Expert Systems With Applications 299 (2026) 129972

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

iJTyper: An effective type inference framework for incomplete java codes by integrating constraint- and statistics-based methods

Zhixiang Chen [a], Anji Li [a], Neng Zhang [b,*], Jianguo Chen [a], Yuan Huang [a], Zibin Zheng [a]

[a] School of Software Engineering, Sun Yat-sen University, Zhuhai, 519082, China
[b] School of Computer Science, Central China Normal University, Wuhan, 430079, China

ARTICLE INFO

Keywords:
Java code snippets
Type inference
Code analysis

ABSTRACT

Inferring the types of APIs used in incomplete codes (also referred to as code snippets), e.g., those on Q&A forums, is a prerequisite step required to work with the codes. Existing type inference methods proposed for incomplete Java codes can be primarily categorized as constraint-based or statistics-based. The former relies on a pre-built API knowledge base (KB) and the type constraints in code snippets, which imposes higher requirements on code syntax and thus suffers from low recall due to the syntactic limitation. The latter overcomes the syntactic limitation by learning statistical regularities from a code corpus, however it rarely employs the type constraints in code snippets, which may lead to low precision. In this paper, we propose an effective type inference framework, called iJTyper, for incomplete Java codes by integrating the complementary advantages of constraint- and statistics-based methods. For a code snippet, iJTyper first applies a constraint-based method and augments the code context with the inferred API types. Then, it applies a statistics-based method to the augmented code snippet. The types predicted for APIs are further used to improve the constraint-based method by reducing its pre-built KB. iJTyper iteratively executes both methods and performs the code context augmentation and KB reduction mechanisms until a termination condition is satisfied. The final inference results are produced by combining the results of both methods. We implemented a version of iJTyper by integrating two state-of-the-art methods, SnR and MLMTyper, and evaluated iJTyper on two open-source datasets. Results show that 1) iJTyper achieves the highest average precision/recall[1] of 97.3 % and 92.5 % on both datasets; 2) iJTyper improves the average recall of SnR and MLMTyper by at least 7.3 % and 27.4 %, respectively; and 3) iJTyper improves the average precision/recall of the recently popular language model, ChatGPT, by 3.2 % and 0.5 % on both datasets.

1. Introduction

Incomplete codes (also referred to as *code snippets*) are widely used in online forums, e.g., Stack Overflow[2] (SO) and Quora,[3] to illustrate problems or provide examples of API usages (Sadowski et al., 2015). Software developers often search for code snippets to find solutions to coding problems (Bacchelli et al., 2012; Barua et al., 2014; Rosen & Shihab, 2016), which can save valuable time and effort from learning extensive knowledge they have not yet grasped. However, code snippets often do not provide sufficient information to determine the exact types (i.e., fully qualified names (FQNs)) of APIs used in them (Dagenais & Robillard, 2012; Kabir et al., 2025; Terragni et al., 2016). As a result, code snippets generally cannot be compiled and reused directly (Horton & Parnin, 2018; Yang et al., 2016). According to Terragni et al. (2016), more than 91 % code snippets at 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 %. Thus, inferring the types of APIs in code snippets is a prerequisite step required to work with the codes (Dong et al., 2022; Huang et al., 2022; Saifullah et al., 2019).

* Corresponding author.
E-mail addresses: chenzhx69@mail2.sysu.edu.cn (Z. Chen), lianj8@mail2.sysu.edu.cn (A. Li), nengzhang@ccnu.edu.cn (N. Zhang), chenjg33@mail.sysu.edu.cn (J. Chen), huangyuan5@mail.sysu.edu.cn (Y. Huang), zhzibin@mail.sysu.edu.cn (Z. Zheng).

[1] In this work, we focus on the type inference of two most important kinds of APIs, i.e., classes and interfaces, as explained in Section 2.1. For the methods that can infer the types of all classes and interfaces in a code snippet, e.g., iJTyper, MLMTyper, and ChatGPT, their respective precision and recall are the same.

[2] https://stackoverflow.com/

[3] https://www.quora.com/

https://doi.org/10.1016/j.eswa.2025.129972
Received 18 September 2024; Received in revised form 27 September 2025; Accepted 6 October 2025
Available online 9 October 2025

Existing type inference methods for incomplete Java codes can be primarily categorized as constraint-based (Dong et al., 2022; Shokri & Mirakhorli, 2021; Subramanian et al., 2014) or statistics-based (Huang et al., 2022; Phan et al., 2018; Saifullah et al., 2019; Velázquez-Rodríguez et al., 2023). Constraint-based methods typically pre-build a knowledge base (KB) from API libraries. Then, given a code snippet, they extract type constraints (e.g., the methods invoked by an object of a class) from the code snippet and employ heuristic rules to find the types of APIs that can maximally satisfy the constraints. Although these methods could achieve relatively high precision, they impose higher requirements on code syntax, which often results in low recall due to the syntactic limitation of code snippets. Statistics-based methods build statistical models from a code corpus and predict types with the greatest likelihood in the context of APIs. They are hardly affected by the syntactic limitation and could achieve relatively high recall. However, they generally do not take advantage of the type constraints in code snippets, which may lead to low precision. **The complementary nature of constraint- and statistics-based methods justifies their combination.**

In this paper, we propose a simple and effective type inference framework, called iJTyper, for incomplete Java codes by integrating the strengths of constraint- and statistics-based methods. Given a code snippet, iJTyper first applies a constraint-based method, M^c, to it. The inferred API types are used to augment the context of the code snippet. Then, the augmented code snippet is fed to a statistics-based method, M^s. The candidate types predicted for APIs by M^s are further used to reduce irrelevant types in the pre-built KB of M^c to improve M^c. iJTyper iteratively performs both methods and the *code context augmentation* and *KB reduction* mechanisms until a termination condition (e.g., the inference results of both methods become stable) is reached. Finally, iJTyper produces the inference results by combining the results of both methods.

To evaluate iJTyper, we implemented a version of it by integrating two state-of-the-art (SOTA) type inference methods, namely the constraint-based method, SnR (Dong et al., 2022), and the statistics-based method, MLMTyper (Huang et al., 2022). Then, we conducted a series of experiments on two open-source datasets collected from SO, i.e., StatType-SO (Phan et al., 2018) and Short-SO (Huang et al., 2022). Despite the two baselines, SnR and MLMTyper, we also compared iJTyper with the recently popular language model, ChatGPT (OpenAI, 2023). Moreover, we validated the code context augmentation and KB reduction mechanisms used to bridge the constraint- and statistics-based methods. The results are summarized as follows.

- iJTyper achieves the highest average precision/recall of 97.3 % on StatType-SO and 92.5 % on Short-SO, improving the average recall over SnR by 7.3 % and over MLMTyper by 27.4 %.
- iJTyper improves the average precision/recall of ChatGPT from 94.1 % to 97.3 % (+3.2 %) on StatType-SO and from 92.0 % to 92.5 % (+0.5 %) on Short-SO.
- The code context augmentation and KB reduction mechanisms respectively contribute to better performance of the constraint- and statistics-based methods.

The main contributions of this paper are outlined below:

- We propose an effective and easy-to-use type inference framework, iJTyper, for integrating constraint- and statistics-based type inference methods. iJTyper is flexible and can be extended to integrate different constraint- and statistics-based methods by adapting the input and output formats and encapsulating the methods as functions.
- We propose two mechanisms, i.e., code context augmentation and KB reduction, to transfer useful inference results between constraint- and statistics-based methods, which help improve both methods. The key ideas of these mechanisms can be employed by researchers to design better constraint- or statistics-based methods.
- We conducted extensive experiments to evaluate iJTyper in comparison with two SOTA baselines and ChatGPT. Our replication package including the source code and experimental data is released at GitHub.[4]

The rest of the paper is organized as follows. Section 2 introduces the motivation of this work based on an in-depth analysis of constraint- and statistics-based type inference methods. Section 3 describes the details of iJTyper. Section 4 presents experimental results. Section 5 discusses the top-k evaluation on larger code snippets, global correctness and intermediate process statistics, impact of library and version differences, practicality, extensibility, maintainability, ethical considerations, and limitations of iJTyper, as well as the threats to the validity of this work. Section 6 reviews related work. Section 7 concludes the paper and discusses future work.

2. Motivation

2.1. Type inference task

The type inference task in this work is to determine the exact types (i.e., fully qualified names (FQNs)) of Java program elements within a code snippet. While a code snippet may contain five kinds of program elements (i.e., *classes*, *interfaces*, *methods*, *fields*, and *variables*), our inference targets are only classes and interfaces used in unqualified form. Once the FQNs of them are known, the signatures of methods and fields (and hence the types of variables) can be determined.

Fig. 1 shows a code snippet from the SO post '520902', annotated with five kinds of program elements. In this example, after resolving the simple name *AnnotationConfiguration* to *org.hibernate.cfg.AnnotationConfiguration*, we can look up the class declaration[5] to infer that the invoked method *configure()* must resolve to *public AnnotationConfiguration configure() throws HibernateException*. The type of the field *sessionFactory*, or any new variable declared in this function, can then be further inferred based on the return type of the method *buildSessionFactory()*. Since inferring the FQNs of classes and interfaces unlocks the types of other elements, existing methods in Java primarily focus on identifying those two categories. Moreover, classes and interfaces in the *java.lang* package (e.g., *Integer* and *String*) are often excluded as they are implicitly imported by default in Java and can be directly resolved. **Similarly, we focus on inferring the FQNs of classes and interfaces that require explicit import by developers when describing our iJTyper framework.**

2.2. Limitations of type inference methods

Existing type inference methods for Java code snippets can be categorized into two groups: constraint-based (Dong et al., 2022; Shokri, 2021; Subramanian et al., 2014) and statistics-based (Huang et al., 2022; Li et al., 2025; Phan et al., 2018; Saifullah et al., 2019; Velázquez-Rodríguez et al., 2023).

2.2.1. Constraint-based type inference methods

A constraint-based method typically pre-builds a *knowledge base* (KB) that stores the type information of APIs extracted from a set of API libraries. This knowledge includes the types of declared methods and fields, as well as the inheritance and implementation relationships, which are required for type inference. Unlike a symbol table, which is typically used during compilation to track scoped identifiers, the KB in our context focuses on API type information and does not maintain scope information. Thereafter, given a code snippet, the constraint-based method extracts type constraints between APIs hidden in the code

[4] https://github.com/zhxchen-se/iJTyper

[5] https://docs.jboss.org/hibernate/orm/4.2/javadocs/org/hibernate/cfg/AnnotationConfiguration.html

```
1. public class HibernateUserDAO implements UserDAO {
2.
3.     private SessionFactory sessionFactory;
4.
5.     public HibernateUserDAO() {
6.         AnnotationConfiguration annotConf = new AnnotationConfiguration();
7.         annotConf.addAnnotatedClass(User.class);
8.         annotConf.configure();
9.         sessionFactory = annotConf.buildSessionFactory();
10.    }
11. }
```

class
interface
field
variable
method

Fig. 1. Example code snippet annotated with four kinds of program elements.

Table 1
Constraint-based type inference methods.

Method	Publication	Summary
Baker	ICSE 2014	Baker maintains a candidate list for each API and applies heuristic rules to reduce the number of candidates deductively.
DepRes	ASE 2021	DepRes generates a sketch for each API and utilizes a Z3 SMT Solver to handle the constraints, aiming to find the minimum number of types and dependencies.
SnR	ICSE 2022	SnR repairs the input code snippet to a syntactically valid compilation unit and utilizes Datalog to solve the constraints extracted from the AST of the unit. It ranks the candidates based on a prioritization heuristic and selects the top item.

```
1. public class SBooksSearch extends Activity {
2.     private EditText mTextSearch;
3.     @Override
4.     protected void onCreate(Bundle savedInstanceState) {
5.         // TODO Auto-generated method stub
6.         super.onCreate(savedInstanceState);
7.         setContentView(R.layout.sbooks_search);
8.         mTextSearch =   (EditText)findViewById(R.id.edit_search);
9.         Button searchButton = (Button)findViewById(R.id.btn_search);
10.        Intent data = new Intent();
11.        setResult(RESULT_OK, data);
12.    }
13. }
```

Fig. 2. Example code snippet with syntax errors. For the convenience of demonstration, the original code is simplified. A syntax error is that the variable RESULT_OK is used before definition.

snippet, e.g., the methods invoked by an object variable of a class. Finally, the types of APIs that can satisfy the constraints at most are determined based on the KB.

