Reproducibility of issues reported in stack overflow questions: Challenges, impact & estimation☆

Saikat Mondal *, Banani Roy

Department of Computer Science, University of Saskatchewan, Canada

ARTICLE INFO

ABSTRACT

Software developers often submit questions to technical Q&A sites like Stack Overflow (SO) to resolve code-level problems. In practice, they include example code snippets with questions to explain the programming issues. Existing research suggests that users attempt to reproduce the reported issues using given code snippets when answering questions. Unfortunately, such code snippets could not always reproduce the issues due to several unmet challenges that prevent questions from receiving appropriate and prompt solutions. One previous study investigated reproducibility challenges discussed in 400 Java questions and produced a catalog of them (e.g., too short code snippets). However, it is unknown how the practitioners (i.e., SO users) perceive this challenge catalog. Practitioners' perspectives are inevitable in validating these challenges and estimating their severity. In this study, we first surveyed 53 practitioners to understand their perspectives on reproducibility challenges. We attempt to (a) see whether they agree with these challenges, (b) determine the impact of each challenge on answering questions, and (c) identify the need for tools to promote reproducibility. Survey results show that – (a) about 90% of the participants agree with the challenges, (b) "missing an important part of code" most severely hurt reproducibility, and (c) participants strongly recommend introducing automated tool support to promote reproducibility. Second, we extract *nine* code-based features (e.g., LOC, compilability) and build five Machine Learning (ML) models to predict issue reproducibility. Early detection might help users improve code snippets and their reproducibility. Our models achieve 84.5% precision, 83.0% recall, 82.8% F1-score, and 82.8% overall accuracy, which are highly promising. We also validate the effectiveness of our features by predicting the reproducibility status of C# code snippets. Third, we systematically interpret the ML model and explain how code snippets with reproducible issues differ from those with irreproducible issues.

1. Introduction

Stack Overflow (SO) has emerged as one of the largest and most popular technical Q&A sites. This Q&A site is continuously contributing to the body of knowledge in software development (Tahaei et al., 2020; Treude et al., 2011; Vincent et al., 2018). Millions of software developers visit SO daily to solve their programming-related problems (Gao et al., 2021; StackExchange, 2019). Around 6K questions are posted in SO every day (StackExchange, 2019). Among them, many questions discuss code-level problems (e.g., errors, unexpected behavior) (Treude et al., 2011). In practice, such questions often include example code snippets to explain the code issues. Existing research suggests that SO users first attempt to reproduce the issues reported in the questions using the code snippets (Mondal et al., 2019). Upon success, they could submit their appropriate solutions. Reproducibility means a complete agreement between the reported and investigated issues (White et al.,

2015; Mondal et al., 2019). Unfortunately, such programming issues could not always be reproduced due to several unmet challenges of the given code snippets (Mondal et al., 2019; Moran et al., 2016; Soltani et al., 2017). This phenomenon prevents the questions from getting prompt and appropriate solutions. For example, Mondal et al. (2019) show that a question with a code snippet capable of reproducing the reported issue has more than three times higher chance of receiving an acceptable answer compared to a question with a code snippet unable to reproduce the issue. Moreover, the median time delay of receiving an accepted answer is double, and the average number of answers is significantly less for questions with irreproducible issues than those with reproducible issues.

Two existing studies (Yang et al., 2016; Horton and Parnin, 2018) investigate the usability (e.g., compilability, executability) challenges of the code snippets posted on Q&A sites. For example, Yang et al.

Table 1
Challenges preventing issue reproduction.

(C_1) Class/Interface/Method not found
(C_2) Important part of code missing
(C_3) External library not found
(C_4) Identifier/Object type not found
(C_5) Too short code snippet
(C_6) Database/File/UI dependency
(C_7) Outdated code

(2016) analyzes the parsability and compilability of code snippets extracted from the accepted answers of SO. However, their fully automatic analysis only exposed parse and compile errors (e.g., syntax errors, incompatible types). Horton and Parnin (2018) examine the executability of the Python code snippets found on the GitHub Gist system. They identify several flaws (e.g., syntax errors, indentation errors) that prevent the executability of such code snippets. However, a simple execution success does not always guarantee the reproducibility of issues (Mondal et al., 2019). Several studies investigate the quality of SO code snippets by measuring their readability (Buse and Weimer, 2008, 2009; Daka et al., 2015; Posnett et al., 2011; Scalabrino et al., 2016; Treude and Robillard, 2017), and understandability (Lin and Wu, 2008; Scalabrino et al., 2017; Trockman et al., 2018). Unfortunately, their capability of reproducing the issues reported in SO questions was not investigated. Mondal et al. (2019) first investigate the reproducibility of issues reported on 400 SO questions related to Java programming language. Their investigation produces a catalog of challenges (Table 1) that might prevent reproducibility. However, the catalog was not validated by the practitioners (i.e., SO users). Thus, it is unknown how practitioners perceive these reproducibility challenges. The practitioners' perspective is essential for (a) estimating the validity and severity of these challenges and (b) introducing effective tool supports to promote reproducibility.

In this study, we first survey 53 users of SO to understand their perspective on reproducibility challenges (Table 1). This survey investigates three perspectives – (a) agreement (agree/disagree) with the reproducibility challenges, (b) estimation of the potential impact of these challenges, and (c) determination and prioritization of tool support needs. Second, we develop five Machine Learning (ML) models to predict issue reproducibility. Nine code-based features (e.g., line of code, compilability) are extracted to develop these models. Their strength and generalizability are also verified. Third, we systematically interpret the ML model to explain how the code snippets with reproducible issues differ from those with irreproducible issues. Such interpretation might help users improve their code snippets to support reproducibility.

Contribution. This paper significantly extended several aspects of our previous study (Mondal and Roy, 2022). *First,* we attempt to see how well empirical evidence supports practitioners' perspectives on the impact of reproducibility challenges in answering questions. *Second,* we extract nine code-related features to estimate code snippets' capability of reproducing the issues reported in the questions. *Third,* we developed five ML models using these features to classify reproducible and irreproducible issues. *Fourth,* we systematically interpret the results of the ML model to distinguish between code snippets with reproducible and irreproducible issues. *Fifth,* we prepare a dataset by manually investigating the reproducibility of issues reported in 100 C# related questions. We use this dataset to examine the effectiveness of our extracted nine features by predicting the reproducibility status of C# code snippets.

Replication Package that contains the survey questionnaire, all the responses & their analysis, and our prepared C# dataset is shared in our online appendix (Mondal and Roy, 2023).

2. Background and related work

In this section, we start by introducing the *issue reproducibility* and the associated challenges that prevent reproducibility. We then proceed to describe related studies.

2.1. Issue reproducibility

Reproducibility is often closely related to repeatability and replicability (Anda et al., 2008). However, its definition differs across disciplines. In this study, reproducibility refers to a complete agreement between the reported and the investigated issues (White et al., 2015; Mondal et al., 2019). Consider the example question in Fig. 1, where a user was trying to take a person's full name (e.g., John Doe) as input by invoking the `next()` method of Java `Scanner` class. However, when the user printed the name, only the first name (e.g., John) was displayed, and the last name (e.g., Doe) was missed. Specifically, the user did not retrieve the part of the name following the space. She included the definition of the method `processName()`, where she was taking and printing the name. According to our definition, the issue is reproducible when users get only the first part of the name and lose the part after space by invoking the `processName()` method. According to our definition, the issue is reproducible when users invoke the `processName()` method, can retrieve only the first part of the name, and lose the part after space.

2.2. Reproducibility challenges

Mondal et al. (2019) investigated the reproducibility of issues reported in 400 Java-related questions of SO. They could not reproduce about 22% issues due to several unmet challenges of the code snippets, as shown in Table 1. This section briefly discusses the challenges that prevent reproducibility as follows.

(C_1) **Class/Interface/Method not found.** Developers often submit only the code snippets of interest with questions that are neither complete nor compilable. These snippets may lack class definitions or invoke methods from other classes without providing the method definitions. Fig. 2 shows an example question where the developer attempts to reset all the variables to their default values. However, the code did not work as expected, and some variables retained their old values. Unfortunately, this issue could not be reproduced due to missing elements, particularly definitions of the `Game` class and `playGame()` method. These are essential to reproduce the issue. Thus, while attempting to answer the question, one developer commented – *"Can you post more code? The Game class? The class that contains the restart() method?"*.

(C_2) **Important part of code missing.** Code snippets often lack essential code statements, making it challenging to reproduce issues. "An important part" refers to a part of code that cannot be accurately inferred or guessed. For example, adding definitions for the `Game` class and `playGame()` method in Fig. 2 might make the code executable. However, we cannot reproduce the issue. We require the specific code within the class and the method used by the developer to reset the variables.

(C_3) **External library not found.** Resolving external library dependencies is one of the major challenges in reproducing the issues from the submitted code snippets (Mondal et al., 2019). Java has thousands of external (i.e., third-party) libraries with millions of classes and methods. Different libraries contain classes and methods with the same name. Thus, developers face difficulty adding the appropriate libraries if the code snippets miss import statements of the external libraries or question descriptions do not have hints that point to the appropriate libraries. As a result, they could not compile/execute the code and failed to reproduce the question issues. In the code example shown in Fig. 3, the import statement for the library containing the `XMLType` class was missing. Hence, the issue could not be reproduced.

Scanner doesn't see after space

I am <u>writing a program that asks for the person's full name</u> and then takes that input and reverses it (i.e John Doe - Doe, John). I started by trying to just get the input, but <u>it is only getting the first name</u>.

Here is my code:

```java
public static void processName(Scanner scanner) {
    System.out.print("Please enter your full name: ");
    String name = scanner.next();
    System.out.print(name);
}
```

`java` `java.util.scanner`

Fig. 1. An example (Scanner, 2013) question of stack overflow that discusses a programming issue.

Java: Resetting all values in the program

I am working on this program where at the end of the game I ask the user if they want to play again. If they say yes, I need to start a new game. I made a restart() method:

```java
public void restart(){
    Game g = new Game();
    g.playGame();
}
```

However, when I call this method <u>some of the values in my program stay at what they were during the previous game</u>. Is there a game to just clear everything and create a new instance of the game with all the default values?

`java`

Fig. 2. An example (Java, 2009) question of stack overflow whose issue could not be reproduced due to mainly two unmet challenges – (i) class/interface/method not found and (ii) important part of code missing.

