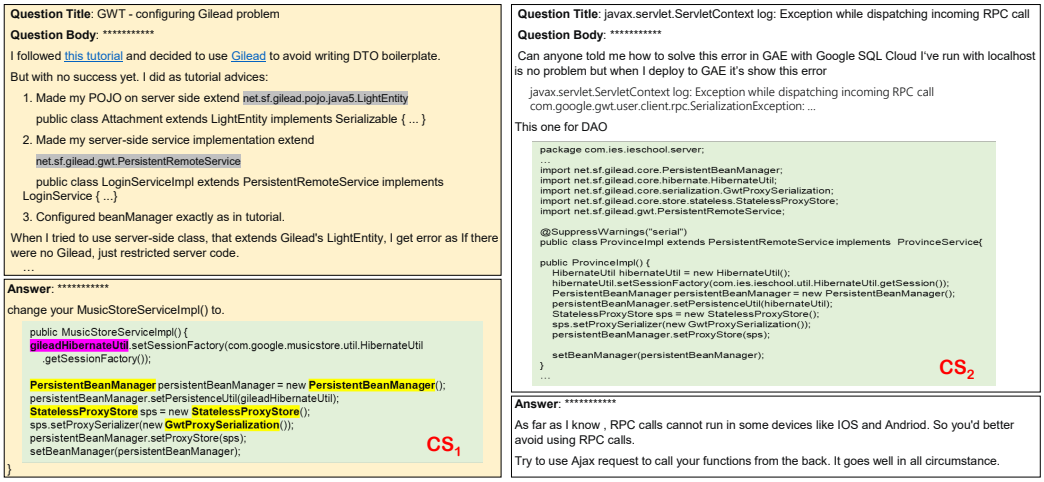


Fig. 2. (a) The SO question post '4702668' with an answer containing a CS, CS_1; and (b) The SO question post '12377211' (with an answer) that contains a CS, CS_2, similar to CS_1.

2.1 Background of Type Inference

The objective of type inference is to identify the concrete types of APIs in a CS. Generally, there are five kinds of APIs: *class*, *interface*, *method*, *field*, and *variable*. Class and interface are more important than the others since the types of classes and interfaces, along with their syntactic knowledge (e.g., the declared methods and fields as well as the inheritance and implementation relationships), can be used to infer the types of the other APIs. For example, Fig. 1 shows a CS with four kinds of APIs from the SO post '442496'. After determining the type of the class '*Inet4Address*' as '*java.net.Inet4Address*', the type of the method '*getByName("thehost")*' invoked by the class can be inferred as '*public static InetAddress getByName(String host) throws UnknownHostException*' by leveraging the syntactic knowledge of the class[5]. The type of the variable *server* can be further inferred based on the return type of the method. Note that the classes that correspond to primitive data types, e.g., *int* and *Integer*, are often excluded as such types are straightforward to recognize. **Therefore, we focus on inferring the types of classes and interfaces except those corresponding to primitive data types.** Moreover, the variables without specifying their associated classes/interfaces, e.g., the variable *gileadHibernateUtil* included in the CS_1 shown in Fig. 2(a), are also considered in this work.

2.2 Efficacy of Employing Crowdsourcing Knowledge for Type Inference

Existing type inference approaches have not fully utilized the rich information outside the target CS to improve the performance. Nowadays, online technical forums (e.g., the largest Q&A forum worldwide, SO) have accumulated tens of millions of technical posts. The posts contain a large number of CSs with descriptions and create a huge crowdsourcing knowledge base that could be used to enhance type inference for a wide range of CSs.

[4]For the approaches that can produce type inference results for all required APIs in a CS, such as MLMTyper, iJTyper, ChatGPT, and CKTyper, their respective precision and recall are the same, as explained in Section 4.
[5]https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/net/Inet4Address.html

Table 1. Type inference results produced for four APIs in CS_1 (see Fig. 2(a)). For each API, we present its ground-truth type and the three types inferred using ChatGPT based on three kinds of input, respectively, namely: 1) CS_1 only; 2) CS_1 with the textual descriptions in the answer post that contains CS_1 and the corresponding question post (see Fig. 2(a)); and 3) the input described in 2) as well as the textual description in the question post that contains CS_2 (see Fig. 2(b)). The notation ✓ in a cell means that the inferred type is correct, i.e., matched with the ground-truth type.

API	**Ground-truth type**	**Inferred type using CS_1 only**	**Inferred type using CS_1 with the textual descriptions in the Q&A posts in Fig. 2(a)**	**Inferred type using CS_1 with the textual descriptions in the Q&A posts in Fig. 2(a) and the question post in Fig. 2(b)**
gileadHibernateUtil	net.sf.gilead.core.hibernate.HibernateUtil	com.google.musicstore.util.GileadHibernateUtil	✓	✓
PersistentBeanManager	net.sf.gilead.core.PersistentBeanManager	com.google.musicstore.persistence.PersistentBeanManager	✓	✓
StatelessProxyStore	net.sf.gilead.core.store.stateless.StatelessProxyStore	com.google.musicstore.proxy.StatelessProxyStore	✓	✓
GwtProxySerialization	net.sf.gilead.core.serialization.GwtProxySerialization	com.google.musicstore.proxy.serialization.GwtProxySerialization	net.sf.gilead.gwt.GwtProxySerialization	✓

Fig. 2 shows two question posts '4702668'[6] and '12377211'[7] from SO along with one of their answers. The answer post of the first question post and the second question post respectively contain a CS, which are denoted as CS_1 and CS_2. Both CSs are similar to each other. We apply ChatGPT to infer the types of four APIs used in CS_1, i.e., *gileadHibernateUtil*, *PersistentBeanManager*, *StatelessProxyStore*, and *GwtProxySerialization*, using the parts 1 and 2 of the prompt template presented in Table 2. We focus on these APIs as their types are non-primitive types which contain additional information to understand the CS, and the types of the other APIs can be inferred based on the types or syntactic knowledge of the four APIs. For example, the variable *sps* is an object of the class *StatelessProxyStore* and thus it has the same type with that class. The type of the method *setSessionFactory* can be determined from the type of *gileadHibernateUtil*. To verify the efficacy of employing crowdsourcing knowledge for type inference, we further apply ChatGPT to perform the same type inference task above with two additional information, respectively, namely: 1) the textual descriptions in the answer post that contains CS_1 and the corresponding question post, as shown in Fig. 2(a); and 2) the textual descriptions described in 1) as well as the textual descriptions in the question post that contains a similar CS of CS_1, i.e., CS_2, as shown in Fig. 2(b). Note that the additional information is input as 'part 3' of the prompt template in Table 2.

Table 1 presents four primary APIs in CS_1, their ground-truth types, and the three types inferred for each of them by employing ChatGPT based on different kinds of input. ChatGPT fails to infer any correct type using CS_1 only. After supplmenting CS_1 with the textual descriptions in the answer post that contains CS_1 and the corresponding question post (see Fig. 2(a)), ChatGPT can correctly infer three API types. After appending more descriptions from the question post containing a similar CS of CS_1, ChatGPT successfully infers all the API types. Based on our analysis, the performance improvement of ChatGPT is because of the additional descriptions about the APIs in the posts. For example, from the descriptions "*I followed ... and decided to to use Gilead ...*" and "*Made my server-side service implementation extend net.sf.gilead.gwt.PersistentBeanManager*" in the question post shown in Fig. 2(a), it can be inferred that the APIs should belong to the package *net.sf.gilead* of the library *Gilead*. This helps ChatGPT determine the correct types of three APIs. However, ChatGPT cannot determine the correct type of *GwtProxySerialization* as it has multiple candidate types in the package, e.g., *net.sf.gilead.gwt.GwtProxySerialization* and *net.sf.gilead.core.serialization.GwtProxySerialization*.