Table 1 presents a list of constraint-based methods. According to the evaluations reported in Dong et al. (2022), Saifullah et al. (2019), SnR achieves the best performance. SnR pre-builds a KB from six popular Java/Android libraries, i.e., Android, GWT, Hibernate, JDK, Joda Time, and XStream, by extracting the classes and interfaces, the methods and fields declared in the classes/interfaces, and the inheritance and implementation relationships between the classes/interfaces. For a code snippet, SnR first attempts to repair it to a syntactically valid compilation unit using a template-based method, such that an Abstract Syntax Tree (AST) can be generated for the unit. If the AST is successfully generated, SnR then extracts type constraints from the AST and represents the constraints using a declarative logic programming language, Datalog (Ullman, 1984). Based on the KB and constraints, SnR infers the types of APIs in the code snippet using several heuristic rules, e.g., minimizing the number of unique libraries used in the code snippet.

A major limitation of constraint-based methods is that they require syntactically correct input code snippets to extract the constraints for type inference. Although some methods, e.g., SnR, address this problem

Table 2
Type inference results produced by SnR and MLMTyper for the code snippet shown in Fig. 2. To distinguish multiple APIs with the same name in the code snippet, two indices (i.e., the line number and the order of occurrence) are used to locate a specific API, e.g., 'Intent[10, 2]' indicates the 2nd class 'Intent' at line 10.

API	SnR	MLMTyper	
Activity[1,1]	–	*android.app.Activity*	✔
EditText[2,1]	–	*android.widget.EditText*	✔
Bundle[4,1]	–	*java.util.Bundle*	✗
EditText[8,1]	–	*android.widget.EditText*	✔
Button[9,1]	–	*android.widget.Button*	✔
Button[9,2]	–	*android.widget.Button*	✔
Intent[10,1]	–	*android.content.Intent*	✔
Intent[10,2]	–	*android.content.Intent*	✔

using a template-based syntax repair method, there still exists a large portion of code snippets that cannot be repaired or can only be partially repaired. Fig. 2 shows a code snippet from the SO post '1254832'. Table 2 presents the type inference results produced by SnR. As can be seen, SnR fails to infer the types for all classes because of an encountered exception, "*java.lang.UnsupportedOperationException: RESULT_OK could not be a valid type name or a variable name*". This exception is caused by the syntax error of using the variable RESULT_OK before definition. Through analysis, RESULT_OK is predefined in the class *android.app.Activity* and should be used after importing the class.

Apart from the compilation limitation, constraint-based methods are additionally limited by 1) the coverage of the pre-built KB; 2) the degree to which the types in the code snippet are constrained; and 3) the capability of the method implementations in extracting the constraints. Due to the limitation arising from the coverage of the pre-built KB, the types of APIs outside the KB cannot be inferred. A similar issue is shared by statistics-based methods, as they are also constrained by the coverage of their training corpus. However, this problem is somewhat mitigated for LLMs due to their training on large-scale corpus. For a code snippet, if an API has multiple candidate types in the KB, in order to determine the right type, the code snippet should contain sufficient constraints for filtering the other candidates; and meanwhile the methods should be able to effectively extract the constraints and use them for type inference. Fig. 3 shows a code snippet from the SO post '3954392', which illustrates the limitation of SnR on the capability of extracting constraints from code snippets. By applying SnR to the code snippet, four classes get their right types, but no type is inferred for the class *Document* [7,1]. We analyzed the intermediate results of SnR and found that the class has nine candidate types in the pre-built KB. Although the KB contains the right type, *com.google.gwt.dom.client.Document*, SnR fails to determine the type. This issue is because SnR cannot extract the full constraints expressed by the cascaded method calls,

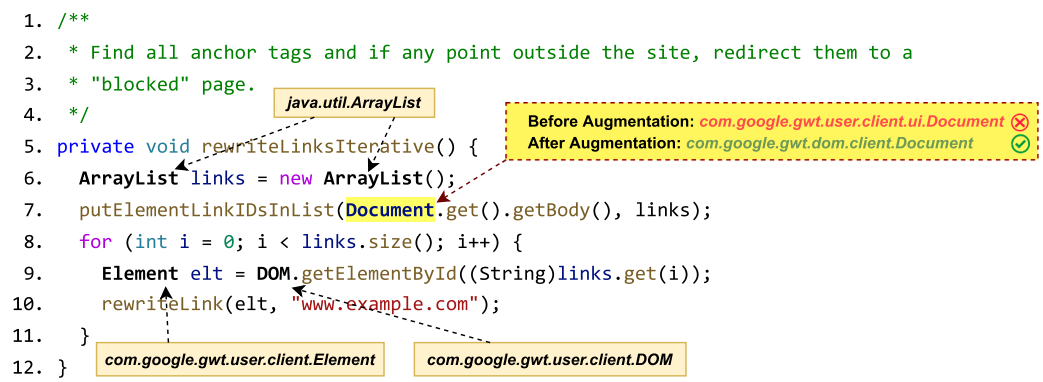

Fig. 3. Example type inference results of the statistics-based method, MLMTyper, improved by augmenting a code snippet with the types of APIs inferred using the constraint-based method, SnR. The types with a brown background are inferred by applying SnR to the original code snippet. The types in the yellow box are predicted by MLMTyper before and after augmenting the original code snippet with the inferred types of SnR.

Table 3
Statistics-based type inference methods.

Method	Publication	Summary
StatType	ICSE 2018	StatType treats the type inference task as a statistical machine translation task and considers both type context and resolution context.
COSTER	ASE 2019	COSTER utilizes both local and global contexts of an API. It calculates a likelihood score, a context similarity score, and a name similarity score based on a pre-built occurrence likelihood dictionary to map an API to its type.
RESICO	SCP 2023	RESICO uses the Word2vec model to vectorize the API reference and its context, which is then fed into a machine learning classifier to obtain type recommendations.
MLMTyper	ASE 2022	MLMTyper transforms the type inference task into a cloze-filling problem and uses a prompt-tuned masked language model (MLM) to complete the missing types.
CKTyper	FSE 2025	CKTyper retrieves crowdsourced knowledge (i.e., texts from SO posts) and inputs them together with code snippets into ChatGPT for type inference.

Document.get().getBody(), and thus cannot distinguish the right type from a noise type, *com.extjs.gxt.ui.client.widget.Document*, which also has a method called *get()* but the return type does not have a method called *getBody()*.

2.2.2. Statistics-based type inference methods

A statistics-based method builds a statistical model from a code corpus using machine learning techniques. The types of APIs in the corpus are given. Using the model, the method infers the types of APIs in a code snippet by maximizing the likelihood of the code snippet. Statistics-based methods eliminate the syntactic limitation of code snippets by treating them as plain text.

Statistics-based methods are workable because of the software naturalness (Hindle et al., 2016). That is, human-written code is mostly simple, repetitive, and predictable, and can be modeled by statistical models to facilitate various tasks, such as type inference (Huang et al., 2022; Phan et al., 2018; Saifullah et al., 2019; Velázquez-Rodríguez et al., 2023), defect localization (Qiu et al., 2021; Zhang et al., 2020), and code completion (Raychev et al., 2014; Svyatkovskiy et al., 2019). Code context is used to capture such naturalness (Saifullah et al., 2019). The key difference between statistics-based type inference methods is the code context that they use to predict the types of APIs. Table 3 presents a list of statistics-based methods. For example, StatType (Phan et al., 2018) leverages type context and resolution context. The former refers to the classes, methods, and fields that occur around an API, e, whose type needs to be inferred, while the latter refers to the type inference decision of the APIs surrounding e. CKTyper (Li et al., 2025) leverages crowdsourced knowledge (i.e., discussion texts from SO posts) to support type inference with ChatGPT. The prompt-tuned masked language model (MLM)-based method proposed by Huang et al. (2022), referred to as MLMTyper in this paper, uses the code line where e is located and the top and down η (=2 by default) adjacent code lines to form a code context block.

According to the evaluations reported in Huang et al. (2022), Velázquez-Rodríguez et al. (2023), MLMTyper achieves the best performance. It transforms the type inference task into a cloze-filling problem and uses a prompt-tuned MLM to complete the missing types. MLMTyper has two execution modes called *leave-one-out* and *all-unknown*. Specifically, MLMTyper predicts each API in a code snippet one by one. For a specific API, e, after building the code context block, the leave-one-out mode predicts the top-k candidate types of e with the known types of all the other APIs in the context. In contrast, the all-unknown mode predicts the types of e without utilizing the types of any other APIs, i.e., assuming that those types are all unknown. The leave-one-out and all-unknown modes respectively represent the upper bound (i.e., the easiest case) and lower bound (i.e., the most difficult case) of type inference scenarios.

A major limitation of statistics-based methods is that they rarely take the type information of APIs and the type constraints in code snippets into consideration, which may lead to low precision. Moreover, the statistics-based methods that use a generative language model suffer from a well-known problem called *hallucination* (Ji et al., 2023), i.e., some predicted types are haphazardly generated and do not actually exist. Fig. 4 shows a code snippet from the SO post '8746084'. Table 4 presents the top-1 types of the classes inferred by MLMTyper. MLMTyper correctly predicts *DateTime* [2,1] as *org.joda.time.DateTime* but mistakenly predicts the types of the other three classes. For example, the return type of the method *org.joda.time.DateTime.toLocalDate()* is *org.joda.time.LocalDate*. However, MLMTyper does not consider this type information and predicts the wrong type for *LocalDate* [3,1]. For the classes *DateTimeFormatter* [1,1] and *DateTimeFormat*[1,1], MLMTyper haphazardly generates two hallucinated types *java.text.DateTimeFormatter* and *java.text.DateTimeFormat* for them. Specifically, when MLMTyper is required to infer types from packages that were not encountered during the prompt learning stage (i.e., zero-shot at the package level), it is prone to errors, as also observed in the packages *com.cloudbees.api.config* and *com.extjs.gxt.ui.client.widge* (Huang et al., 2022). iJTyper mitigates the hallucination issue by filtering out candidates that are not present in the pre-built knowledge base and by assigning higher priority to the results from the constraint-based method (see Section 3).

2.2.3. Insight.

As explained above, both constraint- and statistics-based type inference methods have their limitations. Constraint-based methods often have relatively high precision by leveraging the type information of APIs in KB and the type constraints in code snippets, however their recall may be low due to several factors, e.g., the compilation failure of code

```
1. DateTimeFormatter FORMATTER = DateTimeFormat.forPattern("yyyy-MMM-dd");
2. DateTime dateTime = FORMATTER.parseDateTime("2005-nov-12");
3. LocalDate localDate = dateTime.toLocalDate();
```

Fig. 4. Example code snippet used to demonstrate the complementarity of the constraint-based method, SnR, and the statistics-based method, MLMTyper.

Table 4
Type inference results produced by SnR and MLMTyper for the code snippet shown in Fig. 4. Similar to Table 2, two indices (i.e., the line number and the order of occurrence) are used to locate a specific API in the code snippet.