Getting Error On Xml resultSet in java

when I try run to execute java code it displays error. which I used to get xml type resultset from oracle using following method in java

```java
poxml = XMLType(rs.getOPAQUE(1));
```

<u>it shows cannot find symbol --- symbol : method getOPAQUE(int)</u>

any one can suggest any other method for retrieving the xml resultset values from oracle

`java`

Fig. 3. An example (Getting, 2010) question of stack overflow whose issue could not be reproduced due to mainly three unmet challenges – (i) external library not found, (ii) identifier/object type not found, and (iii) too short code snippet.

(C_4) **Identifier/Object type not found.** Code snippets often use identifiers/objects without declaring them. Sometimes, developers can infer the types of these identifiers/objects by examining assigned values or method invocations. However, when the types are not easily inferred, it can hinder issue reproducibility. Consider the code example in Fig. 3, where the types of `poxml` and `rs` are unknown and hard to guess from the submitted code.

(C_5) **Too short code snippet.** Developers often submit incomplete, too-short code snippets with their questions, making it difficult to reproduce the reported issues. For example, Fig. 3 shows a question where only *one* line of code (LOC) was provided. Thus, guessing the missing statements and making the code compilable/executable to reproduce the issue is challenging. The "too short code snippet" challenge could overlap with "important parts of code missing". However, even a longer code could miss important code segments.

(C_6) **Database/File/UI dependency.** Several code snippets could not reproduce issues due to their complex interactions with databases, external files, and UI elements. Fig. 4 shows an example question where the developer tried to create and load an XML doc but encountered a null exception when attempting to access the doc multiple times. This issue could not be reproduced due to the external file dependency. The XML document could not be created because the question lacks crucial (e.g., file path, content) associated with the file.

(C_7) **Outdated code.** A few code snippets contain outdated code (e.g., deprecated class/API) that could prevent issue reproducibility. In this study, we consider a code outdated when it contains classes/APIs incompatible with JDK-1.8. Sometimes, finding an equivalent or alternative class/API for the outdated ones can be challenging. For example, in Fig. 5, the `AudioStream` and `AudioPlayer` classes are not used in JDK-1.8, making it challenging for developers to find alternatives and reproduce the issue.

2.3. Related work

Yang et al. (2016) analyze the usability (e.g., parsability, compilability) of about 914K Java code snippets extracted from the accepted answers of SO. They expose several parsability (e.g., syntax error) and compilability (e.g., incompatible types) challenges. The authors employ automated tools like Eclipse JDT and ASTParser to parse and compile code snippets. Their analysis identifies challenges that prevent the code snippets from parsing and compiling. Horton and Parnin (2018) investigate the executability of Python code found on the GitHub Gist system. They report the types of execution failures encountered while running Python gists, such as import, syntax, and indentation errors. However, the success of a code snippet's execution does not always guarantee the reproducibility of an issue. A few challenges of compilability/executability might overlap with the reproducibility challenges. However, reproducibility requires testing and debugging that warrant manual analysis, which was not done by Horton and Parnin (2018). Mondal et al. (2019) go beyond code execution by manually investigating the reproducibility of issues reported in 400 Java questions

get resource is null

I have an @Test method with invocationCount=3. Each time this method is run, there is a call for preparing some doc into another method. All it's working GREAT when the @Test is run for the 1st time. The doc is successfully found and loaded:

```java
final DocumentRegistryResource documentRegistryResource =
    RestClientFactory.getInstance().createDocumentRegistryResource(
        getUserRestAuth());
final File importFile = new File(
    this.getClass().getResource("/documents-template-test.xml").getFile());
```

BUT, at the 2nd and 3rd invocation, I receive a null exception:

this.getClass().getResource("/documents-template-test.xml") is no longer found.

`java`

Fig. 4. An example (Stackoverflow, 2010) question of stack overflow whose issue could not be reproduced due to mainly two unmet challenges – (i) database/file/UI dependency and (ii) class/interface/method not found.

Use Sounds in java

When I create an object from below class and send the address of audio to its constructor, I have to send the complete address like:

"C:\Documents and Settings\TheHero\My Documents\NetBeansProjects\project1\src\project1\s1.wav"

But I want to use this code just with the name of sound like: "s1.wav".