[6]https://stackoverflow.com/questions/4702668/gwt-configuring-gilead-problem
[7]https://stackoverflow.com/questions/12377211/javax-servlet-servletcontext-log-exception-while-dispatching-incoming-rpc-call

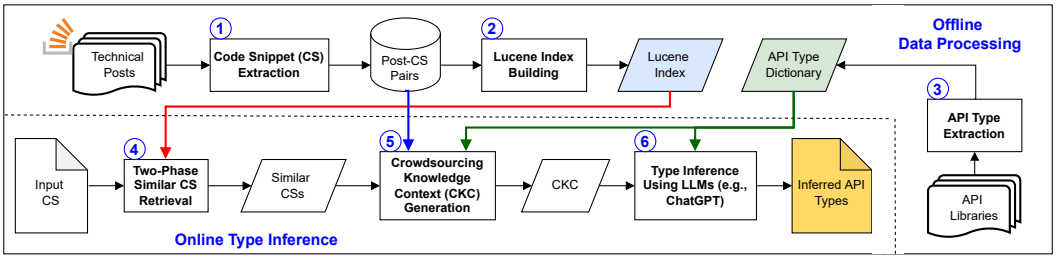


Fig. 3. The framework of CKTyper.

According to the source code of Gilead[8], *net.sf.gilead.core.serialization.GwtProxySerialization* is the real one. This issue is solved by utilizing the import statements in the similar CS, CS_2. **These results show that the crowdsourcing knowledge in the technical posts can be utilized to enhance the type inference task.**

3 Methodology

Fig. 3 shows the framework of CKTyper. The input is a code snippet (CS), and the output is the inferred types of APIs used in the CS. CKTyper consists of an offline data processing component (① - ③) and an online type inference component (④ - ⑥). In the offline component, we build a **crowdsourcing knowledge base (CKB)** by extracting CSs from a set of technical posts and then index the CKB (i.e., post-CS pairs) using Lucene. We also build a dictionary of API types from a set of API libraries. In the online component, we first retrieve a list of CSs similar to an input CS from the CKB based on a code similarity measurement method. A **crowdsourcing knowledge context (CKC)** is summarized from the posts containing the similar CSs. Finally, we use a LLM, e.g., ChatGPT, to infer the types of APIs in the input CS by designing a prompt with the CKC. The hallucination problem with the inferred types is addressed using the API type dictionary.

It is worth mentioning that **CKTyper is a general framework that can be applied to enhance the type inference of CSs written in different programming languages (PLs), e.g., Java and Python, by leveraging crowdsourcing knowledge.** For a specific PL, the application of CKTyper firstly requires to collect two kinds of data related to the PL, namely 1) a set of technical posts from programming forums or tutorials, e.g., SO and CodeProject[9]; and 2) a set of API libraries from API repositories, e.g., Maven[10] and PyPI[11]. The posts are used to construct a CKB of post-CS pairs for retrieving similar CSs of a CS and then generating a CKC from the corresponding posts to promote the type inference on the CS using a LLM. The API libraries are used to construct an API type dictionary for identifying API-related sentences in the posts of similar CSs (for CKC generation) and eliminating the fictional API types inferred by the LLM. Moreover, it requires to adjust the corresponding data processing steps, including the step of CS extraction from the posts by analyzing their formats (e.g., HTML or XML) and the step of API type extraction from the libraries using a specialized parser (e.g., JavaParser[12] or the `ast` module in Python). As a proof of concept, we implement CKTyper for Java CSs by leveraging Q&A posts from SO and Java API libraries from Maven.

[8]https://github.com/emsouza/gilead
[9]https://www.codeproject.com/
[10]https://mvnrepository.com/
[11]https://pypi.org/
[12]https://javaparser.org/

3.1 Offline Data Processing

3.1.1 Code Snippet (CS) Extraction from Technical Posts. We download the official SO data dump released on December 12, 2023. All the technical Q&A posts are included in an XML file, *Posts.xml*. We parse the XML file and collect 1,916,529 Q&A posts related to Java based on the tag 'java' and its synonymous tags (e.g., 'jdk' and 'jre') assigned to the question posts. Then, for each question or answer post, we extract the long CSs enclosed in the HTML tags `<pre><code>` from the body of the post, e.g., CS_1 and CS_2 shown in Fig. 2. If there are multiple CSs extracted from a post, we concatenate them together according to their occurrence order in the post. After extracting CSs from all posts, we build a CKB as a collection of post-CS pairs. Each pair consists of a post (which can be a question post or an answer post) and the concatenated CS from the post. The posts without CSs are excluded in the CKB; and the associations between question posts and answer posts are recorded in CKB. Specifically, the CKB can be represented as $CKB =< Ps, CSs, PCS, QA >$ where Ps represents the entire set of question or answer posts; CSs denotes the entire set of CSs; PCS includes the set of post-CS pairs, e.g., $< p, cs >$ (where p is a post, and cs is the CS extracted from p); and QA is the set of Q&A pairs, e.g., $< p_i, p_j >$ where p_i is a question post and p_j is an answer post of p_i. If the CKB is built from a set of technical posts (e.g., technical tutorials) where do not distinguish questions or answers, then the QA is null.

3.1.2 Lucene Index Building for CKB. To accelerate the retrieval of similar CSs for an input CS from a large-scale CKB during the online type inference process of CKTyper, we build an index for all post-CS pairs, i.e., *PCS*, in the CKB using Lucene. We first preprocess each CS in two steps: 1) *Comment Removal*: We remove the comments started with '//' or enclosed in '/* ... */' as they do not contribute to the functionality of the CS; and 2) *Tokenization*: We segment the CS into tokens using va software-specific tokenizer, S-NER [30]. S-NER perserves software-specifc entities, such as call chains of an API method (e.g., `Thread.sleep()`) and programming operators (e.g., '=='). For example, the second statement of CS_1 shown in Fig. 2(a) is segmented as: *gileadHibernateUtil.setSessionFactory*, *(*, *com.google.musicstore.util.HibernateUtil.getSessionFactory()*, *);*. The software-specific entities help better measure similarities between CSs and improve the performance of CKTyper (see Section 4.4).

We create a document for each preprocessed CS. There may exist duplicate CSs from multiple posts. To quickly locate the set of posts that contain a similar CS retrieved for an input CS (see Section 3.2.2), we append the identifiers of the posts containing the same preprocessed CS to the corresponding document. The identifier of a post can be obtained from the post, e.g., '4702668' of the SO question post shown in Fig. 2(a), or uniquely defined by ourselves. Next, we index all the CS documents using Lucene.

3.1.3 API Type Extraction from Libraries. We collect the jar files of 32 Java API libraries from Maven. These libraries are sufficient for the evaluation of CKTyper using the two open-source experimental datasets (see Section 4.1.1)[13]. We parse each jar using JavaParser and extract the types of classes and interfaces, resulting in an API type dictionary, denoted as $APITypeDict$. This dictionary is subsequently used for two purposes: 1) Identifying API-related sentences from the posts that contain similar CSs of a CS to generate a CKC for the CS (see Section 3.2.2); and 2) Eliminating the non-real API types inferred by a LLM, as described in Section 3.2.3.

[13]For the implementation of CKTyper, the set of API libraries should cover all APIs that need to be inferred in potential CSs. The 32 API libraries collected in this work are sufficient for our evaluation of CKTyper as they include the six API libraries to which the APIs used in the two experiment datasets, i.e., StatType-SO and Short-SO, belong.

3.2 Online Type Inference

As depicted in Fig. 3, for an input CS, the online type inference component of CKTyper contains three steps, which are described in the following three subsections.