API	SnR		MLMTyper	
DateTimeFormatter[1,1]	*org.joda.time.format.DateTimeFormatter*	✔	*java.text.DateTimeFormatter*	✗
DateTimeFormat[1,1]	*org.joda.time.format.DateTimeFormat*	✔	*java.text.DateTimeFormat*	✗
DateTime[2,1]	*org.joda.time.DateTime*	✔	*org.joda.time.DateTime*	✔
LocalDate[3,1]	*org.joda.time.LocalDate*	✔	*java.time.LocalDate*	✗

snippets and the low coverage of the pre-built KB. In contrast, statistics-based methods do not require to compile code snippets and can infer types for all APIs, which could improve the recall. However, their precision may be low since they rarely consider the type information of APIs and the constraints in code snippets and thus may lead to wrong types. Tables 2 and 4 present the type inference results produced by SnR and MLMTyper for two code snippets. SnR achieves higher precision in the second case but a lower recall in the first case. In contrast, MLMTyper achieves a higher precision/recall in the first case but a lower precision/recall in the second case.

We further observe from Tables 2 and 4 that the type inference results produced by SnR and MLMTyper exhibit a certain degree of complementarity. For example, SnR fails to infer types for the code snippet shown in Fig. 2, while MLMTyper can get most of the types right. On the contrary, SnR infers all the four types correctly for the code snippet shown in Fig. 4, but MLMTyper makes three types wrong. Based on this observation, better performance could be achieved by integrating constraint- and statistics-based methods. Specifically, the limitations of constraint-based methods can be partially addressed by statistics-based methods because of their two main advantages: 1) They are not affected by the syntax errors in code snippets; and 2) They employ the context (e.g., token co-occurrence) of APIs to predict types without the need to extract constraints from code snippets. In turn, the types inferred by the constraint-based method could be used to augment the context of APIs in code snippets and improve the statistics-based method. Following this insight, we propose an integrated type inference framework, which is elaborated in the following section.

3. Methodology

Fig. 5 shows our iJTyper framework. The input is a code snippet, and the output is the code snippet with inferred types of the APIs in it. iJTyper contains four steps. At first, iJTyper applies a constraint-based type inference method, M^c, to the input code snippet. Since the inference results may include incomplete or incorrect types due to the limitations of M^c (see Section 2.2.1), iJTyper further feeds the code snippet augmented with the inferred types to a statistics-based type inference method, M^s. The candidate types predicted by M^s could be used to improve M^c by filtering irrelevant types in the pre-built KB of M^c. Thus, iJTyper adopts an iterative mechanism. After performing a round of both methods, iJTyper examines whether a termination condition (e.g., the inference results become stable) is satisfied. If not, iJTyper reduces the pre-built KB of M^c using the top-k candidate types of each API predicted by M^s and re-applies M^c to the code snippet. After the iteration is terminated, iJTyper combines the inference results of both methods to produce the final inference results.

Notice that the goal of iJTyper is to achieve SOTA performance in type inference by reusing existing constraint- and statistics-based methods and enhancing each other by exploiting their inference results. Although in this work, we demonstrate iJTyper by implementing it using two SOTA methods, SnR and MLMTyper, the framework is flexible and can be easily used to integrate different constraint- and statistics-based methods. Additionally, based on our literature review, some constraint- or statistics-based methods are complex; and their replication packages are also difficult to understand and modify. For example, the implementation code of SnR is obfuscated and compiled to a Jar file, making it difficult to replicate and extend the work. iJTyper uses a simple strategy to integrate any constraint- and statistics-based methods as long as they have a workable replication package. Instead of modifying their implementation, iJTyper propose two key mechanisms, i.e., the code context augmentation and KB reduction described in Sections 3.1 and 3.3, respectively, to combine the advantages of both methods by transferring their inference results. **We acknowledge that this strategy sacrifice some efficiency, however it can help users escape from the time and effort required to understand and modify the implementation of constraint- and statistics-based methods; and thus iJTyper could be desired by users in non real-time application scenarios.**

3.1. Constraint-based type inference

Code Context Augmentation. Recall that statistics-based methods treat code snippets as plain text. The input code snippet is viewed as a token sequence, i.e., $CS = t_1 t_2 \ldots t_m$ where some of the tokens are APIs. The set of APIs, denoted as $APIs(CS)$, can be obtained using regular expressions based on a dictionary of APIs built from API libraries. Based on the type inference results of M^c, $APIs(CS)$ is divided into two subsets: the set of APIs with inferred types and the set of APIs without inferred types, which are denoted as $Typed\_APIs^c(CS)$ and $NonTyped\_APIs^c(CS)$, respectively. The superscript 'c' stands for the constraint-based method. The inferred type of an API, $t_i \in Typed\_APIs^c(CS)$, is denoted as $type^c(t_i)$. Then, the augmented code snippet is represented as a new token sequence, i.e., $ACS = y_1 y_2 \ldots y_m$ where the APIs are replaced with their types, i.e.,

$$y_i = type^c(t_i) \text{ if } t_i \in Typed\_APIs^c(CS) \text{ else } t_i, \forall i = 1, \ldots, m \quad (1)$$

3.2. Statistics-based type inference

In this step, iJTyper applies a statistics-based type inference method, M^s, to the augmented code snippet ACS. As a result, a list of candidate types are predicted for each API. However, due to the limitations of statistics-based methods described in Section 2.2.2, the precision may not be high, as demonstrated in Fig. 4. Since the recall of statistics-based methods is relatively high, the average recall of the top-3 candidate types predicted by MLMTyper is 91 % (Huang et al., 2022) (which means that the vast majority of the right types of APIs are included in the top-3), the top-k (e.g., $k = 3$) candidate types predicted by M^s can be used to reduce irrelevant types in the pre-built KB of M^c and thus improve the performance of M^c.

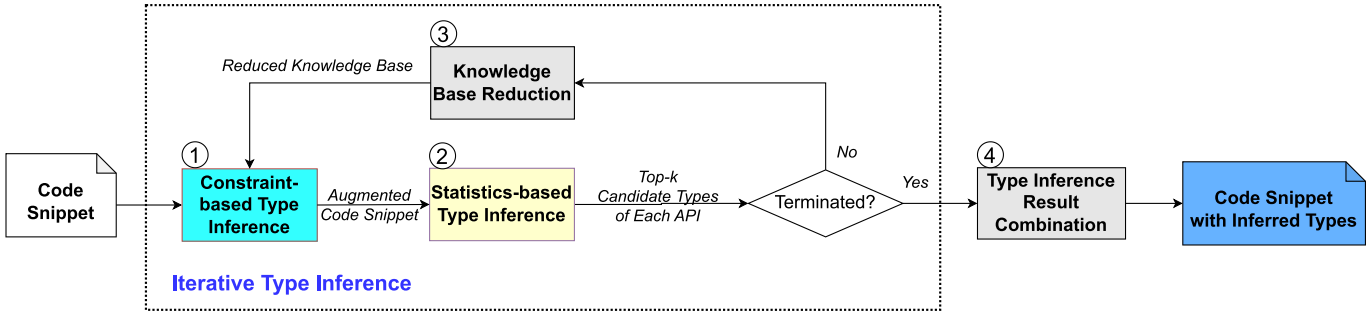

Fig. 5. The framework of iJTyper.

Notice that if M^s employs a generative language model, there can be candidate types haphazardly generated due to the hallucination problem, as demonstrated in Fig. 3. Such hallucinated candidates may affect the efficacy of the KB reduction strategy described in Section 3.3. Specifically, if the right type of an API is outside the top-k candidates, reducing the pre-built KB of M^c based on the top-k candidates will miss the right type. To solve this problem, we filter the candidates that do not exist in the pre-built KB of M^c. The refined top-k candidate types of each API $e \in APIs(CS)$, denoted as $types_k^s(e)$ (where the superscript 's' stands for the statistics-based method), are subsequently used to reduce the pre-built KB of M^c.

3.3. Knowledge base reduction

Constraint-based type inference methods search for the types of APIs in a code snippet from the pre-built KB. Generally, the KB should cover a large number of API libraries to ensure a good capability of the methods. As a result, there can be multiple candidate types of an API from different libraries in the KB. If a code snippet does not contain sufficient constraints to determine the right type of an API from the candidates, the methods will not be able to produce the desired answer. This limitation can be alleviated by leveraging the type inference results of statistics-based methods. Specifically, we introduce a KB reduction module in iJTyper to reduce the pre-built KB of M^c by removing types irrelevant to the APIs in CS according to the (refined) top-k candidate types predicted for each API using M^s. In addition, considering that the comprehensive type information of APIs is necessary for constraint-based methods to solve the constraints from code snippets, we implement the KB reduction module in two phases.

Phase 1: Candidate Class & Interface Type Collection. As explained in Section 2.1, we focus on the type inference of classes and interfaces in code snippets. In this phase, we build a set of candidate types, $CanCITypes$, that can probably contain the right types of all classes and interfaces in the input code snippet CS. At first, we collect the top-k candidate types predicted by M^s for each class/interface, resulting in an initial $CanCITypes$. Since the performance of existing statistics-based methods is not perfect (e.g., both the precision and recall of MLMTyper are not 100 %), it is possible that the top-k candidate types of a class or an interface do not contain the right type. Based on this consideration, if a class or an interface has a type inferred by M^c, we add the type to $CanCITypes$ as it might be the right type. Finally, $CanCITypes$ can be formally defined as

$$CanCITypes = \left(\bigcup_{e \in CIs(CS)} types_k^s(e) \right) \cup \{type^c(e) | e \in Typed\_APIs^c(CS) \cap CIs(CS)\} \tag{2}$$

where $CIs(CS)$ represents the set of classes and interfaces in CS. Note that $CIs(CS)$ is a subset of $APIs(CS)$, i.e., $CIs(CS) \subseteq APIs(CS)$.

Phase 2: Reduced KB Construction. In this phase, we construct a reduced KB by extracting the declaration information of each class/interface type $ci\_type \in CanCITypes$ from the pre-built KB, denoted as pre_KB, of M^c. The declaration information includes the methods and fields declared in ci_type, the super classes/interfaces of ci_type and their declared methods and fields. The reduced KB is formally defined as

$$Reduced\_KB = \bigcup_{ci\_type \in CanCITypes} Declare\_Info(ci\_type, pre\_KB) \tag{3}$$

where $Declare\_Info(ci\_type, pre\_KB)$ represents the declaration information of ci_type in pre_KB, i.e.,

$$Declare\_Info(ci\_type, pre\_KB) = \bigcup_{e \in \{ci\_type\} \cup SupCITypes(ci\_type, pre\_KB)} e \cup e.Methods \cup e.Fields \tag{4}$$

where $SupCITypes(ci\_type, pre\_KB)$ represents the set of super class/interface types of ci_type in pre_KB. For a class or an interface type, e, $e.Methods$ and $e.Fields$ are the set of methods and the set of fields declared in e, respectively. The reduced KB also preserves the relationships between the APIs in it.

3.4. Iterative type inference

Based on the reduced KB, iJTyper re-applies the constraint-based method M^c to the input code snippet CS to obtain new inferred types of APIs. Since many noise types may be excluded in the reduced KB, the inferred types could be improved. As an example, Fig. 6 shows a code snippet from the SO post '39005622'. Before reducing the pre-built KB, the types inferred by SnR for the interface *Converter*[1,1] and the class *UnmarshallingContext*[16,1], i.e., *com.sun.xml.internal.ws.commons.xmlutil.Converter* and *com.sun.xml.internal.bind.v2.runtime.unmarshaller.UnmarshallingContext*, respectively, are both wrong. After reducing the KB, SnR infers the right types for them because the two wrong types are excluded in the reduced KB.

Furthermore, the new types with higher quality inferred by M^c can be used to better augment CS and improve the statistics-based method M^s again. Considering this **mutual promotion effect** between both methods, iJTyper adopts an iterative mechanism, as shown in Fig. 5. To control the iteration, we set two termination conditions: 1) The inference results of M^c and M^s become stable (i.e., remain unchanged); or 2) The number of iterations exceeds a threshold δ ($=10$ by default). The second condition is used to avoid infinite iteration due to repeated oscillations in the results, which is observed in an variant of iJTyper (see Table 6 in Section 4.2). After performing a round of both methods, iJTyper examines the termination conditions. The iteration ends when either of the two conditions is satisfied.