I copied the sound into the src folder and then use "s1.wav" and the result was "java.io.FileNotFoundException". If I use the complete address of sound when I run the program on another computer, it does not work. I'm right?

```java
public class MusicPlayer{
    private String filename;

    public MusicPlayer(String filename) {
        this.filename = filename;
    }
    public void play() {
        try {
            InputStream in = new FileInputStream(filename);
            AudioStream as = new AudioStream(in);
            AudioPlayer.player.start(as);

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

`java`

Fig. 5. An example (Sounds, 2009) question of stack overflow whose issue could not be reproduced due to mainly outdated code challenge.

using the included code snippets. They produce a catalog of challenges (e.g., an important part of code missing) that prevent reproducibility. However, listening to the practitioners to determine whether they agree with these challenges is essential. Furthermore, understanding the potential impact of these challenges in answering questions is also important, which was not addressed by any earlier studies.

Several researchers investigate the challenges of reproducing software bugs and security vulnerabilities (Erfani Joorabchi et al., 2014; Rahman et al., 2020; Mu et al., 2018). Erfani Joorabchi et al. (2014) analyze 1643 non-reproducible bug reports to find causes of their non-reproducibility and reveal six root causes (e.g., environmental differences, insufficient information). Rahman et al. (2020) also investigate 576 non-reproducible bug reports from two popular software systems (e.g., Firefox and Eclipse). They identify 11 challenges (e.g., bug duplication, missing information, ambiguous specifications) that might prevent bug reproducibility. They surveyed 13 developers to understand how they handle non-reproducible bugs. They found that developers either close these bugs or solicit further information. Mu et al. (2018) analyze 368 security vulnerabilities. Their study suggests that individual vulnerability reports often lack sufficient information (e.g., configurations, OS) to reproduce the reported vulnerabilities. Their survey of experts in software security suggested that, besides

internet-scale crowd-sourcing and some interesting heuristics, manual efforts (e.g., debugging) based on experience are the sole way to retrieve missing information from reports. Our study is similar in methodology and problem aspects. However, our research context differs from theirs since we study how practitioners perceive reproducibility challenges of issues reported in SO questions.

Tahaei et al. (2020) analyze 1733 privacy-related SO questions to understand developers' challenges and confusion while dealing with privacy-related topics. Ebert et al. (2019) investigate the reasons (e.g., missing rationale) and impacts (e.g., merge decision is delayed) of confusion in code reviews. Their study shows how developers handle confusion in code reviews, such as asking for more information, improving their understanding of existing code, and discussing issues offline. They surveyed developers to gain insights for researchers and tool developers. While our study uses a similar method, our research goals are different. We surveyed 53 developers to understand how reproducibility challenges impact SO answers and identify tool requirements to address these challenges to promote reproducibility.

Ford et al. (2018) implement a month-long, just-in-time mentorship program for SO. Mentors guide the novices with informative feedback on their questions, aiming to reduce delays in getting answers or adverse feedback. However, human mentorship is costly and challenging

to sustain. Horton and Parnin (2019) develop DockerizeMe, a system that infers dependencies necessary to execute Python code snippets without import errors. Terragni et al. (2016) propose CSnippEx, which automatically converts Java code snippets into compilable Java source code files. We seek the practitioners' recommendations on the design requirement to introduce an automatic tool to support reproducibility. As an initial step, we extract nine code-related features and develop five ML models to predict the reproducibility (reproducible/irreproducible) of issues.

3. Study design

We divided the study into two phases: (1) we survey practitioners for their perspectives on reproducibility challenges, and (2) we predict the reproducibility status using ML models.

3.1. Practitioners' perspectives on reproducibility challenges

In this phase, we survey 53 practitioners (i.e., SO users) to understand their perspectives on reproducibility challenges and estimate the impacts of these challenges on answering questions. We also seek practitioners' suggestions on the tool's design to support reproducibility. In this phase, we answer five research questions.

RQ₁:What challenges do practitioners perceive as contributing to the lack of reproducibility of issues reported on Stack Overflow questions? Mondal et al. (2019) produce a reproducibility challenge catalog (Table 1). Nonetheless, developers' feedback on such empirical findings is essential to increase confidence in these findings (Rahman et al., 2020). To answer RQ₁, we present four example SO questions (issue descriptions + code snippets). Participants were asked to reproduce the reported issues using the provided code snippets. Upon failure, we ask for their agreement with the listed reproducibility challenges by Mondal et al. (2019). Fifty-three software developers validated these challenges, with an agreement level ranging from 80% to 94%. Participants also report three additional challenges they encountered.

RQ₂: What are the perceived impacts of the reproducibility challenges in answering a Stack Overflow question? Understanding the impact of each challenge is important to determine which ones must be addressed to receive appropriate solutions. To answer RQ₂, we present each challenge with five options – (1) *not a problem*, (2) *moderate*, (3) *severe*, (4) *blocker*, and (5) *no opinion*. We then ask participants to select one of these options. Most participants assessed "an important part of code missing" as a blocker and "outdated code" as not a problem.

RQ₃: Do empirical findings support the practitioners' perspective on the impacts of the reproducibility challenges? Examining empirical findings against practitioners' perspectives on reproducibility challenges ensures practical and impactful insights. We categorize the questions with irreproducible issues based on their specific challenges. Then, we determine the correlation between question category and question meta-data. Our analysis shows that *important part of code missing, too short code snippet,* and *external library not found* prevent questions from getting answers, including acceptable ones. This evidence aligns with practitioners' perspectives on the impacts of reproducibility challenges.

RQ₄: How do developers prioritize to address the reproducibility challenges? Prioritizing challenges is essential to identify those that need to be fixed within tight constraints (e.g., time or budget). Participants were asked to prioritize three reproducibility challenges. The top three priorities identified are – (i) an important part of code missing, (ii) class/interface/method not found, and (iii) too short code snippet.

RQ₅: What are the interactive tool design requirements to address the reproducibility challenges? Questions with reproducible issues have a significantly higher chance of receiving timely answers (Mondal et al., 2019). However, SO's current question submission system cannot address reproducibility issues. Aiming to introduce tool support, we ask practitioners for tool design requirements to ensure practical solutions to promote reproducibility. They suggest introducing browser/IDE plugins to assist question submitters in improving their code snippets to support reproducibility. They also recommended tools to identify (1) missing important parts of code and (2) reproducibility challenges according to their severity by statically analyzing the code snippets.

3.2. Prediction of reproducibility status

In this phase, we develop ML models by extracting code-based features to predict whether code snippets included in questions can reproduce the reported issue. We answer the following research question.

RQ₆: Can we predict the reproducibility status of the issues reported in Stack Overflow questions? We extract nine code-based features by analyzing two types of code snippets – *can reproduce* & *cannot reproduce* question issues. We developed five supervised ML models using these features to classify reproducible issues from irreproducible ones. Our models can predict issue reproducibility with 84.5% precision, 83.0% recall, 82.8% F1-score, and 82.8% overall accuracy. To validate our extracted features, we tested two machine learning models to predict issue reproducibility reported in 100 C# related questions and found promising results.

4. Practitioners' perspectives on reproducibility challenges (RQ₁–RQ₅)

In this section, we first survey practitioners to understand *"what they say"* (**RQ₁**, **RQ₂**, and **RQ₃**) about reproducibility challenges and the impact of these challenges to answer questions. We then investigate whether empirical evidence supports practitioners' opinions (**RQ₄**). Next, we see the code modification plan to understand *"what developers do"* (**RQ₄**) to reproduce the issues. Then, we find *"what practitioners suggest"* (**RQ₅**) to get recommendations for designing interactive tools to promote reproducibility. We utilize appropriate tools to analyze the survey responses and answer our research questions.

4.1. Survey

We conduct an online survey to validate and expand the catalog of reproducibility challenges (see Table 1) identified by Mondal et al. (2019). Furthermore, we seek participants' viewpoints on the impacts of these challenges and their potential fixes. We follow Kitchenham and Pfleeger's guidelines for personal opinion surveys (Kitchenham and Pfleeger, 2008). However, we also consider the guidance and ethical issues from the established best practices (Groves et al., 2009; Singer and Vinson, 2002).

Survey Design. Our survey includes different questions (e.g., multiple-choice and free-text answers). Before asking questions, we explain the survey's purpose and our research goals. We assure the participants that their information must be treated confidentially. First, we piloted the survey with three practitioners to get feedback on its length and clarity. Based on their input, we made minor adjustments and finalized the survey. We informed participants that the survey would take 45-50 min based on the pilot survey. Responses from the pilot survey are excluded from the results presented in this paper. Our survey comprises five parts as follows.

(i) *Consent and Prerequisite.* In this part, we ask the participants to confirm their consent to participate in this survey and agree to process their data. We set two constraints for participating in this survey – (1) participants must answer SO questions and (2) have experience in Object-Oriented Programming (OOP), preferably Java.