3.2.1 Two-Phase Similar CS Retrieval. In this step, we retrieve a list of CSs similar to the input CS, referred to as *ics*, from the CKB built offline. Since there can be a large number (e.g., millions) of CSs in the CKB, it could be time-consuming to accurately measure the similarities between all existing CSs and *ics*.[14] We adopt a two-phase strategy for similar CS retrieval by leveraging the Lucene index built for the CKB (see Section 3.1.2) and a code similarity measurement method. **Our strategy can retrieve similar CSs accurately while ensuring a good efficiency.**

- **Phase 1.** We initially retrieve a list of top N (=1,000 by default) similar CSs for *ics*, denoted as $SimCS^{l}_{N}(ics)$ (where the superscript 'l' stands for Lucene), from the indexed CKB using the Lucene search engine. The similarity between a CS and *ics* is measured based on their common tokens using the BM25 algorithm [18].
- **Phase 2.** We measure the similarity between each $cs \in SimCS^{l}_{N}(ics)$ and *ics* using the SOTA code similarity measurement method, CrystalBLEU [8]. CrystalBLEU extends the BLEU [14] metric by removing trivially shared n-grams[15] between codes, and thus can precisely evaluate code similarity. After sorting the CSs in $SimCS^{l}_{N}(ics)$ by their CrystalBLEU similarities in descending order, we obtain a final list of top n (e.g., 10) similar CSs for *ics*, denoted as $SimCS^{c}_{n}(ics)$, where the superscript 'c' stands for CrystalBLEU.

As described above, after reducing the CKB to a limited set of N CSs in Phase 1 using the Lucene search engine, the efficiency of Phase 2 using CrystalBLEU can be guaranteed.

3.2.2 Crowdsourcing Knowledge Context (CKC) Generation. As demonstrated in the motivating example, enriching a CS with relevant crowdsourcing knowledge can improve the performance of LLMs like ChatGPT in inferring the types of APIs. Inspired by this, we generate a crowdsourcing knowledge context (CKC) for *ics* from the posts containing the top n similar CSs of *ics*, i.e., $SimCS^{c}_{n}(ics)$. At first, we collect the posts as $Ps(ics) = \bigcup_{cs_i \in SimCS^{c}_{n}(ics)} Ps(cs_i)$ where $Ps(cs_i)$ represents the set of posts containing cs_i. $Ps(cs_i)$ can be obtained from the CS document of cs_i (see Section 3.1.2). In Q&A forums like SO, an answer post is associated with the corresponding question post. If a similar CS is from an answer post, then the question post of the answer post, which can be obtained from the QA set of CKB, is also added to $Ps(ics)$.

The content of every post in $Ps(ics)$ consists of two parts: the CS and the text describing the question or answer. Moreover, the textual description may contain some sentences involving APIs. Two typical example posts can be found in Fig. 2. The title of a question post is also included as part of the textual description. Therefore, four types of data can be used for CKC generation, namely

- **Full Content (Full).** Using the full content of every post, including the CS and textual description.
- **CS Only (CS).** Applying the CS of every post only.
- **Textual Description Only (Desc).** Utilizing the textual description of every post only.
- **API-related Sentences Only (APISens).** Employing the API-related sentences in the textual description of every post only. We extract API-related sentences in two steps: 1) *Sentence Segmentation*: We segment the textual description to sentences using the `sent_tokenize`

[14] According to our experiments, it takes 14.79s to calculate CrystalBLEU similarities between 50,000 CSs and an input CS.
[15] The trivially shared n-grams are those frequently appeared in codes written in a programming language, e.g., 'for (' and 'System.out.println(' in Java codes.

Table 2. The prompt template designed for instructing ChatGPT to infer the types of a specified list of APIs used in a CS with the generated CKC.

Part	Content
1	For the given code snippet, infer the concrete types (i.e., fully qualified names (FQNs)) of the specified list of APIs used in it. Please recommand the top [k] most possible FQNs for each API and output all results in JSON format like: { 'API1': ['FQN1', 'FQN2', ...], 'API2': ['FQN1', 'FQN2', ...], ... }.
2	— Code snippet: [code snippet] — APIs whose FQNs need to be inferred: ['API1', 'API2', ...]
3	The following information is related to the code snippet and can be used to infer the FQNs of the given APIs: [CKC generated for the code snippet]

module of the NLTK toolkit[16]; and 2) *API-related Sentence Identification*: We identify API-related sentences from the set of segmented sentences based on the API type dictionary, i.e., $APITypeDict$, built offline (see Section 3.1.3). Since the APIs mentioned in textual descriptions are often simple names, e.g., *LightEntity* shown in Fig. 2(a), instead of their FQNs, e.g., *net.sf.gilead.pojo.java5.LightEntity*, we adopt a fuzzy API matching strategy. If a sentence contains the simple name of an API type in $APITypeDict$, then the sentence is identified as an API-related sentence.

We collect each type of data from all posts in $Ps(ics)$ to create a dataset for CKC generation. For simplicity, the collected data can be directly used as a CKC, which may also help improve the type inference of a LLM on *ics*, as shown in the motivating example. However, this simple strategy may lead to some issues. Specifically, there can be a relatively large amount of collected data, which may exceed the length of context window size that can be handled by the LLM or incur unexpected expenses of time and costs required to invoke a paid LLM's APIs (e.g., GPT-4). Moreover, there can be some noises (e.g., irrelevant pieces of code or text) in the collected data, which may negatively affect the type inference. To address these issues, we propose to generate a concise CKC by summarizing the collected data. We use the SOTA text summarization algorithm, PEGASUS [31]. PEGASUS has a key parameter δ (=0.3 by default) to specify the ratio of the input text to be summarized. In the default implementation of CKTyper, the collected set of API-related sentences is used as the input of PEGASUS for CKC generation, according to the evaluation results presented in Tables 5 and 6.

3.2.3 Type Inference Using A LLM. Based on the CKC generated for the input CS, *ics*, a LLM is used by CKTyper to infer the types of APIs in *ics*. As ChatGPT has been widely evaluated and used in practice, we apply ChatGPT for CKTyper. We design a prompt template to instruct ChatGPT to infer the types of a specified list of APIs in a CS by leveraging the CKC generated for the CS. As presented in Table 2, the prompt template contains three parts:

- **Part 1. (Task Instruction)** We instruct ChatGPT to understand the requirements of the type inference task, including the input and output format. For a given CS and a specified list of APIs used in the CS, we require ChatGPT to recommand the top k (e.g., 3) possible types for each API and output the type inference results in JSON format.

[16] https://www.nltk.org/

- **Part 2. (CS & APIs Input)** We provide the CS and a specified list of APIs whose types need to be inferred. By adjusting the specified APIs, we can infer the types of all APIs or a subset of interested APIs (e.g., the classes and interfaces belonging to the six API libraries in our experiments) in the CS. The APIs in a CS are obtained using the programming rules of Java and regular expressions.
- **Part 3. (CKC Input)** We offer the CKC generated for the input CS and notify ChatGPT to use the CKC for type inference.

Due to the hallucination nature of LLMs (e.g., ChatGPT), some API types recommanded by ChatGPT do not exist, as demonstrated in the motivating example (see Section 2.2). We solve this problem by filtering out the inferred types that are not included in the API type dictionary, $APITypeDict$. Notice that if there is no recommended type left for an API after removing the hallucinations, CKTyper will instruct ChatGPT to infer the types again until the recommendation list is not empty or two consecutive recommendation lists are both empty. To avoid removing the real API types by mistake, we suggest building a comprephensive API type dictionary from a large set of API libraries. When we aim to infer the types of APIs from a specific set of API libraries, it is necessary to build the dictionary using those libraries.

4 Evaluation

In this section, we evaluate the proposed CKTyper by answering four research questions (RQs):

RQ1: What are the settings of the three key parameters n, k, and δ, in CKTyper to achieve the best performance?

RQ2: How effective is CKTyper in comparison with SOTA type inference approaches?

RQ3: How well can the multiple mechanisms (e.g., CKC generation) adopted in CKTyper contribute to type inference?