3.5. Type inference result combination

After the iterative type inference process is terminated, iJTyper combines the inference results of M^c and M^s to produce the final inference results using two phases.

Phase 1: Combining the Inference Results of Each Method. Through our observation from the experiments, there can be fluctuations

```java
1. public class IntegerConverter implements Converter {
2.
3.     @SuppressWarnings("rawtypes")
4.     @Override
5.     public boolean canConvert(Class clazz) {
6.         return clazz.equals(Integer.class);
7.     }
8.
9.     @Override
10.    public void marshal(Object object, HierarchicalStreamWriter writer,
11.            MarshallingContext context) {
12.    }
13.
14.    @Override
15.    public Object unmarshal(HierarchicalStreamReader reader,
16.            UnmarshallingContext context) {
17.        String text = (String)reader.getValue();
18.        Integer number = Integer.parseInt(text.trim());
19.        return number;
20.    }
21. }
```

Before KB Reduction: com.sun.xml.internal.ws.commons.xmlutil.Converter
After KB Reduction: com.thoughtworks.xstream.converters.Converter

Before KB Reduction: com.sun.xml.internal.bind.v2.runtime.unmarshaller.UnmarshallingContext
After KB Reduction: com.thoughtworks.xstream.converters.UnmarshallingContext

Fig. 6. Example of the type inference results of the constraint-based method, SnR, improved by leveraging the knowledge base (KB) reduced based on the (refined) top-3 candidate types predicted using the statistics-based method, MLMTyper. The types in the yellow boxes are inferred by SnR before and after reducing the pre-built KB.

in the results produced by M^c and M^s during the iteration process. One case is that the inferred type(s) of an API in one round may differ from those in another round. Another case is that the inference of an API may fail in one round but succeed in another round. Unfortunately, there is no way to design the optimal strategy to find the maximum number of right types inferred for APIs produced in different rounds, without knowing the ground truth. Here, we combine the inference results of each method using a simple strategy, i.e., keeping the last successfully inferred type(s) of each API. After this phase, we denote the type of an API, e, inferred by M^c as $comb\_type^c(e)$ and denote the (refined) top-k candidate types of e predicted by M^s as $comb\_types^s_k(e)$.

Phase 2: Combining the Inference Results of Both Methods. In this phase, we combine the type inference results of M^c and M^s obtained in the first phase. Since the precision of constraint-based methods is generally higher than that of statistics-based methods, we combine the type(s) of an API, e, inferred by both methods according to two criteria: 1) If only M^s predicts the top-k candidate types of e, i.e., $comb\_type^c(e) == null \wedge comb\_types^s_k(e) \neq \emptyset$, then the top-1 type, $comb\_types^s_1(e)$, is determined as the type of e; and 2) If both methods produce types for e, i.e., $comb\_type^c(e) \neq null \wedge comb\_types^s_k(e) \neq \emptyset$, then the type inferred by M^c, $comb\_type^c(e)$, is accepted as the type of e.

4. Evaluation

In this section, we evaluate iJTyper by answering the following research questions (RQs):

RQ1: How do the internal settings of iJTyper affect its performance?
RQ2: Can iJTyper improve SOTA type inference methods?
RQ3: What are the contributions of the code context augmentation and KB reduction mechanisms used in iJTyper?

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

We used two open-source datasets of Java code snippets collected from SO: StatType-SO (Phan et al., 2018) and Short-SO (Huang et al., 2022). StatType-SO contains 268 code snippets involving 4086 APIs. The APIs included in each code snippet are primarily from one of six popular libraries, namely Android, GWT, Hibernate, JDK, Joda Time, and XStream. StatType-SO has been widely used in prior studies (Dong et al., 2022; Huang et al., 2022; Phan et al., 2018; Saifullah et al., 2019; Velázquez-Rodríguez et al., 2023). Short-SO contains 120 code snippets involving 525 APIs. The APIs in each code snippet are also related to one of the same six libraries as StatType-SO. The ground truth for both StatType-SO and Short-SO has been publicly released along with the dataset and has been checked by us. The main difference between the two datasets is that all code snippets in Short-SO contain less than three lines of code, whereas the code snippets in StatType-SO are longer with an average of 28 lines of code.

4.1.2. Implementation of iJTyper and baselines

We implemented a prototype of iJTyper by integrating two SOTA type inference methods: the constraint-based method, SnR (Dong et al., 2022), and the statistics-based method, MLMTyper (Huang et al., 2022). The details of SnR and MLMTyper are described in Section 2. We implemented them based on the replication package[6] of SnR released at Zenodo and the replication package[7] of MLMTyper released at GitHub. In addition, since ChatGPT (OpenAI, 2023) has been widely used to address various software engineering tasks, we also chose the ChatGPT implemented based on the GPT-3.5 (text-davinci-002-render-sha) model as a baseline. An online version of it is available on the official webpage[8] provided by OpenAI. To guide ChatGPT in inferring the types of APIs in a code snippet, we designed a prompt template as illustrated in Fig. 7.

[6] https://doi.org/10.5281/zenodo.5843327
[7] https://github.com/SE-qinghuang/ASE-22-TypeInference
[8] https://chat.openai.com

```
Given a Java code snippet, please infer the fully qualified names
(FQNs) of the specified API elements in it. The answer should
be outputed in JSON format, such as {'API element': '<FQN>', ...}.

Java Code:

1. public class gwt_class_49 {
2.   private void rewriteLinksIterative() {
3.     ArrayList links = new ArrayList();
4.     putElementLinkIDsInList(Document.get().getBody(), links);
5.     for (int i = 0; i < links.size(); i++) {
6.       Element elt = DOM.getElementById((String) links.get(i));
7.       rewriteLink(elt, "www.example.com");
8.     }
9.   }

API elements:
['ArrayList', 'Document', 'Element', 'DOM']
```

Fig. 7. Prompt for guiding ChatGPT to infer the FQNs of API elements in a code snippet.

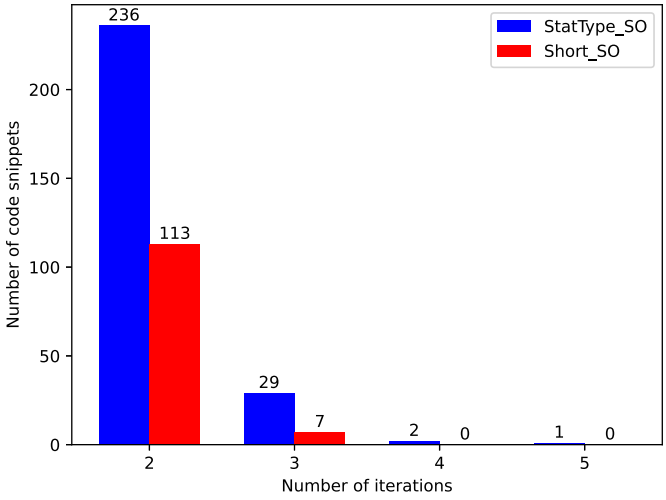

Fig. 8. Distribution of the number of iterations of iJTyper performed on two datasets.

4.1.3. Metrics

To measure the performance of iJTyper and the baselines, we adopted two widely used metrics: precision and recall. For a code snippet, precision measures the percentage of the types correctly inferred by a method; and recall measures the percentage of the APIs whose types are correctly inferred by a method.

$$Precision = \frac{\#\text{Correctly Inferred Types}}{\#\text{Inferred Types}} \tag{5}$$

$$Recall = \frac{\#\text{Correctly Inferred Types}}{\#\text{Requested Types}} \tag{6}$$

where "#Requested Types" is the number of APIs whose types are requested to be inferred. **The precision and recall of MLMTyper, ChatGPT, and iJTyper are equal on a given code snippet, since they are or incorporate statistics-based methods that are capable of inferring types for all requested APIs, thus "#Inferred Types" == "#Requested Types".**

Note that for a fair comparison, we only consider the top-1 type predicted for an API by MLMTyper. For each of the two datasets, after measuring the precision and recall of a method on every code snippet, we calculated the average precision and recall of the method on the set of code snippets related to each of the six libraries as well as the average precision and recall of the method on all code snippets. According to the explanation above, the average precision and recall of the methods that can infer the types of all required APIs are also the same. Moreover, we conducted a Wilcoxon signed-rank test (Wilcoxon, 1992) on the results of iJTyper and the baselines, using significance levels of $\alpha = 0.05$, 0.01, and 0.001.

4.2. RQ1: How do the internal settings of iJTyper affect its performance?

Motivation. In iJTyper, there are two internal settings. One is the parameter k which means the number of candidate types predicted by the statistics-based method M^s that are used to reduce the pre-built KB of the constraint-based method M^c. The setting of k may affect the quality of the reduced KB and the performance of M^c and iJTyper. Intuitively, a large k (e.g., 10) may retain too many noise types of APIs, while a small k (e.g., 1) may exclude the right types of some APIs. The other one is the execution order of M^c and M^s. As shown in Fig. 5, iJTyper applies M^c before M^s. However, it is able to exchange the execution order of both methods. We need to validate the impacts of the parameter k and the execution order on the performance of iJTyper. Moreover, since iJTyper is an iterative framework, we want to investigate the number of iterations required by iJTyper to produce the final inference results for code snippets and the performance of iJTyper during the iterations.

Approach. We applied iJTyper to each code snippet in StatType-SO and Short-SO with different settings of $k \in \{1, 3, 5, 10\}$. For a specific setting of k, we measured the average precision and recall of iJTyper (which are the same as explained in Section 4.1.3) on all code snippets in each dataset. We also measured the average precision/recall of iJTyper based on the intermediate results produced after each iteration. After determining the proper setting of k, we implemented a variant of iJTyper, referred to as iJTyper-SC, by executing M^s before M^c. We applied iJTyper-SC to all code snippets and also measured its average precision/recall on each dataset.

Results. iJTyper converges within up to three iterations on 99.2 % code snippets. Fig. 8 shows the distribution of the number of iterations performed by iJTyper on all code snippets in the two datasets. As can be seen, iJTyper converges on 349 (89.9 %), 36 (9.3 %), 2 (0.5 %), and 1 (0.3 %) code snippets after two, three, four, and five iterations, respectively. We manually inspected three code snippets that required four to five iterations and found that some variables in these snippets had limited available constraint information, which led to oscillations in the results produced by the constraint-based method.

iJTyper achieves the best performance after the second iteration. Table 5 presents the average precision/recall of iJTyper on the two datasets under different settings of k within five iterations. The average precision/recall achieved by iJTyper after the first iteration is relatively high, i.e., over 95 % on StatType-SO and over 90 % on Short-SO, respectively. After the second iteration, iJTyper reaches the highest average precision/recall of 97.5 % and 92.5 % on both datasets. This result confirms the necessity of the iterative type inference mechanism used in iJTyper. However, with more iterations, the average precision/recall of iJTyper decreases slightly and finally becomes stable on StatType-SO. We examined the experimental results of each iteration and found that iJTyper's decrease in average precision/recall during the third and fourth iterations on StatType-SO is caused by the inference of two classes, i.e., *javax.persistence.Entity* and *javax.persistence.Table*, in a code snippet. iJTyper correctly inferred them in the second iteration. After that, the former was inferred as *org.hibernate.annotations.Entity* in the third iteration; and the latter was inferred as *org.hibernate.annotations.Table* in the fourth iteration. The inference results of the fifth iteration are the same as those of the fourth iteration, thus the iterative process converged. One possible explanation for the observed flip in results is that, throughout all iterations, the statistics-based component consistently performed poorly (correctly predicting only 2 out of 7 FQNs). The constraint-based component failed to correct these errors during the third iteration and was instead negatively impacted in subsequent iterations, ultimately leading to a decline in overall performance. In fact, both *org.hibernate.annotations.Entity/Table* and *javax.persistence.Entity/Table* can be used in the context of the code snippet, but the latter is more portable and recommended.