(ii) *Participants Information.* This part collects participants' OOP and professional software development experience, professional background, SO account age, and question-answering experience.

(iii) *Agreement to the Reproducibility Challenges.* We present four SO questions (see Figs. 2, 3, 4 & 5) from Mondal et al. (2019) published dataset. We aim to select the fewest questions that cover all reproducibility challenges and minimize common challenges among questions. Our selected questions share only one common challenge (i.e., Class/Interface/Method not found) between Q1 (Fig. 2) & Q3 (Fig. 4). Our primary focus is to validate the challenge catalog. We thus ask participants to reproduce the question issues using the provided code snippets. Upon selection, "I cannot reproduce", we ask for their agreement with the provided challenges that could prevent reproducibility. However, they have the option to report additional challenges. We ask the following three questions and analyze participants' agreement with Mondal et al. (2019) findings on reproducibility status and challenges.

(Q$_1$) Do the participants reproduce the reported issues? (*I can reproduce the issue/I cannot reproduce the issue*)

(Q$_2$) Do the participants agree with the reproducibility challenges? (*Yes/No*)

(Q$_3$) Do the participants find additional challenges reproducing the issues? (*Text*)

On the contrary, upon selection, "I can reproduce", we ask for their code to see the editing actions needed to reproduce the issues.

(iv) *Impact and severity of the Reproducibility Challenges.* Here, each challenge is presented with five options: (1) *not a problem*: no difficulties in answering; (2) *moderate*: irritating but answerable appropriately; (3) *severe*: time-consuming but answerable; (4) *blocker*: unable to reproduce the issue and thus unable to answer; (5) *no opinion*: no specific view. In particular, we ask the question to the participants as follows.

What are the impacts of the reproducibility challenges in answering a question? (*not a problem/moderate/severe/blocker/no opinion*)

Participants were asked to prioritize the three most severe reproducibility challenges by answering the following question.

Which three challenges the participants would like to prioritize above others? (*first choice/second choice/third choice*)

(v) *Tool Support Needs.* Finally, we gather participants' opinions on an intelligent tool's needs and design requirements to promote reproducibility. We also offer six tool support options (Table 7) and employ a 5-point Likert Scale (Joshi et al., 2015; Vagias, 2006) to estimate the participants' consent. In particular, we ask two questions as follows.

(Q$_1$) What kind of tool support do the participants need to assist with the reproducibility challenges? (*Text*)

(Q$_2$) How do the participants agree with our tool support options? (*see options from Table 7*)

Recruitment of Survey Participants. We recruit participants who provide their consent through the following two ways.

(i) *Snowball Approach:* We reach out to participants from companies worldwide through personal contacts. Using a snowballing method (Bi et al., 2022), we encourage them to invite colleagues with similar experiences to join our survey. Out of 48 participants who consented, 44 were allowed to complete the survey. The remaining four did not meet our participation constraints.

(ii) *Open Circular:* We find participants by posting our study description and goals in specialized Facebook groups where software developers discuss programming issues. We also use LinkedIn, a large professional network, to reach potential participants. Through this open circular, we recruited 15 participants who met our criteria and finally received nine valid responses.

Fig. 6 summarizes the Java experience and professions. Out of the 53 participants, 34 (about 64%) had five years or less of Java experience (see Fig. 6(a)). Less experienced developers tend to be more active on SO. Most participants were software developers (45.3%) or academic practitioners (37.7%) (e.g., faculty members, students, and postdoctoral researchers) (see Fig. 6(b)). However, four technical leads and five research engineers also participated. Other details of the

participants can be found in our online appendix (Mondal and Roy, 2023).

Survey Data Analysis. We received 53 valid survey responses and analyzed them using appropriate tools and techniques. For multiple-choice questions, we calculated the percentage for each option chosen. Likert-scale ratings were used to estimate agreement levels. Reproducibility challenges were ranked using the Borda count method (Yamashita and Moonen, 2013). We also manually investigate participants' comments on the impacts of these challenges and their tool design requirements.

4.2. Agreement analysis to the catalog of reproducibility challenges (RQ1)

Table 2 shows the agreement summary. For question 1 (Fig. 2), 94.3% of the participants could not reproduce the issue, while 5.7% reported successful reproduction. Two reproducibility challenges were identified for this code example: *Class/Interface/Method not found* and *Important part of code missing*. These challenges were encountered by 92% of participants, with 8% disagreeing. A similar pattern of agreement and disagreement regarding reproducibility status and associated challenges were observed for questions 2 (Fig. 3) and 3 (Fig. 4). For question 4 (Fig. 5), about 26% of developers were able to reproduce the issue, where the challenge was *outdated code*. The code snippet included classes (e.g., AudioStream) that are not compatible with JDK-1.8. Some developers could reproduce the issue using previous JDK versions or alternative classes. Participants were asked to submit modified code snippets. Some developers used alternative classes, and a few submitted codes were different from the code provided in the original question.

We then analyze the agreement based on the participants' Java experience and profession. Table 3 summarizes the results. Participants with more experience (over five years) agree 8% more on average with the reproducibility status than those with less experience (five years or less). Conversely, more experienced participants agree 7% less with the reproducibility challenges. Experienced participants might have the skills to reproduce issues, while less experienced participants are more eager to resolve challenges (e.g., finding external libraries and fixing database dependencies). Profession-wise analysis shows that technical leads and research engineers agree less with reproducibility status and challenges. They might not be actively programming and have a hectic task schedule, leading to guesses about the reproducibility status and challenges. They mainly disagree on the reproducibility status and challenge of question 4 (see Fig. 5), where the issue was "outdated code".

In addition to the given challenges, participants report three additional challenges they encountered.

(i) *Error log/stack trace missing.* Error logs or stack traces provide meaningful insights into program failures. Some issues cannot be reproduced without them.

(ii) *System dependency or environment setup is missing.* Some programming issues are tied to specific Operating Systems (e.g., Windows), IDEs (e.g., Eclipse), and software versions (e.g., Python 3). These issues cannot be reproduced when questions miss those details.

(iii) *Sample input–output missing.* Several issues require sample input–output to be reproduced.

Table 4 presents all reproducibility challenges, combining those from Mondal et al. with additional challenges from the survey.

Summary. On average, about 88% of participants cannot reproduce the issues using the code snippets, and 89% encounter the given challenges. Only about 11% claim they could reproduce the issues and thus disagree with the challenges. Our analysis by experience and profession shows that the lowest agreement level is 70%, which is acceptable. Such agreements support and validate the reproducibility status and challenge catalog (Table 1).
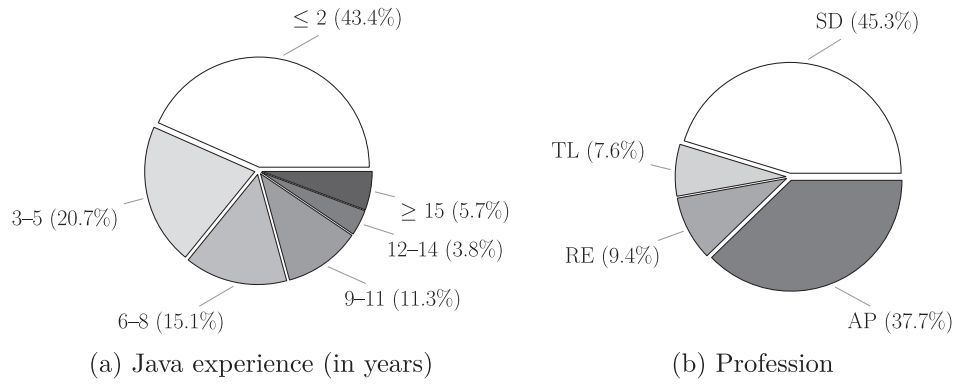
(a) Java experience (in years) (b) Profession

Fig. 6. Information of the survey participants(**SD:** Software Developer, **TL:** Technical Lead, **RE:** Research Engineer, **AP:** Academic Practitioner).

Table 2
Developers agreement to the reproducibility status and the challenges that prevent reproducibility.

Question No.	Reproducibility status		Reproducibility challenges	Reproducibility challenges	
	Irreproducible	Reproducible		Agree	Disagree
01 (see Fig. 2)	94.3%	5.7%	• Class/Interface/Method not found	92.0%	8.0%
			• Important part of code missing	92.0%	8.0%
02 (see Fig. 3)	92.5%	7.5%	• External library not found	93.9%	6.1%
			• Identifier/Object type not found	89.8%	10.2%
			• Too short code snippet	93.9%	6.1%
03 (see Fig. 4)	92.5%	7.5%	• Class/Interface/Method not found	81.6%	18.4%
			• Database/File/UI dependency	89.8%	10.2%
04 (see Fig. 5)	73.6%	26.4%	• Outdated code	79.5%	20.5%
Overall	**88.2%**	**11.8%**	–	**89.1%**	**10.9%**

Table 3
Agreement analysis by experience and profession (SD: Software Developer, TL: Technical Lead, RE: Research Engineer, AP: Academic Practitioner).

| Analysis by Java experience					
Experience	Reproducibility status			Reproducibility challenges	
	Irreproducible	Reproducible		Agree	Disagree
≤5 years	91.2%	8.8%		85.1%	14.9%
>5 years	82.9%	17.1%		92.2%	7.8%
Analysis by profession					
Profession	Reproducibility status			Reproducibility challenges	
	Irreproducible	Reproducible		Agree	Disagree
SD	86.5%	13.5%		89.8%	10.2%
AP	98.8%	1.2%		85.6%	14.4%
TL & RE	69.5%	30.6%		75.8%	24.2%
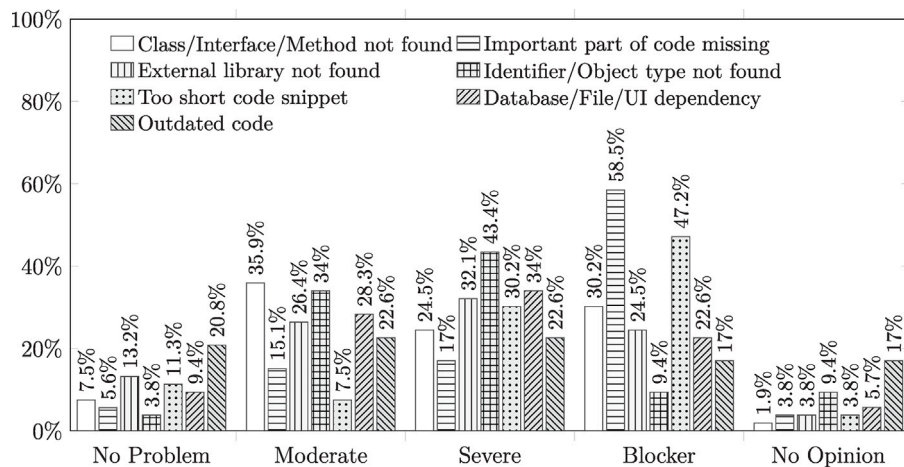
Fig. 7. Impacts on the reproducibility challenges answering questions.

Table 4

Combined reproducibility challenges from Mondal et al. (C_1-C_7) and the survey (C_8-C_{10}).

(C_1) Class/Interface/Method not found
(C_2) Important part of code missing
(C_3) External library not found
(C_4) Identifier/Object type not found
(C_5) Too short code snippet
(C_6) Database/File/UI dependency
(C_7) Outdated code
(C_8) Error log/stack trace missing
(C_9) System dependency or environment setup is missing
(C_{10}) Sample input–output missing

Table 5

Practitioners' comments (unaltered) behind the choices of impacts to answer questions.

"I really think not finding 'Class/Interface/Method' is a blocker problem, because how can I understand a code without their method explained and reproduce. Same case happens in the case of 'Too short code snippet' and 'Important part of code missing'".
"Without Class/Method definition, it's nearly impossible to identify the exact error. Besides, it is helpful, if individuals give some sample input and output examples. Dependency on Database: this is kind of blocker to run the sample code".
"If the given code part is too small, solution can not be made by depending only few line of code".