RQ4: How efficient is CKTyper in type inference?

Our experimental environment is a workstation with an Intel(R) Xeon(R) Silver 4210R CPU (@2.40GHz) and an NVIDIA GeForce RTX 3090 GPU, running Ubuntu 20.04.3 LTS.

4.1 Experimental Setup

4.1.1 Dataset. To evaluate CKTyper, we use two open-source datasets of Java CSs collected from SO: StatType-SO [17] and Short-SO [11], which have been widely used for evaluation in prior studies [7, 11, 17, 21, 26]. Specifically, StatType-SO contains 268 CSs involving 4,086 APIs; and Short-SO contains 120 CSs involving 525 APIs. The APIs used in each CS of both datasets are primarily from one of the six popular Java/Android API libraries: Android, GWT, Hibernate, JDK, Joda Time, and XStream. The main difference between the two datasets is that all CSs in Short-SO contain less than three lines of code, whereas the CSs in StatType-SO are longer with an average of 28 lines of code. It is worth mentioning that we take the APIs belonging to these six libraries as the target of type inference. As described in Section 3.1.3, our API type dictionary are built from 32 libraries. The additional libraries could help infer the contextual APIs around the target APIs, thus improving type inference on the target ones (see Section 4.4).

4.1.2 Baselines. As described previously, there are three main categories of existing type inference approaches, i.e., constraint-based, statistically-based, and hybrid. To evaluate CKTyper, we select one SOTA approach from each category of approaches:

- **SnR** [7] (a constraint-based approach) pre-builds a knowledge base (KB) that contains the APIs and their inheritance /implementation relationships from the six libraries above. Given a CS, SnR first tries to repair it to a syntactically valid compilation unit using a template-based method and then extracts type constraints from the Abstract Syntax Tree (AST) generated

for the unit. The constraints are represented using a declarative logic programming language, Datalog [12]. Based on the pre-built KB and extracted constraints, SnR infers the types of APIs in the CS using heuristic rules, e.g., minimizing the number of unique libraries used in the CS. We implement SnR based on its replication package[17].

- **MLMTyper** [11] (a statistically-based approach) transforms the type inference task into a cloze-style fill-in-blank problem and uses a masked language model (MLM) to complete the missing types. For a specific API, e, in a CS, after building a code context block (including the code line where e is located and the top and down η (=2 by default) code lines), MLMTyper predicts the top k (e.g., 3) candidate types of e with the known types of all the other APIs in the context. We implement MLMTyper using its replication package[18].
- **iJTyper** [3] (a hybrid approach) integrates the complementary advantages of SnR and MLMTyper. For a CS, iJTyper first infers the types of APIs using SnR and arguments the CS with the inferred types. It then predicts the top k types of APIs for the argumented CS using MLMTyper. The predicted types are in turn used to refine the pre-built KB of SnR by filtering out irrelevant types, thus improve the performance of SnR. The improved type inference results of SnR are again used to generate a new argumented CS and promote MLMTyper. iJTyper iteratively performs the two approaches until the inference results are stable. The final inference results are produced by combining the results of both approaches. We implement iJTyper based on its replication package[19].

Apart from the three baselines, we also compare CKTyper with ChatGPT as it has been used for type inference. We choose the version of ChatGPT implemented based on the GPT-3.5 (gpt-3.5-turbo-0125) model. The hyperparameters of ChatGPT are set by default according to the API documentation[20]. The prompt template given in Table 2 without the part 3 of CKC input is used to guide ChatGPT to infer the types of APIs in a CS.

4.1.3 Metrics. To measure the performance of CKTyper and the baselines, we adopt two widely used metrics: precision and recall. For a CS, precision measures the percentage of the API types correctly inferred by an approach; and recall measures the percentage of the APIs whose types are correctly inferred by an approach.

$$Precision = \frac{\#\text{Correctly Inferred Types}}{\#\text{Inferred Types}} \tag{1}$$

$$Recall = \frac{\#\text{Correctly Inferred Types}}{\#\text{Requested Types}} \tag{2}$$

where '#Requested Types' represents the number of APIs whose types are requested to be inferred. **For the four approaches except SnR that can infer the types of all requested APIs in a CS, their respective precision and recall are the same.** Since SnR can only recommend one type for an API at most, for a fair comparison, we also only consider the top 1 type recommended for each API by the other approaches when measuring their precision and recall on a CS.

For each of the two datasets, StatType-SO and Short-SO, after measuring the precision and recall of an approach, *app*, on every CS, we calculate the average precision and recall of *app* on the set of CSs with respect to each of the six libraries as well as the average precision and recall of *app* on all CSs. Therefore, the average precision and recall of the approaches that can infer the types of all required APIs are also the same.

[17] https://doi.org/10.5281/zenodo.5843327
[18] https://github.com/SE-qinghuang/ASE-22-TypeInference
[19] https://anonymous.4open.science/r/iJTyper-0A4D
[20] https://platform.openai.com/docs/api-reference/chat

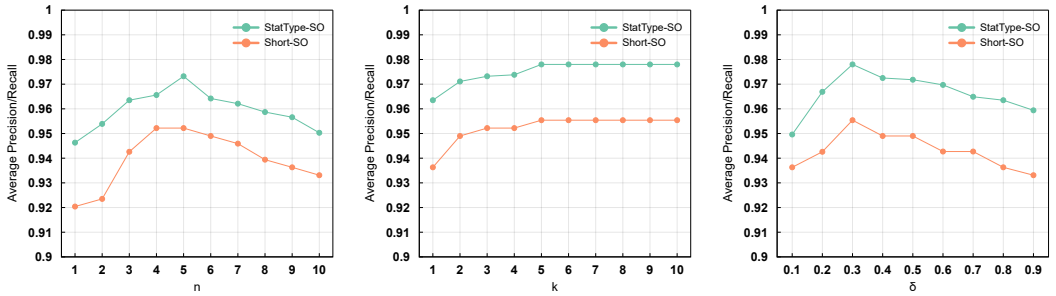


Fig. 4. Average performance of CKTyper with different settings of the three parameters: n, k, and δ.

4.2 RQ1: What are the settings of the three key parameters n, k, and δ, in CKTyper to achieve the best performance?

Motivation. As described in Section 3.2, there are three key parameters in CKTyper, namely 1) n which is the number of similar CSs retrieved for an input CS; 2) k which is the number of candidate types recommended for each API by ChatGPT; and 3) δ which is the summarization ratio used by PEGASUS for CKC generation. These parameters have impacts on the performance of CKTyper. Specifically, a small n may exclude some similar CSs and thus neglect useful information in the corresponding posts for CKC generation, which may limit the improvement of CKTyper. A large n may include dissimilar CSs and generate CKC with noises from the corresponding posts, which may also reduce the performance of CKTyper. Moreover, a large k will guide ChatGPT to recommend more candidate types for each API and increase the probability of recommending the correct type, thus promoting the performance of CKTyper. Meanwhile, a large k can increase the expenses of time, cost, and computational resources required to invoke the ChatGPT APIs. A large value of δ may lead to a lengthy CKC with noises, while a small value may discard some useful information in the posts. Both cases may negatively affect the performance of CKTyper. Therefore, it is necessary to determine the optimal settings of these parameters to facilitate the use of CKTyper.

Approach. We first run CKTyper on each CS in StatType-SO and Short-SO by varying n from 1 to 10 with a step of 1 while fixing k=3 and δ=0.3. For each setting of n, we measure the average precision/recall of CKTyper on all CSs in each dataset. After determining the proper n, we run CKTyper on each CS with different settings of k from 1 to 10 as well as the proper n and δ=0.3. We also measure the average precision/recall of CKTyper achieved under each setting of k and identify the proper k. Finally, we run CKTyper on each CS by setting δ from 0.1 to 0.9 with a step of 0.1, along with the proper n and k. The proper δ is also determined based on the average performance achieved using each setting of δ.