Table 5
Average precision and recall (which are the same) of iJTyper with different settings of top-k.

Iteration	StatType-SO				Short-SO			
	Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10
1	95.8 %	95.7 %	95.7 %	95.8 %	90.9 %	90.9 %	90.9 %	90.9 %
2	**97.5 %**	**97.5 %**	**97.5 %**	**97.5 %**	**92.5 %**	**92.5 %**	**92.5 %**	**92.5 %**
3	97.4 %	97.4 %	97.4 %	97.4 %	92.5 %	92.5 %	92.5 %	92.5 %
4	97.3 %	97.3 %	97.3 %	97.3 %	–	–	–	–
5	97.3 %	97.3 %	97.3 %	97.3 %	–	–	–	–

Table 6
Average precision and recall (which are the same) of iJTyper-SC on two datasets.

Iteration	StatType-SO	Short-SO
	Precision/Recall	Precision/Recall
1	86.6 %	84.3 %
2	86.8 %	84.5 %
3	86.7 %	84.5 %
4	86.8 %	–
5	86.7 %	–

The setting of k has negligible impacts on the performance of iJTyper. As listed in Table 5, under different settings of k, the average precision/recall values of iJTyper only have a little difference at the first iteration on StatType-SO and remain the same in all the other cases. This result indicates that the setting of k nearly has no impact on the performance of iJTyper. We analyzed the intermediate results produced by MLMTyper and found that most of the candidate types predicted by MLMTyper are haphazardly generated. When refining the top-k candidates for each API, iJTyper filters out the invalid candidates that do not exist in the pre-built KB of SnR. As a result, the refined top-k candidate types often contain only 0–2 types. Moreover, if MLMTyper could predict the right type of an API, the right type is often the top-1 candidate. These are the reasons why the different settings of k do not affect the performance of iJTyper. Based on the analysis, we set $k = 3$ in the subsequent experiments to ensure generality.

In addition, we validated the efficacy of our strategy that removes hallucinated types predicted by the statistics-based method based on the pre-built KB of the constraint-based method. By comparing iJTyper's performance with and without this strategy, we observe that the average precision/recall of iJTyper decreases from 97.3 % (with the strategy) to 94.9 % (without the strategy) on StatType-SO. In Short-SO, the average precision/recall decreases from 92.5 % (with the strategy) to 91.2 % (without the strategy). The degradation mainly comes from the reduced average precision/recall of the statistics-based method integrated by iJTyper. These results confirm the effectiveness of our strategy that removes hallucinated types predicted by the statistics-based method.

Executing M^s before M^c in iJTyper leads to lower performance. Table 6 presents the average precision/recall of iJTyper-SC on both datasets. We observe a performance oscillation between 86.8 % and 86.7 % on StatType-SO, which is caused by the inference of a class switched back and forth between the right type, *org.hibernate.cfg.Configuration*, and a wrong type, *java.util.Configuration*. Moreover, the average precision/recall of iJTyper-SC on both datasets is much worse than that of iJTyper, i.e., 86.7 % versus 97.3 % on StatType-SO and 84.5 % versus 92.5 % on Short-SO. We tracked the inference process and found that MLMTyper fails to predict the right types for a considerable number of APIs at the beginning, which has negative impacts on the KB reduction and thus affects the performance of SnR and iJTyper.

4.3. RQ2: Can iJTyper improve SOTA type inference methods?

Motivation. iJTyper aims to improve type inference by integrating existing constraint- and statistics-based methods. As a demonstration, we implemented a version of iJTyper using two SOTA methods, SnR and MLMTyper. It is necessary to evaluate the effectiveness of iJTyper by comparing it with the two methods. Moreover, we want to examine whether iJTyper could achieve better results than the recently popular and widely used tool, ChatGPT.

Approach. We applied SnR, MLMTyper, and iJTyper to each code snippet in StatType-SO and Short-SO. For ChatGPT, we used the prompt template shown in Fig. 7 to obtain the types (i.e., FQNs) of APIs that we focus on (i.e., classes and interfaces) in each code snippet. For example, Table 7 presents the types of APIs in the code snippet shown in Fig. 7 inferred using the four methods. Based on the inference results, we measured the average precision and recall of each method and tested the statistical significance of the performance difference between iJTyper and the three baselines (see Section 4.1.3).

Results. iJTyper outperforms SnR and MLMTyper in terms of both precision and recall. Table 8 presents the average precision and recall of iJTyper and the baselines. iJTyper achieves the highest average precision/recall of 97.3 % and 92.5 % on both datasets. The performance of MLMTyper is much lower than that (i.e., 91 %) of MLMTyper reported in (Huang et al., 2022). This is because although MLMTyper can predict right types for most APIs in the top-3 candidates, the right types do not always appear in the top-1. iJTyper outperforms SnR with respect to almost all the six libraries on both datasets. For example, SnR often makes mistakes with *java.util.Date* and *java.sql.Date*, while iJTyper can distinguish between these types. iJTyper improves the average recall of SnR and MLMTyper by at least 7.3 % and 27.4 %, respectively. In terms of average precision, iJTyper improves SnR by 0.9 % on StatType-SO and by 0.6 % on Short-SO.

iJTyper outperforms ChatGPT in terms of precision and recall on both datasets. As listed in Table 8, iJTyper improves the average precision/recall of ChatGPT by 3.2 % and 0.5 % on StatType-SO and Short-SO, respectively. The improvement on StatType-SO is associated with p-values less than 0.05. Specifically, iJTyper outperforms ChatGPT on the code snippets related to the three libraries, i.e., GWT, Hibernate, and XStream, in StatType-SO, but is slightly worse than ChatGPT on the code snippets related to Android and Joda Time. For Hibernate, ChatGPT often mistakenly infers the types under *org.hibernate* as those under *javax*, e.g., *org.hibernate.validator.AssertTrue* and *javax.validation.constraints.AssertTrue*. The latter type is a general standard while the former type is a specific implementation of the standard under Hibernate. ChatGPT cannot accurately determine which package is expected in different cases, so it chooses the general one. For XStream, ChatGPT often incorrectly infers the types under *com.thoughtworks.xstream.converters*. As an example, *com.thoughtworks.xstream.converters.MarshallingContext* is mistakenly inferred as *com.thoughtworks.xstream.core.MarshallingContext* which is haphazardly generated due to the hallucination problem of ChatGPT. A possible explanation is that there is not enough information about this package in the training corpus to make ChatGPT remember the types under it. iJTyper performs better in these situations because of its integration of the knowledge employed by both constraint- and statistics-based methods.

4.4. RQ3: What are the contributions of the code context augmentation and KB reduction mechanisms used in iJTyper?

Motivation. In iJTyper, we propose two key mechanisms: 1) *Code context augmentation* which is used to augment the context of the input code snippet for improving the statistics-based method; and 2) *KB reduction* which is used to filter irrelevant types of APIs in the pre-built KB of the constraint-based method and thus improve the performance of the method. We need to validate whether both mechanisms actually improve the corresponding methods.

Approach. After performing iJTyper on each code snippet in the two datasets, we measured the precision and recall of the two constituent methods inside iJTyper to study the promotion efficacy of both mechanisms. The internal constraint- and statistics-based methods are referred

Table 7
Type inference results of four methods on the example code snippet shown in Fig. 7.

API	SnR		MLMTyper	
ArrayList[3,1]	*java.util.ArrayList*	✔	*java.util.ArrayList*	✔
ArrayList[3,2]	*java.util.ArrayList*	✔	*java.util.ArrayList*	✔
Document[4,1]	–		*com.google.gwt.user.client.ui.Document*	✗
Element[6,1]	*com.google.gwt.user.client.Element*	✔	*javax.swing.text.Element*	✗
DOM[6,1]	*com.google.gwt.user.client.DOM*	✔	*javax.swing.text.DOM*	✗
API	ChatGPT		iJTyper	
ArrayList[3,1]	*java.util.ArrayList*	✔	*java.util.ArrayList*	✔
ArrayList[3,2]	*java.util.ArrayList*	✔	*java.util.ArrayList*	✔
Document[4,1]	*com.google.gwt.dom.client.Document*	✔	*com.google.gwt.dom.client.Document*	✔
Element[6,1]	*com.google.gwt.dom.client.Element*	✗	*com.google.gwt.user.client.Element*	✔
DOM[6,1]	*com.google.gwt.dom.client.DOM*	✗	*com.google.gwt.user.client.DOM*	✔

Table 8
Average precision and recall of four methods on two datasets. Note that for MLMTyper, ChatGPT, and iJTyper, their respective average precision and recall are the same. The data shared with Tables 9 and 10 are marked using the same background colors for correspondence.

Library	StatType-SO					Short-SO				
	MLMTyper Precision/Recall	SnR Precision	SnR Recall	ChatGPT Precision/Recall	iJTyper Precision/Recall	MLMTyper Precision/Recall	SnR Precision	SnR Recall	ChatGPT Precision/Recall	iJTyper Precision/Recall
Android	71.9%	98.4%	92.6%	97.9%	97.2%	69.1%	88.7%	85.5%	95.8%	89.1%
GWT	78.7%	97.5%	86.6%	96.0%	98.9%	68.9%	87.8%	80.0%	92.7%	86.7%
Hibernate	42.3%	96.4%	95.1%	88.7%	93.8%	38.6%	89.3%	87.7%	70.4%	89.5%
JDK	86.9%	100.0%	92.4%	100.0%	100.0%	90.7%	98.0%	92.6%	100.0%	98.2%
Joda Time	59.8%	94.8%	82.9%	98.8%	96.0%	66.7%	94.6%	72.9%	100.0%	100.0%
XStream	55.8%	91.5%	90.4%	83.0%	98.0%	56.5%	92.8%	90.6%	93.2%	91.8%
Average	65.9%***	96.4%**	90.0%***	94.1%*	97.3%	65.1%***	91.9%*	84.9%***	92.0%	92.5%

***p<0.001, **p<0.01, *p<0.05

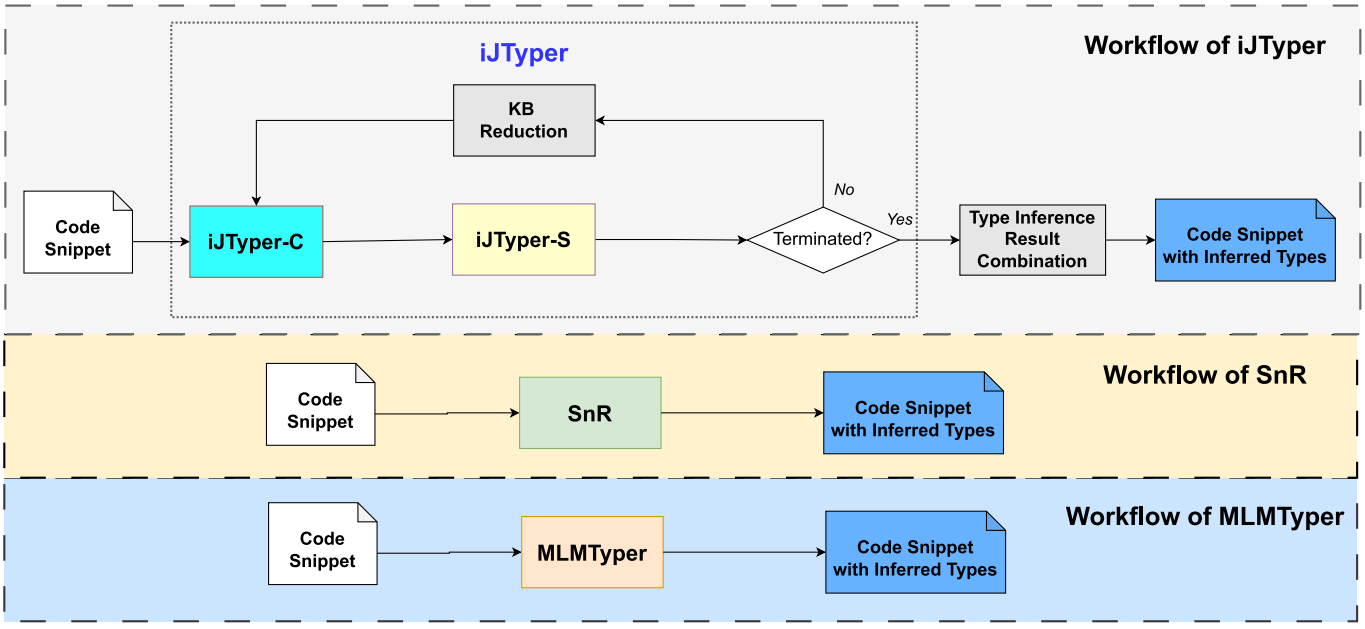

Fig. 9. Comparison among iJTyper-C, iJTyper-S, SnR, and MLMTyper.

to as iJTyper-C and iJTyper-S, respectively. Notice that these two methods are different from the original SnR and MLMTyper. Specifically, the inference results of iJTyper-C and iJTyper-S are improved by leveraging the two mechanisms during the iterations of iJTyper. Their difference are shown in Fig. 9. For each method, we combined the final successfully inferred types of APIs in a code snippet, as described in Section 3.5. Then, we measured the average precision and recall of both methods with respect to each library and each dataset. We also tested the statistical significance of the performance difference between iJTyper-C (*resp.* iJTyper-S) and its original version, SnR (*resp.* MLMTyper).