"As I already answered most of the time we are irritated to help with question without proper playground. (class not found, Important code missing... etc.)".

4.3. Impacts of each of the reproducibility challenges to answer questions (RQ2)

We analyze practitioners' agreement with the reproducibility challenge in Section 4.2. However, understanding how these challenges affect answering questions is crucial for determining which ones need to be addressed early. Fig. 7 shows how participants assess the impact of each reproducibility challenge. The majority (about 36%) perceive "Class/Interface/Method not found" as moderate, with around 30% considering it a blocker. About 59% of participants consider "An important part of code missing" a blocker. "Too short code snippet" is also considered a blocker by around 47% of participants. Regarding "Identifier/Object type not found", approximately 43% perceive it as severe, while 34% perceive it as moderate.

We then attempt to identify the top challenges in each of the five categories. According to Fig. 7, "Class/Interface/Method not found" is the top moderate challenge category. In the severe category, "Identifier/Object type not found" ranks highest, while "Important part of code missing" ranks highest in the blocker category. While "Outdated code" received the highest percentage in the 'no problem' and 'no opinion' categories, participants also perceived this challenge as a moderate (22.6%) or severe (22.6%) problem. Migrating outdated code or utilizing alternative classes/APIs could mitigate this challenge, reducing its impact on developers. We also attempt to gain more insights by asking practitioners for justification for their choices. Even though it was optional, we received a few of their comments (see Table 5). For example, one developer noted that without the Class/Method definition, reproducing the exact issues is nearly impossible.

Summary. Most developers could not reproduce issues and thus could not answer questions if important code statements were missing. Dealing with other challenges, such as Class/Interface/Method and Identifier/Object type not found, could irritate and waste developers' valuable time and prevent/delay appropriate solutions. However, dealing with outdated code is relatively easy.
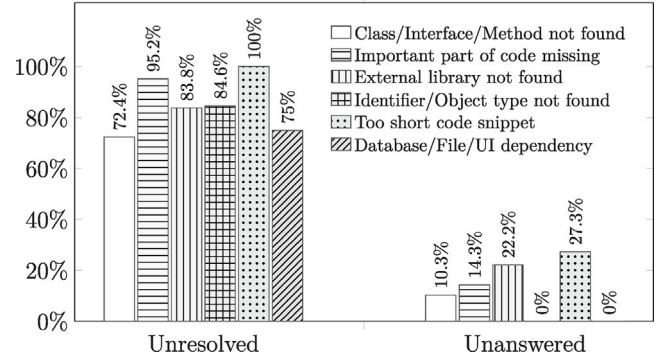

Fig. 8. Percentage of unresolved & unanswered questions vs. reproducibility challenges.

4.4. Agreement between practitioners' opinion & empirical findings on impact analysis (RQ3)

In the previous section, we saw practitioners' opinions on the impact of each reproducibility challenge. This section examines how much empirical evidence supports these opinions.

The dataset published by Mondal et al. (2019) contains 87 questions with irreproducible issues. We first categorize these questions based on their challenges. Then, we collect the question meta-data (e.g., presence of accepted answers, no of answers, and score) and compare this meta-data against question categories to see the impact of these challenges. We skip the question with *outdated code* challenge due to its single occurrence in the dataset.

Fig. 8 shows the percentage of unresolved (questions without acceptable answers) and unanswered questions against each challenge type. All of the questions associated with *too short code snippet* challenge remained unresolved. For *important part of code missing*, 95.2% of questions remained unresolved. These findings align with RQ2 where *important part of code missing* & *too short code snippet* challenges were considered blockers. Additionally, 27.3% of questions were unanswered due to *too short code snippet*, with the second highest unanswered questions attributed to *external library not found* challenge. Interestingly, *External library not found* was also assessed as the second most severe challenge in RQ2.

Fig. 9(a) shows the box plots representing the number of answers against each challenge. The median number of answers is only one for the reproducibility challenges *important part of code missing, too short code snippet*, and *external library not found*. These findings suggest that these challenges significantly affect the chance of getting answers, supporting the practitioners' opinion in RQ2.

We then analyze the quality of questions against each reproducibility challenge. In SO, the quality of a post is subjectively approximated by a metric called *score* (i.e., upvotes − downvotes) (Mondal et al., 2021; Yao et al., 2015; Neshati, 2017; Duijn et al., 2015). Fig. 9(b) shows the box plots of question scores vs. reproducibility challenges. The median score for questions is consistently zero, regardless of the reproducibility challenges. This finding suggests that questions with irreproducible issues struggle to attract user interest for upvotes, as they are often perceived as low quality.

Summary. Empirical evidence confirms the practitioners' opinion on the impacts of reproducibility challenges on answering questions. The analysis between challenge-based question category vs. question meta-data shows that *important part of code missing, too short code snippet*, and *external library not found* significantly hinder questions from receiving acceptable answers or any answers at all.
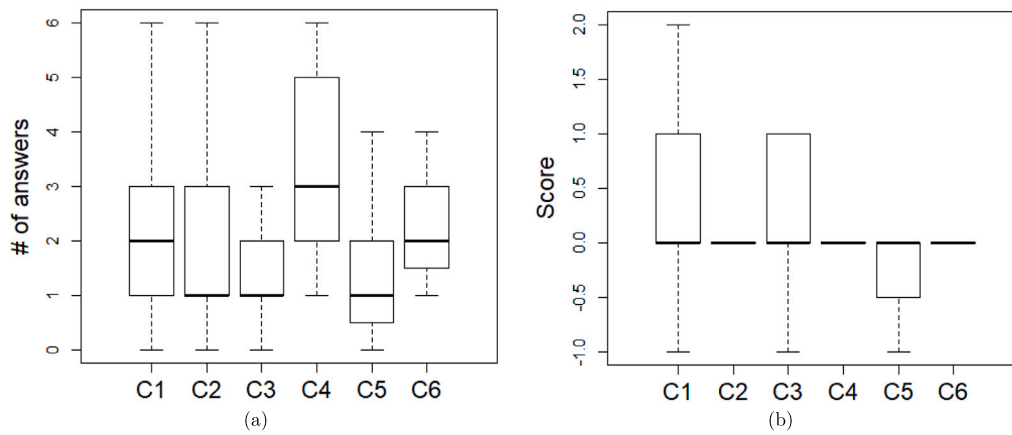
Fig. 9. Answers & score vs. reproducibility challenges (**C1:** Class/Interface/Method not found, **C2:** Important part of code missing, **C3:** External library not found, **C4:** Identifier/Object type not found, **C5:** Too short code snippet, **C6:** Database/File/UI dependency).
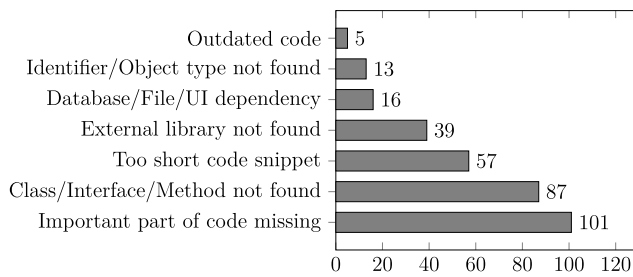

Fig. 10. The rank of reproducibility challenges according to their score.

4.5. Severity analysis of the reproducibility challenges (RQ4)

It is not always possible to fix all the reproducibility challenges. Thus, it is important to prioritize more severe challenges and fix them within a strict budget (e.g., limited time).

We employ the Borda count (Yamashita and Moonen, 2013) to analyze and rank the challenges based on their severity. This method assigns points to candidates based on their rank order: the first-ranked candidate receives the highest points, decreasing sequentially. In our case, with three options to choose from, the first option receives 3 points, the second 2 points, and the third 1 point. Fig. 10 shows the severity points for each challenge. The "important part of code missing" receives the highest (101), while "Outdated code" gets the lowest (5) point. Based on our analysis, the top three priorities are – (1) important part of code missing, (2) class/interface/method not found, and (3) too short code snippet. These findings match those from the impact analysis (see Section 4.3).

Summary. Based on the severity analysis, question submitters should include crucial code parts that cannot be guessed and define necessary classes/methods. They should also avoid submitting code snippets that are too short (e.g., 1–2 lines) to get quick and appropriate solutions.

4.6. Tool support and design requirements to promote reproducibility (RQ5)

To improve question quality by assuring reproducibility for quicker, appropriate answers, we plan to introduce interactive tools (e.g., browser/IDE plugins) and seek practitioners' opinions. Table 6 presents a few valuable recommendations from participants (the complete list is available in our online appendix (Mondal and Roy, 2023)). For example, one participant suggested "A tool that can sense my written code snippet and see what important part might be missing there".

We received 29 responses (out of 53) with practical tool design recommendations. We disregarded the rest as they were either too

Table 6
Practitioners' tool support recommendations (unaltered) to promote reproducibility.

"The tool that helps me to guess the missing code, library version, missing input data depending on the issue context. It will be great if the tool generates missing information for me depending on the question/issue".
"A software that can ensure the missing cases and force the user to complete them, Auto code generator".
"A tool that helps to guess the missing important code depending on the question context".
"A tool that can sense my written code snippet and see what important part might be missing there".
"It will provide suggestions to reproduce after pasting the code in IDE".

Table 7
Assessment of the tool support options.

Options	Mean value
(1) A tool that suggests users improve the code examples to support reproducibility	4.43
(2) A tool that warns users about reproducibility challenges that are severe or may block the reproducibility	4.25
(3) An IDE (e.g., Eclipse) plugin to support reproducibility	4.25
(4) A browser plugin to support reproducibility	4.02
(5) A website that guides users to improve reproducibility	3.85
(6) A static analyzer that examines code snippets to find reproducibility challenges	4.15

generic or unclear. The first author conducted an opening coding to summarize these recommendations as follows.

• *Plugin/Linter/Live code running environment.* About 52% (14 out of 29) of participants suggested adding a browser/IDE plugin, linter, or live coding environment. They specifically requested tools to check the parsability/executability of the code snippets, crucial for reproducing question issues (Mondal and Roy, 2022). These tools can help improve code snippets for better reproducibility.

• *Missing code finder.* About 24% of participants propose implementing a tool that engages with question submitters when code snippets miss an important part that is mandatory to reproduce the issue.

• *Notify if the library inclusion is missed.* Finding external libraries is a major challenge, especially when they are not imported in code snippets. About 21% of survey participants recommend tools that help them include these libraries as needed, which can save time.

• *Environmental setup-related information finder.* About 7% of the participants asked for a tool to analyze the questions' texts to identify missing environmental setup details (e.g., software version, Operating System, IDE), which are crucial for reproducibility.

• *Missing input–output detector.* According to the survey, 6.9% of participants suggest tools for identifying necessary input–output examples to reproduce issues.

• *Auto code generator.* About 7% of participants suggested an auto code generator to generate missing code or search for similar code snippets for reproducibility.

• *Missing definition/declaration finder.* 3.4% of participants suggested a tool to add missing class/method definitions and identifier/object declarations in code snippets, which could promote reproducibility.

• *Remind if the included code snippet is too short.* From the responses, 3.4% of participants want tools to warn question submitters if they include code snippets that are too short. It is hard to identify the problem accurately from such brief code, and guesses often lead to incorrect answers that do not solve the actual problem.

• *Code readability estimator.* A few participants (3.4%) suggest tools to estimate the readability of the code snippets. Poor readability hinders code comprehension and reproducibility.