Results. Fig. 4 shows the average performance of CKTyper achieved with different settings of the three parameters on two datasets. We have the following findings:

- As n increases from 1 to 5, the average precision/recall of CKTyper increases at first and reaches peak values of 97.32% and 95.22% on StatType-SO and Short-SO, respectively, when n=5. After that, the performance of CKTyper decreases as n increases from 5 to 10. This result confirms that there exists a threshold for the number of similar CSs whose crowdsourcing knowledge can maximize the performance of CKTyper. A smaller or larger n will either miss some useful knowledge in similar CSs or introduce noises from dissimilar CSs, thus reducing the performance of CKTyper.
- As k increases from 1 to 10, the average precision/recall of CKTyper first increases, then reaches the highest points (with values of 97.80% and 95.54% on StatType-SO and Short-SO,

respectively) when k=5, and thereafter becomes stable. This result validates that more correct API types are discovered by requiring ChatGPT to recommend more candidate types for each API; and all the correct types that can be recommended by ChatGPT are included in the top 5 candidates.

- As δ increases from 0.1 to 0.9, the average precision/recall of CKTyper exhabits a similar trend to that achieved with different settings of n. The peak values of 97.80% and 95.54% on both datasets are reached when δ=0.3. This result means that under the setting of δ=0.3, the CKC generated by PEGASUS can concisely represent useful crowdsourcing knowledge in the posts of similar CSs for enhancing the type inference task. Either a smaller or a larger setting of δ reduces the performance of CKTyper due to the disposal of some useful knowledge or the reservation of some noises in the posts.

According to our findings, **it is suggested to set the three parameters as n=5, k=5, and δ=0.3**, which can ensure a good performance of CKTyper.

4.3 RQ2: How effective is CKTyper in comparison with SOTA approaches?

Motivation. The objective of CKTyper is to improve the performance in type inference by leveraging crowdsourcing knowledge from technical posts. As a proof of concept, we implement a prototype of CKTyper for Java CSs based on the Q&A posts from SO. We aim to evaluate whether CKTyper could achieve better performance than existing SOTA type inference approaches and the popular ChatGPT which has been used for type inference.

Approach. As described in Section 4.1.2, we select three SOTA type inference approaches, i.e., SnR, MLMTyper, and iJTyper, as baselines. We implement them using their replication packages. Moreover, we choose the version of ChatGPT implement using GPT-3.5 as another baseline. We apply CKTyper and the baselines to each CS in the two experiment datasets. Then, we measure the average precision and recall of the approaches on the CSs related to each of the six API libraries as well as each dataset. Furthermore, we test the statistical significance of the performance difference between CKTyper and each baseline using the Wilcoxon signed-rank test [28], a statistical hypothesis testing method used for comparing two matched samples.

Results. Tables 3 and 4 present the average performance of CKTyper and four baselines on the two datasets, respectively. **The performance order of these approaches is CKTyper > iJTyper > ChatGPT > SnR > MLMTyper. CKTyper significantly outperforms all the baselines in most cases.** More specifically, CKTyper improves the average precision/recall of iJTyper, ChatGPT, and MLMTyper by at least 0.49%, 4.89%, and 30.46%, respectively, on both datasets. CKTyper also improves the average precision and recall of SnR by at least 1.37% and 7.8%, respectively. These results confirm the effectiveness of employing crowdsourcing knowledge for type inference.

The worst performance of MLMTyper is because there are a considerable number of API types incorrectly generated by the underlying masked language model. Although SnR achieves a higher precision than MLMTyper, its recall is a bit low (no more than 90%) due to several factors, such as the incorrect extraction of complex type constraints (e.g., method call chains) and the insufficient constraints in the input CS. The performance of iJTyper is close to that of CKTyper on StatType-SO. However, the improvement of CKTyper over iJTyper is statistically significant. On Short-SO, CKTyper improves iJTyper by 3.02%. The relatively good performance of iJTyper mainly due to the integration of SnR and MLMTyper. However, since iJTyper does not employ external knowledge outside the input CS, its performance is limited, especially on the CSs with insufficient information, e.g., those in Short-SO. Although CKTyper achieves the optimal performance on both datasets, the performance on the CSs belonging to particular API libraries, e.g., XStream and Hibernate, is lower. By tracking the result of each online step, there is no similar CSs retrieved for a number of

Table 3. Average performance of five approaches on StatType-SO. In each cell of SnR, the value with 'P:' is precision, and the value with 'R:' is recall. For the other approaches, the value in each cell is both precision and recall as they are the same.

Approach	StatType-SO						
	Android	GWT	Hibernate	Joda-Time	JDK	XStream	Average
MLMTyper	71.91%	78.71%	42.30%	59.80%	86.87%	55.78%	65.90%***
SnR	P: 98.36% R: 92.59%	P: 97.48% R: 86.55%	P: 96.35% R: 95.08%	P: 94.83% R: 82.91%	P:100% R: 92.42%	P: 91.53% R: 90.44%	P: 96.43%*** R: 90.00%***
ChatGPT	97.17%	96.04%	90.84%	94.29%	**100%**	80.36%	92.91%***
iJTyper	97.22%	98.89%	93.77%	95.58%	**100%**	**98.01%**	97.31%*
CKTyper	**99.30%**	**99.09%**	**96.27%**	**99.43%**	**100%**	93.30%	**97.80%**
***: p<0.001, **: p<0.01, *: p<0.05							

Table 4. Average performance of five approaches on Short-SO. The meanings of the cell values refer to Table 3.

Approach	Short-SO						
	Android	GWT	Hibernate	Joda-Time	JDK	XStream	Average
MLMTyper	69.09%	68.89%	38.60%	66.67%	90.74%	66.47%	65.08%***
SnR	P: 88.68% R: 85.45%	P: 87.80% R: 80.00%	P: 89.29% R: 87.72%	P: 94.59% R: 72.92%	P: 98.04% R: 92.59%	P: 92.77% R: 90.59%	P: 91.86%* R: 84.88%***
ChatGPT	**100%**	88.37%	73.68%	93.75%	97.87%	91.55%	90.45%**
iJTyper	89.09%	86.67%	**89.47%**	**100%**	98.15%	91.76%	92.52%
CKTyper	**100%**	**97.67%**	84.21%	95.83%	**100%**	**97.18%**	**95.54%**
***: p<0.001, **: p<0.01, *: p<0.05							

CSs from our CKB. Consequently, the CKCs generated for such CSs contain a lot of noises, thus reducing the performance of CKTyper.

4.4 RQ3: How well can the multiple mechanisms (e.g., CKC generation) adopted in CKTyper contribute to type inference?

Motivation. There are multiple mechanisms adopted in CKTyper: 1) *S-NER* which is a software-specific tokenizer used to segment CSs for building their Lucene index and may help better retrieve the top N similar CSs for an input CS, *ics*, using Lucene; 2) *CrystalBLEU* which is a code similarity measurement method used to retrieve the final top n similar CSs for *ics*; and 3) *CKC generation (CKCG)* which generates a concise CKC from the posts of the top n similar CSs using PEGASUS; and 4) *Fictional type removal (FTR)* which removes the non-real API types recommended by ChatGPT based on the API type dictionary. Particularly, in the CKC generation mechanism, four types of data in the posts of similar CSs, namely *Full*, *CS*, *Desc*, and *APISens* (see Section 3.2.2), can be used as the input of PEGASUS. We need to validate whether these mechanisms contribute to better performance of CKTyper in type inference.