Results. The code context augmentation and KB reduction mechanisms can promote the constraint- and statistics-based methods. Tables 9 and 10 present the average precision and recall, respectively, of iJTyper-C, iJTyper-S, SnR, and MLMTyper on both datasets.

Table 9
Average precision of iJTyper-C and iJTyper-S in comparison with SnR and MLMTyper. The data shared with Tables 8 and 10 are marked using the same background colors for correspondence.

Library	StatType-SO				Short-SO			
	MLMTyper	iJTyper-S	SnR	iJTyper-C	MLMTyper	iJTyper-S	SnR	iJTyper-C
Android	71.9%	82.4%	98.4%	99.0%	69.1%	85.5%	88.7%	90.6%
GWT	78.7%	93.0%	97.5%	98.8%	68.9%	75.6%	87.8%	90.5%
Hibernate	42.3%	61.6%	96.4%	94.4%	38.6%	71.9%	89.3%	91.1%
JDK	86.9%	99.5%	100.0%	100.0%	90.7%	98.2%	98.0%	98.0%
Joda Time	59.8%	94.5%	94.8%	97.2%	66.7%	97.9%	94.6%	100.0%
XStream	55.8%	86.9%	91.5%	98.4%	56.5%	87.1%	92.8%	92.8%
Average	65.9%***	86.3%	96.4%**	98.0%	65.1%***	86.0%	91.9%	93.8%

***p<0.001, **p<0.01, *p<0.05

Table 10
Average recall of iJTyper-C and iJTyper-S in comparison with SnR and MLMTyper. The data shared with Tables 8 and 9 are marked using the same background colors for correspondence.

Library	StatType-SO				Short-SO			
	MLMTyper	iJTyper-S	SnR	iJTyper-C	MLMTyper	iJTyper-S	SnR	iJTyper-C
Android	71.9%	82.4%	92.6%	93.2%	69.1%	85.5%	85.5%	87.3%
GWT	78.7%	93.0%	86.6%	92.2%	68.9%	75.6%	80.0%	84.4%
Hibernate	42.3%	61.6%	95.1%	93.1%	38.6%	71.9%	87.7%	89.5%
JDK	86.9%	99.5%	92.4%	92.9%	90.7%	98.2%	92.6%	92.6%
Joda Time	59.8%	94.5%	82.9%	86.4%	66.7%	97.9%	72.9%	77.1%
XStream	55.8%	86.9%	90.4%	98.0%	56.5%	87.1%	90.6%	90.6%
Average	65.9%***	86.3%	90.0%***	92.7%	65.1%***	86.0%	84.9%*	86.9%

***p<0.001, **p<0.01, *p<0.05

Note that the precision in Table 9 is calculated using the number of types that each method can actually infer as the denominator (see Section 4.1.3). The denominator values are different in precision and recall calculations for constraint-based methods (i.e., iJTyper-C, SnR) as they cannot provide answers for certain APIs. Thus, the precision and recall of constraint-based methods are not equal.

iJTyper-C increases the average precision of SnR by 1.6 % on StatType-SO and by 1.9 % on Short-SO. In terms of average recall, iJTyper-C improves SnR by 2.7 % and 2.0 % on StatType-SO and Short-SO, respectively. Most of the improvements are associated with p-values less than 0.05 except for the average precision on Short-SO. Although iJTyper-C achieves higher precision, this does not necessarily imply superior overall performance compared to iJTyper, taking its lower recall (i.e., less inferred types) into account. iJTyper-S improves MLMTyper by 20.4 % on StatType-SO and by 20.9 % on Short-SO in terms of average precision/recall. All these results demonstrate that the code context augmentation and KB reduction mechanisms used in iJTyper can promote the performance of the constituent constraint- and statistics-based methods.

5. Discussion

Evaluation of k in Larger Code Snippets. Although our study primarily focuses on short code snippets from SO, we still constructed a new dataset with larger code snippets to further evaluate the impact of the setting of k. Specifically, we randomly selected 10 Java class files from each of six libraries (Android, GWT, Hibernate, JDK, Joda Time, and XStream). The selection was performed without filtering by functionality (e.g., utilities vs. domain-specific classes) in order to obtain a representative but unbiased sample. This process resulted in a new dataset named Lib-Sources, comprising a total of 60 Java classes. In this dataset, each class file contains an average of 192.6 lines of code, with the smallest file comprising 34 lines and the largest 804 lines. Among them, 14 files (23.3 %) fall within 34–100 lines, 27 files (45.0 %) within 101–200 lines, 18 files (30.0 %) within 201–500 lines, and only 1 file (1.7 %) exceeds 500 lines. In contrast, the average lines of code per snippet in StatType-SO is 28, while all code snippets in Short-SO contain less than 3 lines of code. Similar to Section 4.2, we applied iJTyper to each code snippet using different settings of $k \in \{1, 3, 5, 10\}$. The experimental results show that for any $k \in \{1, 3, 5, 10\}$, iJTyper achieves identical precision/recall per iteration on the Lib-Sources dataset: 65.7 % in the first iteration, and 66.5 % in both the second and third iterations. This further demonstrates that the setting of k has a negligible impact on the performance of iJTyper.

Global Correctness Statistics. The inference results in a given code snippet are mutually dependent. The result at one location may potentially affect the inference at another. To evaluate the capability of each method in achieving globally accurate type inference, we quantified the number of code snippets for which the inference results attained 100 % precision/recall, and calculated the corresponding percentage relative to the entire dataset. Results are shown in Table 11. In StatType-SO, the comparative results of the three methods in terms of 100 % recall percentage (iJTyper > ChatGPT > MLMTyper > SnR) and 100 % precision percentage (iJTyper > SnR > ChatGPT > MLMTyper) are consistent with their ranking in Table 8. iJTyper has the most code snippets with fully correct inferences. In Short-SO, the 100 % recall/precision percentage of ChatGPT (73.3 %) is slightly higher than that of iJTyper (71.7 %).

Intermediate Process Statistics. iJTyper employs an iterative inference process, in which the results produced by the constraint-based

Table 11
Statistics of code snippets with 100 % recall or precision.

Dataset	#Dataset Snippets	Method	#100 % Recall Snippets	100 % Recall Percentage	#100 % Precision Snippets	100 % Precision Percentage
StatType-SO	268	iJTyper	231	86.2 %	231	86.2 %
		ChatGPT	219	81.7 %	219	81.7 %
		SnR	55	20.5 %	220	82.1 %
		MLMTyper	177	66.0 %	177	66.0 %
Short-SO	120	iJTyper	86	71.7 %	86	71.7 %
		ChatGPT	88	73.3 %	88	73.3 %
		SnR	39	32.5 %	83	69.2 %
		MLMTyper	71	59.2 %	71	59.2 %

method are assigned higher priority during result combination. We analyzed the intermediate results of iJTyper to examine whether the constraint-based method effectively filters out error types introduced by the statistics-based method, as well as the degree to which the intermediate error types propagate across iterations. Specifically, we examined the results produced in the final iteration by both methods, checking how many error types were successfully filtered out by the constraint-based method and how many cases represented erroneous filtering (i.e., cases where correct types were erroneously replaced with incorrect ones). Moreover, for each file, we compared the inference results from the first iteration with those from the final iteration to quantify how many erroneous results were corrected in subsequent iterations and how many persisted.

In most cases, iJTyper correctly filtered out the error types. On StatType-SO and Short-SO, the constraint-based method successfully filtered out 205 and 25 error types introduced by the statistics-based method, respectively. In contrast, cases where the constraint-based method incorrectly replaced the correct types inferred by the statistics-based method occurred only 6 times in StatType-SO and 3 times in Short-SO, which is substantially fewer than the successful cases.

A substantial portion of initial inference errors persisted across iterations, with correction rates varying significantly between different datasets. In StatType-SO, a total of 75 APIs (100 %) were incorrectly inferred in the first iteration, of which 33 error types (44.0 %) were corrected in subsequent iterations, while 42 error types (56 %) persisted until the final iteration. In Short-SO, the corresponding numbers were 28 (100 %), 4 (14.3 %), and 24 (85.71 %), respectively. One possible explanation is that longer code provides more opportunities for iJTyper to leverage its error-correction capabilities.

Impact of Library and Version Differences. iJTyper leverages a knowledge base constructed for the six libraries, which are constrained to a limited set of libraries and versions. In contrast, ChatGPT performs type inference without access to library-specific information, meaning the exact library and version of a code snippet are often not specified. Since multiple versions may define distinct but semantically valid type candidates, it is possible that ChatGPT's predictions are penalized as incorrect but would be reasonable under another version. We manually reviewed all the results in ChatGPT's evaluation and found that only the inference of *Instant* in Short-SO was affected. The original ground truth was *org.joda.time.Instant*, but we determined that ChatGPT's prediction of *java.time.Instant* is also acceptable. Taking this into account, ChatGPT's overall recall on Short-SO increases from 92.01 % to 92.35 %, which remains slightly lower than iJTyper's 92.52 %. Therefore, the impact of library and version differences on ChatGPT's evaluation is limited in our study.

Practicality. Type inference of APIs in code snippets is necessary to perform a wide range of tasks with the code snippets, such as API usage mining (Uddin et al., 2020; Zhang et al., 2018), code search (Maji et al., 2021; Thummalapenta & Xie, 2007), and code repair (Mesbah et al., 2019; Terragni et al., 2016). As demonstrated in our evaluation, iJTyper achieves high precision and recall (see Table 8) and thus can effectively support these general tasks. Moreover, iJTyper shows potential in restricted scenarios that demand data privacy, controllability, and deployment flexibility. In confidential software projects, the source code cannot be uploaded to the cloud due to privacy requirements, which limits the use of large-scale pre-trained models such as ChatGPT. iJTyper employs a lightweight masked language model with only 125 million parameters, enabling low-cost local deployment and providing dependable support for type inference and downstream applications such as code refactoring in such scenarios. Furthermore, real-world software projects often rely on custom or private libraries, which are seldom covered by the training data of LLMs, potentially leading to degraded performance. In contrast, iJTyper offers greater flexibility by allowing preemptive extension of its knowledge base to encompass such libraries, thereby complementing the capabilities of LLM-based type inference methods.