Additionally, Table 7 lists our six tool options. Participants rated the first three options as "highly influential" (mean value $\geqslant$ 4.21) and the remaining three as "influential" (3.41 $\leqslant$ mean value $\leqslant$ 4.20).

Summary. Practitioners prefer browser/IDE plugins that suggest improving code examples for better reproducibility, such as identifying crucial missing code and external libraries. They also recommend tools that assess reproducibility challenges by severity.

5. Prediction of reproducibility status

To introduce recommended tool supports, we first attempt to develop ML models to predict whether issues are reproducible. The steps for developing, evaluating, interpreting, and validating ML models are discussed below.

5.1. Selection of dataset

We use Mondal et al.'s dataset (Mondal et al., 2019), which consists of 357 questions (270 reproducible + 87 irreproducible), to extract features and develop ML models.

5.2. Feature extraction

We extract nine source code-related features to predict the reproducibility of issues using ML models. We focus on the *executability* and *challenges that prevent reproducibility* to extract these features. The nine features are discussed below.

Line of Code (LOC): We extract code snippets from questions using `<code>` tags under `<pre>`, remove blank lines, and count the lines of code (LOC) to assess their length. Fig. 11 shows that while the median LOC is similar (e.g., $LOC = 19$) for both reproducible and irreproducible issues, the average LOC is higher for irreproducible issues (i.e., $Avg(LOC) = 53.28$) compared to reproducible ones (i.e., $Avg(LOC) = 32.43$). Notably, 13.8% (12 out of 87) of code snippets in questions with irreproducible issues are very short (1–2 LOC), whereas this figure is 5.9% (16 out of 270) for questions with reproducible issues. Thus, LOC can be an important estimator of a code snippet's ability to reproduce reported issues.

Presence of a Method: We check if code snippets include user-defined methods. A method is a block of code designed for specific tasks. Including methods helps reproduce an issue when it is related to that task. In our dataset, about 67% of code snippets from questions with reproducible issues include methods, compared to 55% from questions with irreproducible issues.

Presence of Main Method: The main method can help code snippets be executable and reproducible issues. In our dataset, 55% of code snippets from questions with reproducible issues include the main method, compared to 21% for questions with irreproducible issues.
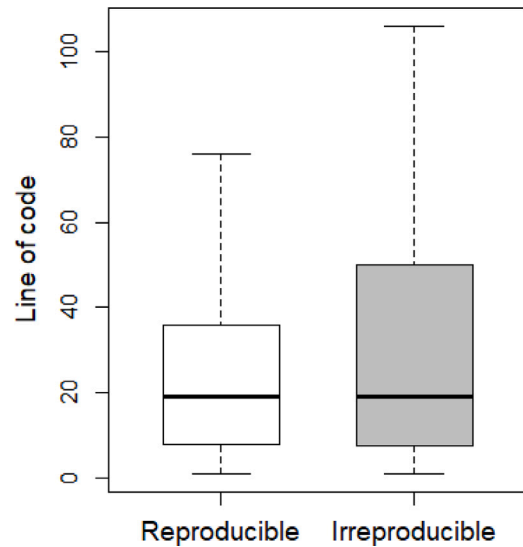

Fig. 11. Line of code of the code snippets included in the questions with reproducible & irreproducible issues.

Presence of a Class: Many Java code snippets are just a few bare statements without being encapsulated correctly in a class, which is essential for parsing or compiling. Yang et al. (2016) improved parse rates by 13.02% and compilation rates by 1.42% by encapsulating snippets with a dummy class when absent. Missing this class is a key challenge affecting reproducibility (Mondal et al., 2022). In our dataset, 54% of code snippets reproducing reported issues included a class, compared to 39% for code snippets with irreproducible issues.

Parsability: We examine how well code snippets can be parsed. Using `Java Development Kit (JDK) 1.8` and `JavaParser`,[1] we can successfully parse 50.7% of code snippets that reproduce reported issues, and 37.9% of code snippets that do not.

Compilability: Successfully executing code snippets is crucial for reproducing reported issues unless the issue is related to a compilation error. We use `javac` to check if code snippets compile successfully. Our analysis shows that 31.1% of code snippets with reproducible issues can be compiled successfully, compared to 1.1% with irreproducible issues.

Inclusion of Native Library: It is crucial to include necessary libraries for compiling and executing code to reproduce issues. IDEs (e.g., Eclipse) often suggest native libraries for common tasks. However, many users copy code into a text editor and then try to execute it from the command line to reproduce issues. Thus, they recommend adding import statements for these libraries. Our feature values are defined as follows.

(a) −1: Required import statements for native libraries were absent.

(b) 0: No native library was needed, or it was unclear from the code snippets.

(c) 1: Necessary import statements for native libraries were present.

Our analysis finds that 51% (82 out of 162) of code snippets with reproducible issues include the required import statements, while only 34% (16 out of 46) of code snippets with irreproducible issues do so.

Inclusion of External Library: We follow an approach similar to *Inclusion of Native Library* to determine feature values for *Inclusion of External Library* as follows.

(a) −1: Required import statements for external libraries were absent.

(b) 0: No external library was needed, or it was unclear from the code snippets.

---

[1] http://javaparser.org.

(c) 1: Necessary import statements for external libraries were present.

Exception Handling: In some cases, unhandled exceptions could prevent issue reproducibility unless the issue is related to those exceptions. We thus check if the code throws any unexpected exceptions and assign feature values as follows:

(a) −1: Exceptions were not handled, but we could handle them.

(b) 0: No exception handling was needed to compile/execute the code snippets, or it was unclear from the code snippets.

(c) 1: Necessary exceptions were handled correctly, or the reported issues were related to unexpected exceptions.

5.3. Resolve class imbalance problem

Our dataset (Section 5.2) has a class imbalance problem, with 270 reproducible samples and only 87 irreproducible samples. Standard ML techniques (e.g., Random Forest) are biased towards the majority class and often misclassify the minority class. To address this, we use the Synthetic Minority Oversampling Technique (SMOTE) (Chawla et al., 2002), which creates synthetic examples for the minority class. We oversample the *irreproducible* class to equalize it with the *reproducible* class.

5.4. Machine learning model selection

Studies suggest that supervised ML classifiers (e.g., Random Forest) can perform equally well or even better than deep learning (DL) techniques (Fu and Menzies, 2017; Beyer et al., 2018) for small datasets. Moreover, DL techniques are more complex, costly, and prone to overfitting with small datasets (Beyer et al., 2018). Therefore, we chose five popular supervised ML classifiers with different learning strategies to classify the issue's reproducible/irreproducible status. These algorithms are selected because (1) they can build reliable models to predict reproducibility status using our features, and (2) they are widely used in relevant studies (Mondal et al., 2023b; Saha et al., 2013; Ponzanelli et al., 2014; Rahman and Roy, 2015; Beyer et al., 2018).

Random Forest (RF) is a popular supervised ML technique due to its simplicity and versatility (Breiman, 2001). It creates an ensemble of decision trees, namely 'forest', trained with the 'bagging' method. The idea behind ensemble learners is that multiple weak learners combine to form a strong learner to improve performance (Polikar, 2006). RF scales well to any number of dimensions and offers acceptable performance. Instead of searching for the most important feature when splitting nodes, it looks for the best feature within a random subset (Mondal et al., 2023b), which helps prevent model overfitting.

eXtreme Gradient Boosting (XGBoost) is a scalable tree-boosting technique (Chen et al., 2015) that predicts a target variable by combining estimates from simpler weak models. It works for classification and regression and offers improved speed and performance compared to traditional gradient-boosted decision trees (GBM) (Mondal et al., 2023b). XGBoost supports parallel and distributed computing, making it faster than other algorithms, and handles different data types and relationships well. It also includes built-in cross-validation and various regularization techniques to prevent overfitting.

Artificial Neural Network (ANN) mimics the brain's neural networks. Unlike perceptrons, which only handle linearly separable data, ANN (Multi-layered Perceptrons) can manage non-linearly separable data. It consists of interconnected neurons arranged in layers. ANN depends on three fundamental aspects: (1) input and activation functions, (2) network architecture, and (3) weights of each input connection. The behavior of an ANN depends on its current connection weights, which start as random values. During training, instances from the training set are repeatedly exposed to the network. The network adjusts its weights slightly to minimize the difference between its output and the desired values.

Table 8
Experimental results.

Technique	Category	Precision	Recall	F1-Score	Accuracy
RF	Reproducible	82.7%	**83.0%**	**82.8%**	**82.8%**
	Irreproducible	**82.9%**	82.6%	**82.8%**	
XGBoost	Reproducible	81.8%	81.5%	81.6%	81.7%
	Irreproducible	81.6%	81.9%	81.7%	
ANN	Reproducible	**84.5%**	66.7%	74.5%	77.2%
	Irreproducible	72.5%	**87.8%**	79.4%	
NB	Reproducible	83.2%	60.7%	70.0%	74.1%
	Irreproducible	68.9%	**87.8%**	77.2%	
KNN	Reproducible	75.3%	70.0%	72.6%	73.5%
	Irreproducible	72.0%	77.0%	74.4%	

Naive Bayes (NB) is a simple yet popular and effective ML classifier. It assumes that input features are independent. It is a probabilistic algorithm based on Bayes' Theorem, which uses the Maximum A Posteriori decision rule in Bayesian settings for classification. NB can also be represented as a basic Bayesian network.

K-Nearest Neighbors(KNN) is a non-parametric technique for classification and regression problems (Goldberger et al., 2005). It does not use the training dataset for generalization, making the training phase faster than other algorithms. In KNN, K determines how many nearby neighbors influence the classification of a data point. It finds the closest K neighbors to a target point using measures like Euclidean distance. Each neighbor votes for its class, and the class with the most votes predicts the outcome.

5.5. Performance analysis

We use default settings from *Scikit-learn* (Pedregosa et al., 2011) for our models and apply 10-fold cross-validation. This method splits the data into ten folds, training on nine and testing on one fold iteratively until all the data are used for training and testing. Cross-validation is one of the most widely used data resampling methods to assess the model's generalization ability and to prevent overfitting (Hastie et al., 2009; Hart et al., 2000; Berrar, 2019). We then evaluate the performance of our prediction models using four performance metrics: (1) *precision* that measures the ratio of correctly classified reproducibility status against all samples classified into that class, (2) *recall* that measures the ratio of correctly classified reproducibility status against the observed samples as true instances, (3) *F1-score* that offers the harmonic mean of precision and recall, and (4) *classification accuracy* that measures the ratio of correctly classified reproducibility status into true & false classes against all samples. These metrics give a clear picture of how well our models perform.