Approach. We implement nine variants of CKTyper, i.e.,

- **CKTyper-S-NER** replaces S-NER with the Stanford Tokenizer[21] designed for general English texts. For example, the second statement of CS_1 shown in Fig. 2(a) is segmented as: *gileadHibernateUtil, ., setSessionFactory, (, com, ., google, ., musicstore, ., util, ., HibernateUtil, ., getSessionFactory, (,),), ;* using the Stanford Tokenizer.
- **CKTyper-CrystalBLEU** removes CrystalBLEU from CKTyper. The top n (e.g., 5) similar CSs of an input CS are directly retrieved using the Lucene search engine.

[21] https://nlp.stanford.edu/software/tokenizer.html

Table 5. Average precision/recall (which are the same) of nine CKTyper variants on StatType-SO.

Approach	StatType-SO						
	Android	GWT	Hibernate	Joda Time	JDK	Xstream	Average
CKTyper-S-NER	99.30%	**99.09%**	94.92%	98.86%	**100%**	90.63%	97.04%
CKTyper-CrystalBLEU	97.89%	95.43%	93.22%	96.57%	**100%**	88.39%	94.98%
CKTyper-CKGG	98.94%	96.64%	94.23%	97.67%	**100%**	89.28%	95.93%
CKTyper-FTR	98.24%	98.48%	94.58%	99.43%	**100%**	91.07%	96.77%
CKTyper+CKCG(Full)	98.94%	95.73%	94.58%	96.58%	**100%**	87.50%	95.39%*
CKTyper+CKCG(CS)	98.59%	96.95%	91.86%	97.71%	**100%**	86.61%	95.04%**
CKTyper+CKCG(Desc)	**99.65%**	96.34%	95.59%	97.14%	**100%**	92.86%	96.77%
CKTyper+CKCG(ReduAPISens)	98.59%	97.26%	93.17%	**100%**	97.96%	94.20%	96.61%
CKTyper+CKCG(APISens)	99.30%	**99.09%**	**96.27%**	99.43%	**100%**	**93.30%**	**97.80%**

Table 6. Average precision/recall (which are the same) of nine CKTyper variants on Short-SO.

Approach	Short-SO						
	Android	GWT	Hibernate	Joda Time	JDK	Xstream	Average
CKTyper-S-NER	**100%**	**97.67%**	82.46%	93.75%	**100%**	95.77%	94.59%
CKTyper-CrystalBLEU	**100%**	93.02%	77.19%	95.83%	**100%**	97.18%	93.63%
CKTyper-CKGG	**100%**	93.02%	80.70%	**100%**	**100%**	95.77%	94.58%
CKTyper-FTR	**100%**	**97.67%**	75.44%	95.83%	**100%**	95.77%	93.63%
CKTyper+CKCG(Full)	**100%**	93.02%	78.95%	95.83%	**100%**	95.77%	93.63%
CKTyper+CKCG(CS)	**100%**	95.35%	77.19%	93.75%	**100%**	97.18%	93.63%
CKTyper+CKCG(Desc)	**100%**	93.02%	80.70%	**100%**	**100%**	**98.59%**	95.22%
CKTyper+CKCG(ReduAPISens)	**100%**	93.02%	83.64%	91.49%	100%	97.10%	94.17%
CKTyper+CKCG(APISens)	**100%**	**97.67%**	**84.21%**	95.83%	**100%**	97.18%	**95.54%**

- **CKTyper-CKCG** does not generate a concise CKC for an input CS using PEGASUS. It directly uses the data collected from the posts of similar CSs as a CKC.
- **CKTyper-FTR** does not filter out the fictional API types recommended by ChatGPT.
- **CKTyper+CKCG(Full)** generates the CKC for an input CS using the full content of the posts of similar CSs.
- **CKTyper+CKCG(CS)** generates the CKC for an input CS using the CSs in the posts of similar CSs.
- **CKTyper+CKCG(Desc)** generates the CKC for an input CS using the textual descriptions in the posts of similar CSs.
- **CKTyper+CKCG(APISens)** generates the CKC for an input CS using the API-related sentences from the posts of similar CSs. This variant is the default implementation of CKTyper.
- **CKTyper+CKCG(ReduAPISens)** generates the CKC for an input CS only using the sentences related to APIs in the six target libraries from the posts of similar CSs.

We apply the nine variants of CKTyper with the parameter settings determined in RQ1 to every CS in StatType-SO and Short-SO. Afterwards, we measure their average precision/recall with respect to each of the six API libraries and each of both datasets.

Results. Tables 5 and 6 present the average precision/recall of nine CKTyper variants achieved on the two datasets, respectively. The performance of the four variants that removes a mechanism or replaces a specialized mechanism with a general one, namely CKTyper-S-NER, CKTyper-CrystalBLEU, CKTyper-CKCG, and CKTyper-FTR, is worse than that of CKTyper (i.e., CKTyper+ CKCG(APISens)) by 0.76% (resp. 0.95%), 2.82% (resp. 1.91%), 1.87% (resp. 0.96%), and 1.03% (resp. 1.91%) on StatType-SO

Table 7. Time costs (in seconds) of five approaches.

Approach	Min	Max	Mean	Median	Std
SnR	6.01	17.83	10.70	10.62	3.29
MLMTyper	2.04	42.76	**9.31**	7.73	5.98
ChatGPT	1.17	11.04	**3.99**	5.53	2.72
iJTyper	22.83	291.50	79.01	73.36	45.07
CKTyper	6.06	16.77	**9.70**	11.09	**2.28**

(resp. Short-SO). The more the performance is reduced, the greater contribution of the corresponding mechanism. Therefore, **the result indicates that all the four mechanisms contribute to CKTyper. The contribution order of them is: CrystalBLEU ≥ CKCG <> FTR > S-NER**, where '<>' means '*could be less or greater than*'.

The performance order of the five variants that generate CKCs using different types of data in the posts of CSs similar to the input CS is: CKTyper+CKCG(APISens) > CKTyper+CKCG(Desc) > CKTyper+CKCG(ReduAPISens) > CKTyper+CKCG(Full) > CKTyper+CKCG(CS). This result implies that **the textual descriptions in the posts contain more useful knowledge than the CSs for CKCG. The sentences related to APIs (including both the target APIs from the six libraries and the contextual APIs from other libraries) in the textual descriptions have the highest value for CKCG.** The reasons could be explained as follows. In a post, the CS can be divided into two parts: 1) the piece of code similar to the input CS, *ics*, which could hardly provide additional information for inferring the API types used in *ics*; and 2) the rest piece of code, which often exhabits different functionalities implemented using APIs different from the target APIs used in *ics*. Generally, only a small percentage of statements involving the target APIs, e.g., the import statements of CS_2 shown in Fig. 2(b), are useful for CKCG. The co-occurrence between the target APIs and other APIs may also has kind of uselessness. In contrast, the textual descriptions in a post, especially the API-related sentences, explain the problem or functionality about the CS, thus they are useful for CKCG. However, descriptions irrelevant to the CS or target APIs may introduce noises in the generated CKC and therefore lead to the lower performance of CKTyper+CKCG(Desc) compared with CKTyper+CKCG(APISens).

4.5 RQ4: How efficient is CKTyper in type inference?

Motivation. As reported in RQ2, CKTyper achieves significantly better performance than SOTA baselines. However, we aim to systematically verify that the execution performance of CKTyper is good enough for practical use using the two open-source datasets.

Approach. We record the amount of time (in seconds) required by CKTyper and the baselines to infer the types of APIs in each CS in StatType-SO and Short-SO. From the results, we calculate various statistics for each approach, including the minimum (min), maximum (max), mean, median, and standard variation (std).

Results. Table 7 presents the statistics calculated from the time costs of five approaches. **In terms of average execution time, the performance order of the approaches is: ChatGPT (3.99s) > MLMTyper (9.31s) > CKTyper (9.70s) > SnR (10.70s) > iJTyper (79.01s)**, where > means 'is faster than'. Particularly, **the efficiency of CKTyper is the most stable as its time costs have the minimum std (i.e., 2.28).** The average execution time of CKTyper is lower than that of ChatGPT by 5.71s because we have three additional steps, e.g., similar CS retrieval and CKC generation, to perform before and after invoking ChatGPT to infer the types of APIs in a CS. However, this sacrifice of execution time should be acceptable in practice as CKTyper

Table 8. Average Precision/Recall of CKTyper with different LLMs on StatType-SO.