Extensibility. iJTyper integrtes type inference methods in a non-intrusive manner, thereby can be extended to incorporate various constraint- and statistics-based methods. In addition to its current implementation based on SnR and MLMTyper, we have explored and evaluated its integration with additional methods, i.e., a constraint-based method, Baker (Subramanian et al., 2014) and a statistics-based method, ChatGPT (OpenAI, 2023). We implemented three additional versions of iJTyper, namely iJTyper-BM (Baker + MLMTyper), iJTyper-BC (Baker + ChatGPT), and iJTyper-SnC (SnR + ChatGPT). The implementations involved 84 lines of code modifications, primarily consisting of encapsulating the new methods into separate functions and adapting the input/output formats. If the underlying baselines evolve in the future, only adjustments to their input/output formats would be necessary, without requiring substantial additional modifications. We measured the precision and recall of Baker and the three additional iJTyper versions on StatType-SO and Short-SO. Moreover, we compared these versions with CKTyper (Li et al., 2025). The results are shown in Table 12.

In most cases, iJTyper is able to improve the precision and recall of both integrated methods. All iJTypers in Short-SO, along with iJTyper-SnC in StatType-SO, achieve higher precision and recall compared to their respective baselines. In StatType-SO, iJTyper-BM improves recall by 51.0 % compared to Baker and by 15.5 % over MLMTyper. While its precision slightly drops 4.7 % compared to Baker, Baker's low recall (30.4 %) means that its higher precision was achieved on far fewer inferred types. The enhanced constraint-based component in iJTyper-BM achieves 90.4 % precision, still a 4.3 % improvement over Baker. For iJTyper-BC in StatType-SO, it improves both the precision (by 5.4 %) and recall (by 61.1 %) of Baker. Its precision/recall slightly lower than that of ChatGPT by 2.6 %, primarily due to Baker's low precision (73.2 %) on the GWT library, which adversely impacts the overall performance after integration. Compared to CKTyper, a statistics-method solely based on ChatGPT, iJTyper-SnC (which integrates SnR and

Table 12
Performance of Baker, CKTyper and different implementations of iJTyper.

Library	StatType-SO						Short-SO					
	Baker Precision	Baker Recall	CKTyper Precision/Recall	iJTyper-BM Precision/Recall	iJTyper-BC Precision/Recall	iJTyper-SnC Precision/Recall	Baker Precision	Baker Recall	CKTyper Precision/Recall	iJTyper-BM Precision/Recall	iJTyper-BC Precision/Recall	iJTyper-SnC Precision/Recall
Android	91.5%	16.6%	99.3%	71.8%	87.8%	99.3%	74.1%	36.4%	100.0%	81.8%	85.4%	89.6%
GWT	73.2%	33.2%	99.1%	73.1%	81.0%	98.1%	85.7%	80.0%	97.7%	91.1%	97.6%	100.0%
Hibernate	84.3%	48.5%	96.3%	70.6%	88.3%	96.7%	73.1%	66.7%	84.2%	77.2%	88.9%	90.7%
JDK	90.9%	4.7%	100.0%	91.0%	99.3%	100.0%	92.7%	70.4%	100.0%	98.2%	97.7%	100.0%
Joda Time	93.2%	41.2%	99.4%	93.5%	99.4%	99.4%	100.0%	91.7%	95.8%	97.9%	100.0%	100.0%
XStream	83.5%	38.2%	93.3%	88.2%	93.2%	99.1%	95.0%	89.4%	97.2%	91.8%	93.1%	95.8%
Average	86.1%	30.4%	97.8%	81.4%	91.5%	98.8%	86.8%	72.4%	95.5%	89.7%	93.8%	96.0%

ChatGPT) achieves 1.0 % higher precision/recall on StatType-SO and 0.5 % higher on Short-SO.

Although iJTyper-SnC outperforms the version of iJTyper that combines SnR and MLMTyper (improving precision/recall by 1.5 % on StatType-SO and 3.5 % on Short-SO), we note that ChatGPT has a much larger parameter size (175B) (WinnieNwanne, 2024) compared to MLMTyper (125M) (Feng et al., 2020; Huang et al., 2022) and is difficult to deploy locally. Due to this disparity, the comparison is not entirely fair, and therefore we do not treat iJTyper-SnC as the main implementation of iJTyper.

Maintainability. The knowledge base of iJTyper is currently constructed using specific library versions that correspond to the datasets. When the underlying libraries evolve, the updated versions can be used to rebuild the knowledge base, thereby mitigating maintenance issues. In addition, the type inferred by iJTyper for a given simple name is influenced by both the types present in the KB and the context of the code snippet. While information about the project's Java version may implicitly affect the context, iJTyper's type predictions could vary depending on the Java version of the target code snippet.

Ethical Considerations. In our study, we do not consider cases involving the recommendation of deprecated or insecure APIs. Instead, our focus is on evaluating the correctness of the inferred types—whether they align with the answers provided in the dataset (i.e., original Stack Overflow posts)—and on improving performance with respect to this task. To mitigate such risks in iJTyper, a list of insecure types could be maintained and excluded from the knowledge base in future work. A promising direction for future research is to analyze the APIs used in Stack Overflow posts and detect instances of such deprecated or insecure APIs.

Limitation. iJTyper iteratively integrates two type inference methods, leading to the limitation in efficiency. We performed an implementation of iJTyper that integrates SnR and MLMTyper on two open-source datasets, StatType-SO and Short-SO, for three times. We measured the average time cost of four main components of iJTyper, including the two constituent methods and the code context augmentation and KB reduction mechanisms, spent on a code snippet in each dataset, as presented in Table 13. On average, iJTyper takes about 50.82s and 93.45s to analyze a code snippet in Short-SO and StatType-SO, respectively. The time cost of the two mechanisms used to promote the constituent methods is minimal. In addition, iJTyper exhibits its longest average time cost of 143.50s on the Lib-Sources dataset. These results indicate that iJTyper's time cost is influenced by code length, with longer code snippets requiring more processing time on average.

Regarding the time cost of individual baselines, SnR achieves an average time cost of 10.60s on StatType-SO (8.8× faster than iJTyper) and 4.18s on Short-SO (12.2× faster), while MLMTyper records 12.43s (7.5× faster) and 7.50s (6.8× faster) on the respective datasets. Thus, **iJTyper is more suitable for application scenarios that require high quality of inference results but do not require high efficiency.** Specifically, iJTyper could support the off-line type inference for tasks related to incomplete Java code snippets, such as API usage mining, API evolution analysis, and code search. **It is worth noticing that iJTyper can achieve relatively high recall (see Table 5) after the first iteration, allowing users to trade off between performance and efficiency.** Moreover, although iJTyper combines the strengths of constraint- and statistics-based methods, it cannot overcome some problems inherent in both methods, e.g., the out-of-vocabulary issue. That is, iJTyper cannot infer the types of APIs which are neither included in the pre-built KB of the constraint-based method nor exist in the training corpus of the statistics-based method.

Threats to Validity. Threats to internal validity relate to the errors in the implementation of iJTyper and the baselines. We implemented the three baselines, i.e., SnR, MLMTyper, and ChatGPT, using their replication packages or the official implementation. For iJTyper that integrates SnR and MLMTyper, we double-checked the implementation code and

Table 13
Average time cost (in seconds) for analyzing a code snippet by ChatGPT, iJTyper, and the four main components of iJTyper.

Library	StatType-SO						Short-SO					
	SnR	MLMTyper	Code Context Augmentation	KB Reduction	iJTyper iJTyper	ChatGPT ChatGPT	SnR SnR	MLMTyper MLMTyper	Code Context Augmentation	KB Reduction	iJTyper iJTyper	ChatGPT ChatGPT
Android	46.82	29.36	1.60E-04	5.03	92.58	1.28	24.66	15.66	9.21E-05	2.47	46.93	1.15
GWT	54.54	38.64	1.55E-04	6.54	106.96	1.47	24.13	15.48	7.89E-05	3.38	47.67	1
Hibernate	47.06	32.86	2.45E-04	5.91	92.90	1.27	25.17	16.38	8.86E-05	3.36	51.47	1.03
JDK	58.42	43.63	1.61E-04	10.93	126.34	1.28	24.04	16.13	1.03E-04	2.68	49.51	1.05
Joda Time	31.27	21.22	1.25E-04	2.72	60.14	1.14	27.55	17.99	6.20E-05	2.44	53.52	1.02
XStream	39.33	28.05	2.74E-04	5.32	81.78	1.37	24.24	19.68	9.74E-05	4.11	55.84	1.18
Average	46.24	32.29	1.87E-04	6.08	93.45	1.30	24.96	16.89	8.70E-05	3.07	50.82	1.07

tracked the inputs, intermediate results, and outputs of several code snippets to ensure correctness.

Threats to external validity relate to the generalizability of the results. We conducted the experiments on two open-source datasets of Java code snippets related to six popular libraries. Both datasets are widely used in prior studies (Dong et al., 2022; Huang et al., 2022; Phan et al., 2018; Saifullah et al., 2019; Velázquez-Rodríguez et al., 2023). The code snippets in both datasets are characterized by different lengths, and the six libraries have different applications. Based on these features, our evaluation results could have a good generalizability.

6. Related work

In this section, we review existing work on type inference for incomplete Java codes and discuss the contribution of our iJTyper in comparison with them. Moreover, we introduce several representative work on type inference for dynamically typed languages, e.g., Python and JavaScript, which are in a domain similar to but different from ours.

6.1. Type inference for incomplete java codes

Existing type inference methods proposed for incomplete Java codes can be categorized as constraint-based and statistics-based.

Constraint-based Type Inference Methods. Subramanian et al. (2014) proposed Baker which maintains a list of candidate types for each API in a code snippet and then reduces the candidates based on the constraints in the code snippet using heuristic rules. Shokri (2021) proposed DepRes which generates a type sketch for each API and retrieves possible candidates from the pre-built KB. Then, it uses a Z3 SMT solver to solve the SMT problem translated from the type constraints and determine the final types of sketches. Dong et al. (2022) proposed SnR which first tries to repair an input code snippet into a syntactically valid compilation unit using a template-based method in order to generate an AST of the unit. Then, it extracts the constraints from the AST and utilizes Datalog (Ullman, 1984) as the constraint solver. Finally, SnR ranks the candidate types of APIs based on a prioritization heuristic and chooses the top item as the final result. These constraint-based methods generally pre-build a KB from a set of API libraries and require a partial AST of the input code snippet for type inference. The performance of these methods is limited by several factors, e.g., the syntactic incompleteness of code snippets and the API libraries covered by the pre-built KB, as detailed in Section 2.2.1.

Statistics-based Type Inference Methods. Phan et al. (2018) proposed StatType which solves type inference tasks using a machine translation model. It considers two kinds of contexts of an API whose type needs to be inferred, namely 1) type context which refers to the classes, methods, and fields that occur around an API; and 2) resolution context which refers to the type inference decision of the surrounding APIs. Saifullah et al. (2019) proposed COSTER which uses both local and global contexts to calculate a likelihood score, a context similarity score, and a name similarity score based on an occurrence likelihood dictionary to infer the types of APIs. Velázquez-Rodríguez et al. (2023) proposed RESICO which views type inference as a text classification task. It vectorizes APIs with contexts based on the Word2Vec model and then infers the types using a classifier. Huang et al. (2022) proposed MLMTyper which infers types by transforming the type inference problem into a fill-in-blank task and aligning it with the prompt-tuning of a large pre-trained code model. Li et al. (2025) proposed CKTyper, which retrieves context information useful for type inference from Stack Overflow to improve ChatGPT's type inference performance. Although these statistics-based methods can get rid of the syntactic limitation of code snippets, they may suffer from low precision as they rarely leverage type constraints in code snippets. More limitations of these methods are described in Section 2.2.2.