5.6. Model performance to classify reproducibility status

Table 8 summarizes our experimental results. Our models classify reproducible issues with 75.3%–84.5% precision, 60.7%–83.0% recall, and 70.0%–82.8% F1-score. The ANN model achieved the highest precision (84.5%), while the RF model achieved the highest recall (83.0%) and the highest F1-score (82.8%). For irreproducible issues, our models achieved 68.9%–82.9% precision, 77.0%–87.8% recall, and 74.4%–82.8% F1-score. Overall accuracy is above 73%, with the highest being 82.8%. All models showed consistent performance, indicating the strength of our selected features. However, the NB model had comparatively low recall for reproducible issues due to its assumption of feature independence, which is not realistic in our case. Overall, the RF model performed the best.
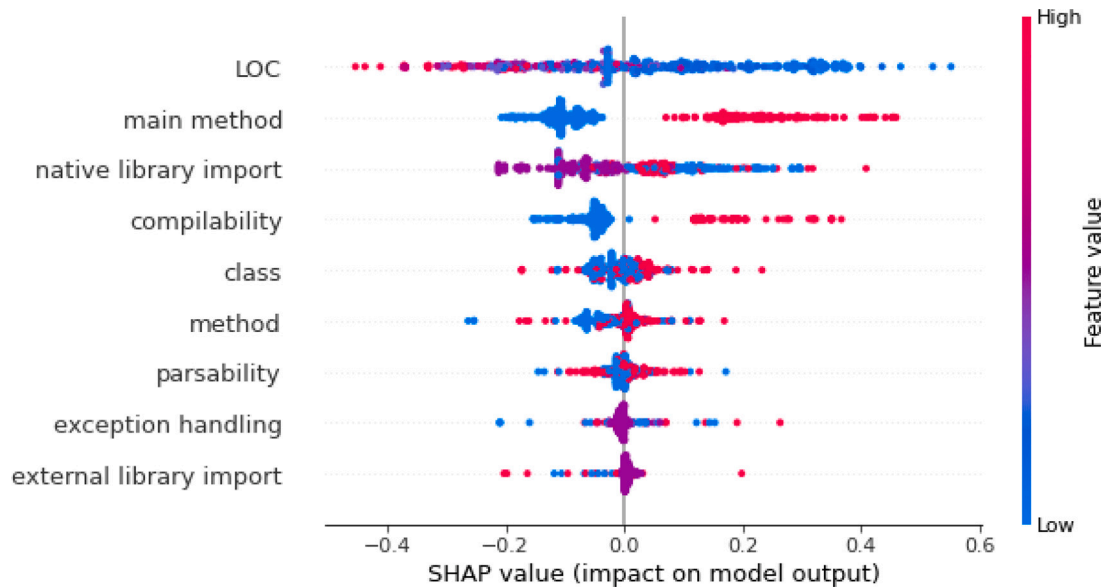
Fig. 12. Feature importance using bee swarm plot (Random Forest model).

5.7. Model performance interpretation

Our classification models separate reproducible issues from irreproducible ones using nine features extracted from the provided code snippets in the questions. This section further explores how these features contributed to the classification. We use SHAP (Lundberg and Lee, 2017), a popular model interpretation framework, to interpret the classification results of our models. The SHAP value is the average marginal contribution of a feature to the model's prediction across all combinations of features (Molnar, 2020; Rahman et al., 2022). It shows whether a feature value increases a model's prediction over a random baseline (Lundberg et al., 2020). In our binary classification, *issue reproducibility* is the *positive* class, and *issue irreproducibility* is the *negative* class. Positive SHAP values indicate an increased prediction of the positive class and vice versa. Figs. 12, 13, 14, 15 & 16 summarize our analysis using SHAP values as follows.

Fig. 12 shows the importance of our selected features using a bee swarm plot from our RF model. The bee swarm plot visualizes the SHAP value of a feature from each of the training instances on the *x*-axis. On the *y*-axis, it sorts all features in descending order based on their total SHAP values. In our dataset, true and false boolean responses are represented by 1 (true) and 0 (false). Blue indicates low feature values, while red indicates high feature values. Thus, small numerical values and false responses are represented in blue, and large numerical values and true responses are represented in red.

According to our RF model, the most important feature is LOC. That means the length of the code snippets in the questions is a key predictor of reproducibility. We observed that a high LOC often leads to negative SHAP values, indicating a higher likelihood of the issue being irreproducible. Our analysis shows that the average LOC for reproducible issues is 32, while it is 53 for irreproducible issues. However, a low LOC can also lead to negative SHAP values. We found that 13.8% of irreproducible issues have code snippets that are too short (LOC = 1–2), compared to only 5.9% for reproducible issues. This suggests that both too-short and too-long code snippets can hinder reproducibility. This observation can be explained in two ways. First, very short code snippets might lack enough detail to make them parsable/compilable, making it hard to reproduce the issue. Second, reproducing the issue from very long code snippets with multiple errors and external dependencies could be tedious.

We also analyze the following three most important features – *main method, native library import,* and *compilability*. Their true responses lead

to positive SHAP values, increasing the model's prediction towards issue reproducibility. In Section 5.2, we see that the main method appears more often in code snippets that reproduce issues. About 55% of these snippets have the main method, compared to only 21% for code snippets that do not reproduce issues. Similarly, code snippets with reproducible issues are 1.5 times more likely to include the required import statements for native libraries. Additionally, 31.1% of snippets with reproducible issues can be compiled successfully, whereas only 1.1% of snippets with irreproducible issues can be compiled. Thus, the presence of the main method, import statements for libraries, or successful compilation might indicate issue reproducibility. Similarly, *presence of method & class*, and *parsability* with their true responses will likely increase our model's prediction towards issue reproducibility.

To verify these findings, we analyze one reproducible issue[2] and one irreproducible issue.[3] The waterfall (Fig. 13) and force plots (Fig. 14) show how different features contribute to classifying an issue as reproducible. In Fig. 13, we see that LOC (8), the presence of import statements for native libraries (*native library import* = 1), the presence of a class (*class* = 1), and parsability success (*parsability* = 1) push the model's prediction towards reproducibility. The presence of a method (*method* = 1) also helps. However, the absence of the main method and compilation failure decrease the model's prediction of reproducibility, as shown by their negative SHAP values.

The force plot in Fig. 13 also illustrates how several features (e.g., LOC, native library import, presence of class, and parsability success) increase our model's prediction towards issues reproducibility, while the absence of the main method and compilation failure decrease it. These findings match our earlier observations (Fig. 12). In Fig. 15, long code (LOC = 59), the absence of the main method and class, and compilation failure prevent the second code snippet[3] from being classified as reproducible. Fig. 16 supports this, though the presence of import statements for native libraries still has a positive SHAP value.

5.8. Statistical analysis of the model features

We use five ML models to classify reproducible from irreproducible issues. We then identify the important features using their SHAP values. This section performs statistical analysis using significance tests to gain
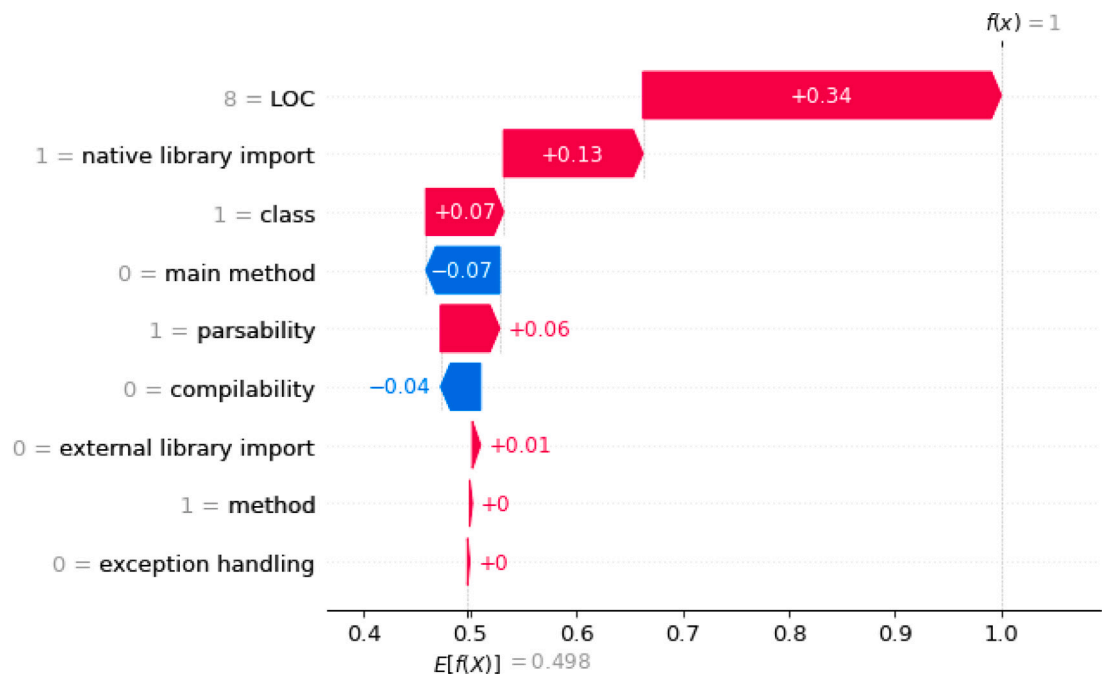
---

2 https://stackoverflow.com/questions/975619.
3 https://stackoverflow.com/questions/790321.

Fig. 13. Features importance using waterfall plot (reproducible issue).

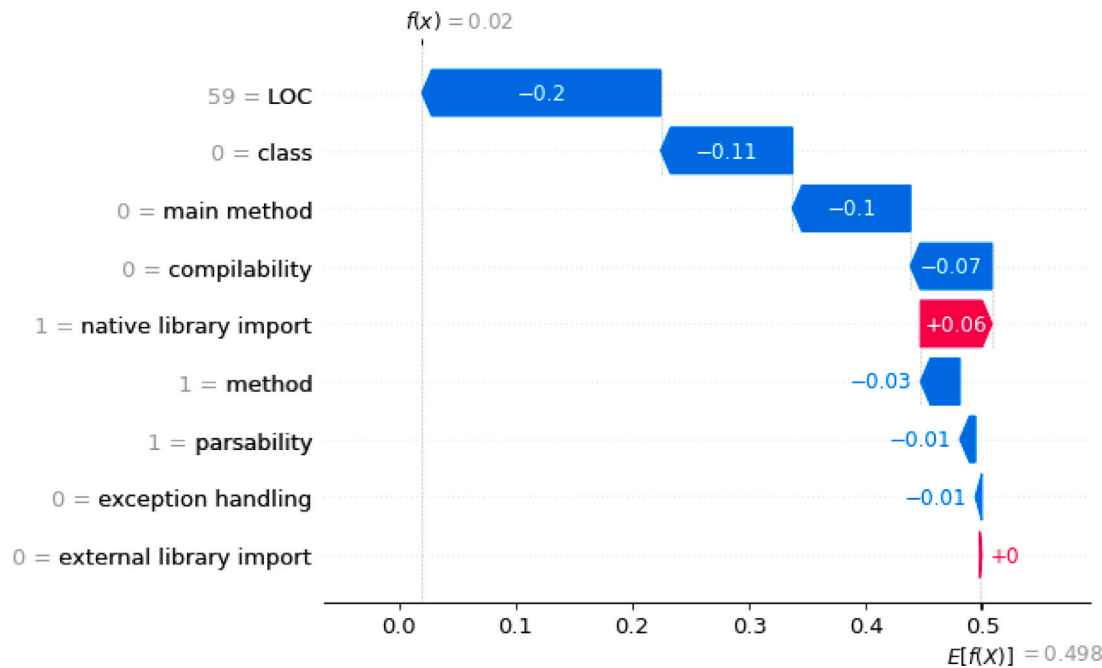**Fig. 14.** Feature importance using force plot (reproducible issue).

Fig. 15. Features importance using waterfall plot (irreproducible issue).

Fig. 16. Feature importance using force plot (irreproducible issue).

Table 9
Impact of model features on issue reproducibility.

Model feature	χ^2	DF	p-value
LOC	6.4	2	$0.04 < 0.05$
Parsability	3.8	1	$0.05 \leq 0.05$
Compilability	30.9	1	$2.67e{-}08 < 0.05$
Presence of method	3.3	1	$0.07 > 0.05$
Presence of main method	28.8	1	$7.95e{-}08 < 0.05$
Presence of class	4.8	1	$0.03 < 0.05$
Native library import	4.7	2	$0.09 > 0.05$
External library import	15.3	2	$4.65e{-}4 < 0.05$
Exception handling	0.7	2	$0.71 > 0.05$

Table 10
Model performances of predicting reproducibility status of C# code snippets.

Technique	Category	Precision	Recall	F1-Score	Accuracy
RF	Reproducible	78.6%	71.0%	74.6%	75.8%
	Irreproducible	73.5%	80.7%	74.6%	
XGBoost	Reproducible	79.3%	74.2%	76.7%	77.4%
	Irreproducible	75.8%	80.7%	78.1%	

further insights into how these features affect issue reproducibility. We have eight categorical features and one numerical feature (*LOC*). We categorize LOC into three categories: short (below 25th percentile), medium (25th–75th percentiles), and long (above 75th percentile). We use χ^2-test (McHugh, 2013) test to analyze the impact of features on issue reproducibility.

Table 9 summarizes our statistical findings. Six of our features (e.g., LOC, comparability) strongly correlate with issue reproducibility, as indicated by SHAP analysis. For example, the ability to compile code significantly affects issue reproducibility (χ^2 value with 1 degree of freedom is 30.9, and *p*-value $\equiv 0 < 0.05$). While SHAP analysis highlighted the importance of *native library import* for model performance, our statistical tests found it insignificantly impacts issue reproducibility. In contrast, *external library import* is significantly associated with issue reproducibility. In practice, native libraries can be imported easily following IDE recommendations, whereas importing external libraries can be more complex and time-consuming. Additionally, features like exception handling and the presence of a method have minimal influence on issue reproducibility in both SHAP and statistical analyses.

5.9. Generalizability of our extracted features

We randomly select 100 questions related to C# to check the effectiveness of our extracted features to predict issue reproducibility in similar languages like Java. We manually investigate the reproducibility of issues reported in the questions and record their status (reproducible/irreproducible). We then extract the features, train and test ML models, and evaluate their performances. These steps are discussed below.

Dataset Preparation. We followed Mondal and Roy (2022) approach in selecting and manually analyzing the dataset. We collected SO questions related to C# posted on or before May 2023 using StackExchange API (StackExchange, 2019). There were 74,890 questions with <c#> tag. We only considered questions with code snippets to reproduce reported issues. Out of 74,890 C# questions, 60,458 (80.7%) included at least one line of code. We extract code snippets from questions using <code> tags under <pre>. However, a question can have multiple code snippets, which we combined (if required) or investigated separately to reproduce the issue.

Questions containing code snippets might not always discuss an issue. Programmers might seek general help or more efficient code, but we target only questions that discuss at least one programming-related issue and have at least one code snippet. We thus look for the keywords (e.g., error, issue, exception, fix) identified by Mondal

and Roy (2022) to find questions that discuss programming-related issues. In this way, we identified a total of 25,364 C# questions. However, Mondal and Roy (2022) found that these keywords could identify target questions with 82.0% precision, 82.4% recall, and 82.3% overall accuracy. We thus randomly selected 140 questions for initial analysis. Among them, we manually analyze 100 questions that genuinely discuss programming-related issues.