Approach	StatType-SO						
	Android	GWT	Hibernate	Joda Time	JDK	Xstream	Average
CKTyper(GPT-3.5)	**99.30%**	**99.09%**	96.27%	**99.43%**	**100%**	93.30%	97.80%
CKTyper(GPT-4)	**99.30%**	97.56%	**96.61%**	**99.43%**	**100%**	92.86%	97.45%
CKTyper(GPT-4o-mini)	**99.30%**	96.65%	95.59%	98.29%	**100%**	90.50%	96.55%
CKTyper(Claude)	**99.30%**	98.78%	95.93%	98.86%	**100%**	**96.43%**	**98.07%**
CKTyper(DeepSeek)	98.59%	96.65%	95.59%	96.57%	**100%**	91.89%	96.42%
CKTyper(Llama)	98.94%	96.34%	95.93%	98.29%	99.32%	94.62%	97.04%

Table 9. Average Precision/Recall of CKTyper with different LLMs on Short-SO.

Approach	Short-SO						
	Android	GWT	Hibernate	Joda Time	JDK	Xstream	Average
CKTyper(GPT-3.5)	**100%**	**97.67%**	**84.21%**	95.83%	**100%**	**97.18%**	**95.54%**
CKTyper(GPT-4)	**100%**	95.24%	78.95%	95.83%	**100%**	95.77%	93.93%
CKTyper(GPT-4o-mini)	**100%**	93.02%	78.95%	**100%**	**100%**	95.77%	94.27%
CKTyper(Claude)	**100%**	95.35%	82.46%	**100%**	**100%**	**97.18%**	**95.54%**
CKTyper(DeepSeek)	**100%**	90.70%	78.95%	**100%**	**100%**	92.96%	93.31%
CKTyper(Llama)	**100%**	93.02%	80.70%	95.83%	**100%**	92.96%	93.31%

can significantly improve the precision/recall of ChatGPT in type inference (see Tables 3 and 4). Although the execution performance of CKTyper is slightly lower than that of MLMTyper by 0.39s, the precision/recall of CKTyper is significantly much better than that of MLMTyper, i.e., 97.80% vs. 65.90% and 95.54% vs. 65.08% on both datasets. In addition, we can observe from Tables 3, 4, and 7 that the average precision/recall of iJTyper is close to that of CKTyper, i.e., 97.31% vs. 97.80% on StatType-SO and 92.52% vs. 95.54% on Short-SO, however the execution performance of iJTyper is much worse than that of CKTyper by 69.31s.

5 Discussion

5.1 Expansibility of CKTyper

CKTyper can achieve higher performance by implementing it using a better LLM. We re-implement CKTyper with five latest LLMs, namely ChatGPT-4 (gpt-4-turbo-2024-04-09), ChatGPT-4o-mini (gpt-4o-mini-2024-07-18), Llama (Llama-3.1-405b), DeepSeek (deepseek-chat), and Claude (claude-3-5-sonnet-20240620). After applying these variants of CKTyper to the CSs in StatType-SO and Short-SO, we measure their average precision/recall. As presented in Tables 8 and 9, the variant CKTyper(Claude) achieves the best performance on both datasets.

5.2 Generatability of CKTyper

To validate whether CKTyper can be applied to CSs written in other programming languages, we re-implement CKTyper for Python and apply the variant to a dataset of 150 Python CSs from a prior study [16]. Since Python's dynamic type system allows variables to share the same outmost type, e.g., *list* and *list[str]*, we use the *Match to Parametric (M2P)* metric proposed in [16] to simplify the evaluation of type inference results. The metric is defined by the ratio of type predictions that share the common outmost types with correct types.

The evaluation results show that CKTyper increases the top-5 M2P of ChatGPT (gpt-4o-mini-2024-07-18) from 80.73% to 84.25%, which demonstrates the generalizability of CKTyper. The reason

why the performance of CKTyper on Python CSs is lower than that on Java CSs might be because there are not enough similar CSs for the Python CSs in Stack Overflow, thus leading to noises in the generated CKCs and lower performance improvement in type inference. Specifically, the average similarities of the top-1 and top-5 CSs retrieved for Python CSs are 0.107 and 0.077, respectively, which are notably lower than those retrieved for Java CSs, i.e., 0.403 and 0.264.

5.3 Threats to validity

Threats to internal validity relate to the errors in the implementation of CKTyper, baselines, and the variants of CKTyper. We implement four baselines, i.e., SnR, MLMTyper, iJTyper, and ChatGPT, using their replication packages or official APIs. We test the re-implementation to avoid errors. For CKTyper and its variants, we have double-checked the implementation and tracked the inputs, intermediate results, and outputs of several CSs to ensure correctness.

Threats to external validity relate to the generalizability of our results. We conduct the experiments on two open-source datasets of Java CSs related to six popular API libraries to ensure the generalizability of our approach. Both datasets are widely used in prior studies [7, 11, 17, 21, 26]. The CSs in the two datasets are characterized by different lengths. And, the six libraries are used by various applications in different domains. Additionally, we re-implement CKTyper for Python and apply it to a dataset of Python CSs, which demonstrates the generalizability of CKTyper.

6 Related Work

Existing type inference studies can be categorized into three groups: constraint-based, statistical-based, and hybrid. We briefly discuss each category as follows.

Constraint-based. Subramanian et al. [24] propose Baker for Java and JavaScript. Based on an oracle of API types built from a large set of libraries, Baker maintains a candidate type list for each API in a CS and tries to shorten the list by leveraging type constraints extracted from the CS. Hassan et al. [9] propose TYPPETE for Python. It encodes the type constraints in a CS as a MaxSMT problem and solves the problem using the Z3 SMT solver [6]. Shokri et al. [22] introduce DepRes, which translates the type constraints in a CS into a type sketch and uses the Z3 SMT solver to determine the API types. Dong et al. [7] propose SnR for Java. It pre-builds a knowledge base (KB) of APIs from libraries. SnR tries to repair an input CS using a template-based method and then extracts type constraints from the generated AST. Finally, it infers the types of APIs based on the KB and constraints. These constraint-based methods require to extract type constraints from the input CS and thus are limited by the syntactic errors in CSs. Our CKTyper does not have this limitation by applying LLMs for type inference.

Statistically-based. Phan et al. [17] propose STATTYPE for Java, which learns frequent and simultaneous FQNs from a code corpus and predicts the types of APIs based on their type context and resolution context in a CS. Saifullah et al. [21] introduce COSTER, which infers the types of APIs based on an occurrence likelihood dictionary built from a code base. Malik et al. [13] propose NL2Type for JavaScript. It trains an LSTM model by leveraging function names and parameter names in a code corpus and then predicts the function types in a CS. Wei et al. [27] introduce LambdaNet for TypeScript, which generates a type dependency graph (TDG) for a CS and uses a graph neutral network to the predict types of APIs. Huang et al. [11] propose MLMTyper which transforms the type inference problem into a fill-in-blank task and trains a masked language model (MLM) from a code corpus to predict the types of APIs in a CS. Velázquez-Rodríguez et al. [26] propose RESICO for Java, which uses the Word2vec [4] model to vectorize APIs and their context and provides type recommendations using a machine learning classifier. These statistically-based methods can get rid of the syntactic errors in CSs, however they generally do not consider the type

constraints in a CS, which limits their performance. Our CKTyper compensates for this limitation by utilizing croudsourcing knowledge to enhance the performance of LLMs in type inference.