6.2. Type inference for dynamically typed languages

Many type inference methods have also been proposed for dynamically typed languages, a scope that slightly differs from our research. Some representative ones are briefly described as follows.

Hackett and Guo (2012) proposed a type inference method for JavaScript, which can generate efficient and type-specialized machine codes by combining a points-to analysis and a just-in-time compiler. Hassan et al. (2018) proposed TYPPETE for Python. It treats the type inference task as a MAXSMT problem and attempts to find a solution that satisfies both mandatory constraints and the maximum number of optional constraints in code snippets. Malik et al. (2019) proposed NL2Type which utilizes natural language information in code snippets, including comments, function names, and parameter names, to predict types of JavaScript functions. It builds a recurrent LSTM network and formulates type inference as a classification problem. Wei et al. (2020) proposed LambdaNet which generates a type dependency graph (TDG) for TypeScript codes and uses a graph neural network to infer types. These methods can also be broadly categorized as constraint-based (Hackett & Guo, 2012; Hassan et al., 2018) or statistics-based (Malik et al., 2019; Wei et al., 2020).

In recent years, several hybrid type inference methods have been proposed for dynamically typed languages by integrating the strengths of constraint- and statistics-based methods. For example, Pandi et al. (2020) proposed OptTyper for TypeScript. It generates type constraints via static analysis and translates them into numerical representations. Then, OptTyper estimates type prediction probability using a machine learning model and combines it with the translated constraints to form an optimization problem. The final types are inferred by finding the minimum of the optimization function. Peng et al. (2022) proposed HiTyper for Python, which obtains type recommendations from a neural network and performs static inference and type rejection on a TDG built from Python codes. (Ye et al., 2023) proposed a type inference method for Python. It encodes the type predictions from the machine learning model as constraints and feeds the encoded constraints into an SMT solver to obtain final type inference results.

The integration strategies used by existing hybrid methods can be classified as: 1) representing the information gained from constraint- and statistics-based methods in a unified manner and then solving the

unified problem (Pandi et al., 2020; Ye et al., 2023); 2) incorporating a statistical learning step into a constraint-based method (Peng et al., 2022). They all require invasive modifications within either the constraint-based, the statistics-based, or both methods. This limits their flexibility to integrate different constraint- and statistics-based methods. In contrast, iJTyper integrates both methods in a non-intrusive way without modifying their implementation, which makes it easier to integrate different methods. Moreover, the key idea of iJTyper to integrate and improve constraint- and statistics-based methods through the code context augmentation and KB reduction mechanisms may be employed to design better type inference methods for dynamically typed languages.

7. Conclusion and future work

We propose iJTyper, an effective and easy-to-use type inference framework for integrating constraint- and statistics-based methods. iJTyper iteratively executes both methods and performs two mechanisms, i.e., code context augmentation and KB reduction, to improve the performance of both methods. After the iterative process, iJTyper produces the final inference results by combining the inference results of both methods. For evaluation, we implemented a version of iJTyper by integrating two SOTA methods, SnR and MLMTyper. Evaluation results on two open-source datasets demonstrate that iJTyper successfully integrated the advantages of constraint- and statistics-based methods and achieved the highest precision/recall of 97.3 % and 92.5 % on the two datasets, outperforming SnR and MLMTyper. iJTyper also achieved better performance than the recently popular tool, ChatGPT.

In future work, we plan to implement and evaluate other versions of iJTyper by integrating more constraint- and statistics-based type inference methods proposed for code snippets written in different programming languages, including Java, Python, and JavaScript. In addition, we decide to apply iJTyper to improve the performance of some software engineering tasks, such as API usage mining and code repair.

CRediT authorship details

Zhixiang Chen: Conceptualization, Methodology, Software, Investigation, Writing - Original Draft, Writing - Review & Editing; **Anji Li:** Software, Investigation, Validation, Writing - Original Draft; **Neng Zhang:** Supervision, Conceptualization, Methodology, Writing - Original Draft, Writing - Review & Editing; **Jianguo Chen:** Writing - Review & Editing; **Yuan Huang:** Writing - Review & Editing; **Zibin Zheng:** Writing - Review & Editing

Data availability

Data will be made available on request.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

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

References

Bacchelli, A., Ponzanelli, L., & Lanza, M. (2012). Harnessing stack overflow for the IDE. In *2012 Third international workshop on recommendation systems for software engineering (RSSE)* (pp. 26–30). IEEE.

Barua, A., Thomas, S. W., & Hassan, A. E. (2014). What are developers talking about? An analysis of topics and trends in stack overflow. *Empirical Software Engineering, 19*, 619–654.

Dagenais, B., & Robillard, M. P. (2012). Recovering traceability links between an API and its learning resources. In *2012 34th international conference on software engineering (ICSE)* (pp. 47–57). IEEE.

Dong, Y., Gu, T., Tian, Y., & Sun, C. (2022). SnR: Constraint-based type inference for incomplete java code snippets. In *Proceedings of the 44th international conference on software engineering* (pp. 1982–1993).

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020* (pp. 1536–1547).

Hackett, B., & Guo, S.-y. (2012). Fast and precise hybrid type inference for javascript. In *Proceedings of the 33rd ACM SIGPLAN conference on programming language design and implementation* (pp. 239–250).

Hassan, M., Urban, C., Eilers, M., & Müller, P. (2018). MaxSMT-based type inference for python 3. In *Computer aided verification: 30th international conference, CAV 2018, held as part of the federated logic conference, floc 2018, oxford, UK, july 14–17, 2018, proceedings, Part II 30* (pp. 12–19). Springer.

Hindle, A., Barr, E. T., Gabel, M., Su, Z., & Devanbu, P. (2016). On the naturalness of software. *Communications of the ACM, 59*(5), 122–131.

Horton, E., & Parnin, C. (2018). Gistable: Evaluating the executability of python code snippets on github. In *2018 IEEE International conference on software maintenance and evolution (ICSME)* (pp. 217–227). IEEE.

Huang, Q., Yuan, Z., Xing, Z., Xu, X., Zhu, L., & Lu, Q. (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* (pp. 1–13).

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., & Fung, P. (2023). Survey of hallucination in natural language generation. *ACM Computing Surveys, 55*(12), 1–38.

Kabir, A., Wang, S., Tian, Y., Chen, T.-H. P., Asaduzzaman, M., & Zhang, W. (2025). ZS4C: Zero-shot synthesis of compilable code for incomplete code snippets using LLMs. *ACM Trans. Softw. Eng. Methodol., 34*(4), 30.

Li, A., Zhang, N., Zou, Y., Chen, Z., Wang, J., & Zheng, Z. (2025). CKTyper: Enhancing type inference for java code snippets by leveraging crowdsourcing knowledge in stack overflow. *Proceedings of the ACM on Software Engineering, 2*(FSE), 21.

Maji, S., Rout, S. S., & Choudhary, S. (2021). DCoM: A deep column mapper for semantic data type detection. arXiv:2106.12871.

Malik, R. S., Patra, J., & Pradel, M. (2019). NL2Type: Inferring javascript function types from natural language information. In *2019 IEEE/ACM 41st international conference on software engineering* (pp. 304–315). IEEE.

Mesbah, A., Rice, A., Johnston, E., Glorioso, N., & Aftandilian, E. (2019). Deepdelta: Learning to repair compilation errors. In *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering* (pp. 925–936).

[OpenAI] (2023). ChatGPT: Optimizing language models for dialogue. Accessed September 6, 2023 https://openai.com/blog/chatgpt.

Pandi, I. V., Barr, E. T., Gordon, A. D., & Sutton, C. (2020). Opttyper: Probabilistic type inference by optimising logical and natural constraints. arXiv:2004.00348.

Peng, Y., Gao, C., Li, Z., Gao, B., Lo, D., Zhang, Q., & Lyu, M. (2022). Static inference meets deep learning: A hybrid type inference approach for python. In *Proceedings of the 44th international conference on software engineering* (pp. 2019–2030).

Phan, H., Nguyen, H. A., Tran, N. M., Truong, L. H., Nguyen, A. T., & Nguyen, T. N. (2018). Statistical learning of API fully qualified names in code snippets of online forums. In *Proceedings of the 40th international conference on software engineering* (pp. 632–642).

Qiu, F., Gao, Z., Xia, X., Lo, D., Grundy, J., & Wang, X. (2021). Deep just-in-time defect localization. *IEEE Transactions on Software Engineering, 48*(12), 5068–5086.

Raychev, V., Vechev, M., & Yahav, E. (2014). Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation* (pp. 419–428).

Rosen, C., & Shihab, E. (2016). What are mobile developers asking about? A large scale study using stack overflow. *Empirical Software Engineering, 21*, 1192–1223.

Sadowski, C., Stolee, K. T., & Elbaum, S. (2015). How developers search for code: A case study. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (pp. 191–201).

Saifullah, C. M. K., Asaduzzaman, M., & Roy, C. K. (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)* (pp. 243–254). IEEE.

Shokri, A. (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)* (pp. 1388–1390). IEEE.

Shokri, A., & Mirakhorli, M. (2021). DepRes: A Tool for Resolving Fully Qualified Names and Their Dependencies. arXiv:2108.01165.

Subramanian, S., Inozemtseva, L., & Holmes, R. (2014). Live API documentation. In *Proceedings of the 36th international conference on software engineering* (pp. 643–652).

Svyatkovskiy, A., Zhao, Y., Fu, S., & Sundaresan, N. (2019). Pythia: AI-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining* (pp. 2727–2735).

Terragni, V., Liu, Y., & Cheung, S.-C. (2016). CSNIPPEX: Automated synthesis of compilable code snippets from Q&A sites. In *Proceedings of the 25th international symposium on software testing and analysis* (pp. 118–129).

Thummalapenta, S., & Xie, T. (2007). Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the 22nd IEEE/ACM international conference on automated software engineering* (pp. 204–213).

Uddin, G., Khomh, F., & Roy, C. K. (2020). Mining API usage scenarios from stack overflow. *Information and Software Technology*, *122*, 106277.

Ullman, J. D. (1984). Principles of database systems. Galgotia Publications.

Velázquez-Rodríguez, C., Di Nucci, D., & De Roover, C. (2023). A text classification approach to API type resolution for incomplete code snippets. *Science of Computer Programming*, *227*, 102941.

Wei, J., Goyal, M., Durrett, G., & Dillig, I. (2020). LambdaNet: Probabilistic type inference using graph neural networks. In *8th international conference on learning representations, ICLR 2020*.

Wilcoxon, F. (1992). Individual comparisons by ranking methods. In *Breakthroughs in statistics: Methodology and distribution* (pp. 196–202). Springer.

[WinnieNwanne] (2024). Comparing GPT-3.5 & GPT-4: A Thought Framework on When To Use Each Model. Accessed July 5, 2025 https://techcommunity.microsoft.com/blog/azure-ai-services-blog/comparing-gpt-3-5--gpt-4-a-thought-framework-on-when-to-use-each-model/4088645.

Yang, D., Hussain, A., & Lopes, C. V. (2016). From query to usable code: An analysis of stack overflow code snippets. In *Proceedings of the 13th international conference on mining software repositories* (pp. 391–402).

Ye, F., Zhao, J., Shirako, J., & Sarkar, V. (2023). Concrete type inference for code optimization using machine learning with SMT solving. *Proceedings of the ACM on Programming Languages*, *7*(OOPSLA2), 773–800.

Zhang, T., Du, Q., Xu, J., Li, J., & Li, X. (2020). Software defect prediction and localization with attention-based models and ensemble learning. In *2020 27th Asia-pacific software engineering conference (APSEC)* (pp. 81–90). IEEE.

Zhang, T., Upadhyaya, G., Reinhardt, A., Rajan, H., & Kim, M. (2018). Are code examples on an online Q&A forum reliable?: A study of API misuse on stack overflow. In *Proceedings of the 40th international conference on software engineering* (pp. 886–896). ACM.