Hybrid. Constraint- and statistically-based methods can be integrated to complement each other and improve type inference [3, 15]. For example, Peng et al. [15] introduce HiTyper for Python, which hybrids deep learning-based and static type inference. It obtains type recommendations from a neural network and performs type rejection and static inference on a TDG built from a Python CS. Chen et al. [3] propose a flexible framework, named iJTyper, which can integrate constraint- and statistically-based methods without modifying their implementations.

Although hybrid methods have demonstrated better performance than constraint- and statistically-based methods, they share a common limitation, that is, they rarely utilize knowledge outside the input CS for type inference. We report this limitation for the first time and propose CKTyper to improve type inference by utilizing crowdsourcing knowledge in technical posts with LLMs.

7 Conclusion and Future Work

The existing type inference approaches have very limited use of rich knowledge external to the input code snippet (CS). In this paper, we propose CKTyper to enhance type inference by leveraging crowdsourcing knowledge from technical posts. As an offline processing, we build a crowdsourcing knowledge base by extracting CSs from technical posts and also build an API type dictionary from a set of libraries. Then, CKTyper infers the types of APIs in a CS by applying a large language model (e.g., ChatGPT) enhanced with a crowdsourcing knowledge context. The context is generated by summarizing relevant knowledge from the posts that contain similar CSs of the input CS. The fictional types recommended by ChatGPT are filtered out using the API type dictionary. We compare CKTyper with three SOTA baselines and ChatGPT on two open-source datasets. The results demonstrate the superiority of CKTyper over the baselines and ChatGPT. The accurate types of APIs inferred for CSs by CKTyper can reduce developers' efforts to understand the CSs and promote the reuse of CSs.

In future work, we plan to implement CKTyper for CSs written in other programming languages, such as Python and JavaScript. We will also develop a web service of CKTyper to faciliate its use for researchers and practitioners. Moreover, CKTyper can be extended with more external knowledge about CSs, e.g., the technical tutorials and blogs of similar CSs.

8 Data Availability

Our replication package of this work, including the implementation code and experimental dataset, is available at GitHub (https://github.com/LeeAnnJ/CKTyper).

Acknowledgments

This work is supported by the National Natural Science Foundation of China (62302536, 62032025) and the Guangdong Basic and Applied Basic Research Foundation (2023A1515012292).

References

[1] Alberto Bacchelli, Luca Ponzanelli, and Michele Lanza. 2012. Harnessing stack overflow for the ide. In *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, 26–30. doi:10.1109/RSSE.2012.6233404

[2] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. 2014. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical software engineering* 19 (2014), 619–654. doi:10.1007/s10664-012-9231-y

[3] Zhixiang Chen, Anji Li, Neng Zhang, Jianguo Chen, Yuan Huang, and Zibin Zheng. 2024. iJTyper: An Iterative Type Inference Framework for Java by Integrating Constraint- and Statistically-based Methods. arXiv:2402.09995 [cs.SE] https://arxiv.org/abs/2402.09995

[4] Kenneth Ward Church. 2017. Word2Vec. *Natural Language Engineering* 23, 1 (2017), 155–162. doi:10.1017/S1351324916000334

[5] Barthélémy Dagenais and Martin P Robillard. 2012. Recovering traceability links between an API and its learning resources. In *2012 34th international conference on software engineering (icse)*. IEEE, 47–57. doi:10.1109/ICSE.2012.6227207

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[7] Yiwen Dong, Tianxiao Gu, Yongqiang Tian, and Chengnian Sun. 2022. SnR: constraint-based type inference for incomplete Java code snippets. In *Proceedings of the 44th International Conference on Software Engineering*. 1982–1993. doi:10.1145/3510003.3510061

[8] Aryaz Eghbali and Michael Pradel. 2022. CrystalBLEU: precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12. doi:10.1145/3551349.3556903

[9] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-Based Type Inference for Python 3. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, 12–19. doi:10.1007/978-3-319-96142-2_2

[10] Eric Horton and Chris Parnin. 2018. Gistable: Evaluating the executability of python code snippets on github. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 217–227. doi:10.1109/ICSME.2018.00031

[11] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2022. Prompt-tuned code language model as a neural knowledge base for type inference in statically-typed partial code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13. doi:10.1145/3551349.3556912

[12] Magnus Madsen and Ondřej Lhoták. 2020. Fixpoints for the masses: programming with first-class datalog constraints. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28. doi:10.1145/3428193

[13] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 304–315. doi:10.1109/ICSE.2019.00045

[14] 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*. ACL, 311–318. doi:10.3115/1073083.1073135

[15] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static inference meets deep learning: a hybrid type inference approach for Python. In *Proceedings of the 44th International Conference on Software Engineering*. 2019–2030. doi:10.1145/3510003.3510038

[16] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R Lyu. 2023. Generative type inference for python. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 988–999. doi:10.1109/ASE56229.2023.00031

[17] Hung Phan, Hoan Anh Nguyen, Ngoc M Tran, Linh H Truong, Anh Tuan Nguyen, and Tien N Nguyen. 2018. Statistical learning of api fully qualified names in code snippets of online forums. In *Proceedings of the 40th International Conference on Software Engineering*. 632–642. doi:10.1145/3180155.3180230

[18] Stephen Robertson, Hugo Zaragoza, and Michael Taylor. 2004. Simple BM25 extension to multiple weighted fields. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*. 42–49. doi:10.1145/1031171.1031181

[19] Christoffer Rosen and Emad Shihab. 2016. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering* 21, 3 (2016), 1192–1223. doi:10.1007/s10664-015-9379-3

[20] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 191–201. doi:10.1145/2786805.2786855

[21] CM Khaled Saifullah, Muhammad Asaduzzaman, and Chanchal K Roy. 2019. Learning from examples to find fully qualified names of api elements in code snippets. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 243–254. doi:10.1145/2901739.2901767

[22] Ali Shokri. 2021. A program synthesis approach for adding architectural tactics to an existing code base. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1388–1390. doi:10.1145/3428193

[23] Ali Shokri and Mehdi Mirakhorli. 2021. Depres: A tool for resolving fully qualified names and their dependencies. *arXiv preprint arXiv:2108.01165* abs/2108.01165 (2021). arXiv:2108.01165 https://arxiv.org/abs/2108.01165

[24] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *Proceedings of the 36th international conference on software engineering*. 643–652. doi:10.1145/2568225.2568313

[25] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. 2016. CSNIPPEX: automated synthesis of compilable code snippets from Q&A sites. In *Proceedings of the 25th international symposium on software testing and analysis*. 118–129. doi:10.1145/2931037.2931058

[26] Camilo Velázquez-Rodríguez, Dario Di Nucci, and Coen De Roover. 2023. A text classification approach to API type resolution for incomplete code snippets. *Science of Computer Programming* 227 (2023), 102941. doi:10.1016/j.scico.2023.102941

[27] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *8th International Conference on Learning Representations, ICLR 2020*. OpenReview.net. https://openreview.net/forum?id=Hkx6hANtwH

[28] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in Statistics: Methodology and Distribution*. Springer, 196–202. doi:10.1007/978-1-4612-4380-9_16

[29] Di Yang, Aftab Hussain, and Cristina Videira Lopes. 2016. From query to usable code: an analysis of stack overflow code snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. Association for Computing Machinery, 391–402. doi:10.1145/2901739.2901767

[30] Deheng Ye, Zhenchang Xing, Chee Yong Foo, Zi Qun Ang, Jing Li, and Nachiket Kapre. 2016. Software-specific named entity recognition in software engineering social content. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 90–101. doi:10.1109/SANER.2016.10

[31] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter Liu. 2020. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization. In *International conference on machine learning (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 11328–11339. http://proceedings.mlr.press/v119/zhang20ae.html

Received 2024-09-10; accepted 2025-01-14