Qualitative Analysis. We hired two C# developers to analyze code snippets from 100 C# questions. One developer who took the lead responsibility has five years of experience, and the other has ten years of C# development experience. They follow a two-step process. First, they understand the reported issues clearly and gather supporting data from the question description. Second, they attempt to reproduce the reported issues using the code snippets and supporting data. Finally, they marked issues as reproducible or irreproducible based on their findings. The second developer investigated issues only when the first developer could not reproduce them. Issues were labeled as *irreproducible* when both developers failed to reproduce them. They were able to reproduce 69 issues, while 31 could not be reproduced.

Environment Setup. We use *Microsoft Visual Studio 2022 Community Edition (version 17.3.6)*[4] for executing code snippets and reproducing programming issues. For parsing C# code snippets, we utilize *Roslyn*, a tool provided by Microsoft that offers rich APIs (e.g., CSharpSyntaxTree.ParseText) for parsing the abstract syntax tree from the C# code (Mondal et al., 2023a). We use the built-in compiler *Microsoft C++ Compiler (MSVC)* to compile C# code snippets and MySQL Workbench[5] 8.0 as our example database for this study. We use a desktop computer with a 64-bit Windows 10 Operating System and 8 GB primary memory (i.e., RAM). About 50%–60% of RAM was allocated (by default) for the Visual Studio IDE.

Model Construction and Performance Analysis. We extract the nine code-related features (Section 5.2) from C# code snippets. We select two top-performing models (i.e., RF and XGBoost) to experiment with C# samples. Using a consistent setup (Section 5.5), we apply 10-fold cross-validation and address class imbalance with SMOTE (Chawla et al., 2002). Performance evaluation includes precision, recall, F1-score, and classification accuracy metrics.

Table 10 shows the experimental results. Both models demonstrate promising performance in predicting the reproducibility status of issues in C# questions. Specifically, they achieve approximately 79% precision, 71% – 74.2% recall, 74.6%–76.7% F1-score, and 75.8%–77.4% overall accuracy. These results align closely with our earlier experiments using Java samples (Section 5.6), indicating that the extracted features effectively predict reproducibility status across statically typed languages like Java and C#.

---

[4] https://visualstudio.microsoft.com/vs/community.
[5] https://www.mysql.com/products/workbench.

Table 11
Editing actions to make the code snippets capable of reproducing the question issues.

Question no.	Reproducibility challenges	Action detail	No Action
01 (see Fig. 2)	(1) Class/Interface/Method not found, and (2) Important part of code missing	Addition of demo classes and methods (62%), object creation, identifier declaration and initialization (20%), invocation of methods (4%)	36%
02 (see Fig. 3)	(1) External library not found, (2) Identifier/Object type not found, and (3) Too short code snippet	Addition of demo classes and methods (26.5%), the inclusion of native and external libraries (14.3%), object creation, identifier declaration and initialization (12.2%), other (8.2%)	61.2%
03 (see Fig. 4)	(1) Class/Interface/Method not found, and (2) Database/File/UI dependency	Addition of demo classes and methods (28%), inclusion of native libraries (12%), other (4%)	68%
04 (see Fig. 5)	Outdated code	Addition of demo classes and methods (20.5%), inclusion of native and external libraries (20.5%), suggest new code (10.3%), other (5.1%)	53.8%

6. Discussion

This section discusses how SO users edit code snippets to reproduce issues aiming to provide automatic code completion insight. We also briefly discuss the potential use of LLMs for these tasks.

We analyze participants' efforts in editing code to reproduce question issues. Specifically, we examine their strategies for addressing reproducibility challenges of the code snippets (Figs. 2 through 5). Survey participants (Section 4.1) were asked to submit modified code snippets, which we manually labeled the editing actions. Table 11 summarizes the participants' code editing actions to reproduce the issues. These actions include - (a) adding demo classes and methods, (b) importing necessary native and external libraries, (c) creating objects, (d) declaring and initializing identifiers, and (e) updating outdated code.

The participants' primary objective in these editing actions is to ensure that the code snippets are compilable and executable. For instance, they observed method invocations without corresponding definitions and thus proceeded to add these definitions. The code snippets were modified based on the hints from question descriptions or guessing. However, our analysis reveals that 36%–68% of participants chose not to modify the code snippets, possibly due to perceived insufficiency or difficulty reproducing the issues accurately. For example, one participant expressed, *"I did not try to reproduce it because it needs much guessing and it may differ much from the original code"*.

In recent years, natural language processing and code completion tasks have advanced significantly with the emergence of large language models (LLMs) such as GPT. LLMs can be leveraged to (a) estimate the reproducibility status and (b) identify crucial missing components in code snippets, which are vital for reproducing issues. LLMs can also be utilized to complete code snippets to make them compilable/executable for reproducing issues.

7. Threats to validity

Threats to internal validity relate to experimental errors and biases (Tian et al., 2014). Our key reproducibility challenges were derived from a qualitative study by Mondal et al. (2019), which could be a source of subjective bias. However, the challenges were validated by 53 developers with an agreement level of about 90% on average. Moreover, the difference between agreement and disagreement on reproducibility challenges is quite large (about 77%) and statistically significant.

Threats to external validity relate to the generalizability of our findings (Tian et al., 2014). Our challenges were derived by analyzing questions related to Java programming problems. Thus, the impacts and tool design requirements can apply to statically typed, compiled programming languages such as C++ and C#. Nonetheless, replicating our study using different languages (e.g., dynamically typed languages like Python) may prove fruitful.

Our extracted features may not predict the reproducibility status of programming languages other than Java. To address this, we manually reproduced issues of 100 C# questions, extracted features, and built ML models. The results were promising, confirming our features' generalizability. However, due to their flexibility, these features may not work well for dynamically typed languages like Python.

We only used data from SO that might cause generalizability threats. We could not validate our findings with other technical Q&A forums because their data is not readily available, and their question and code structures are not well-defined for automatic data collection. However, manually collecting a large number of data samples is impractical. Future research using data from other forums could confirm our results.

Our online survey may have a selection bias. We used both a snowball approach and an open circular for participant selection. Responses were collected anonymously to encourage honesty. Participants ranged from novice to experienced (see Fig. 6(a)) and included mainly software developers, along with other related professions (see Fig. 6(b)). This diversity adds validity to our findings, and any individual biases are mitigated by the large sample size of 53 users.

8. Conclusions

Developers post thousands of questions on SO to resolve their code-level problems and include example code snippets to support the problem descriptions. However, these code snippets cannot always reproduce the question issues due to several unmet challenges, preventing prompt and appropriate solutions. A previous study produced a catalog of reproducibility challenges, but practitioners did not validate them. In this study, we attempt to understand practitioners' perspectives on these challenges and (2) develop five ML models to predict the reproducibility status of issues. Our study findings are five-fold. *First,* about 90% of developers agree with challenge catalog. *Second,* "missing an important part of code" most prevents questions from receiving answers. *Third,* a tool to analyze code snippets and identify reproducibility issues could help users improve their questions. *Fourth,* our ML model can predict issue reproducibility with 84.5% precision, 83.0% recall, 82.8% F1-score & 82.8% overall accuracy. *Fifth,* we systematically interpret the results of the ML model.

In the future, we plan to integrate tool support (e.g., browser plugin) with the SO's question submission system to assist question submitters in improving their code snippets to promote reproducibility. LLMs can be leveraged to analyze code snippets and offer immediate feedback and suggestions for improving them and their reproducibility.

CRediT authorship contribution statement

Saikat Mondal: Writing – original draft, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Banani Roy:** Writing – review & editing, Project administration.

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.

Data availability

Data will be made available on request.

Acknowledgments

References

Anda, B.C., Sjøberg, D.I., Mockus, A., 2008. Variability and reproducibility in software engineering: A study of four companies that developed the same system. IEEE Trans. Softw. Eng. 35 (3), 407–429.

Berrar, D., 2019. Cross-validation.

Beyer, S., Macho, C., Pinzger, M., Di Penta, M., 2018. Automatically classifying posts into question categories on stack overflow. In: Proceedings of the 26th Conference on Program Comprehension. pp. 211–221.

Bi, T., Xia, X., Lo, D., Grundy, J., Zimmermann, T., Ford, D., 2022. Accessibility in software practice: A practitioner's perspective. ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (4), 1–26.

Breiman, L., 2001. Random forests. Mach. Learn. 45 (1), 5–32.

Buse, R.P., Weimer, W.R., 2008. A metric for software readability. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis. pp. 121–130.

Buse, R.P., Weimer, W.R., 2009. Learning a metric for code readability. IEEE Trans. Softw. Eng. 36 (4), 546–558.

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P., 2002. SMOTE: synthetic minority over-sampling technique. J. Artif. Intell. Res. 16, 321–357.

Chen, T., He, T., Benesty, M., Khotilovich, V., Tang, Y., Cho, H., Chen, K., et al., 2015. Xgboost: extreme gradient boosting. pp. 1–4, R package version 0.4-2 1.

Daka, E., Campos, J., Fraser, G., Dorn, J., Weimer, W., 2015. Modeling readability to improve unit tests. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 107–118.

Duijn, M., Kucera, A., Bacchelli, A., 2015. Quality questions need quality code: Classifying code fragments on stack overflow. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, pp. 410–413.

Ebert, F., Castor, F., Novielli, N., Serebrenik, A., 2019. Confusion in code reviews: Reasons, impacts, and coping strategies. In: 26th International Conference on Software Analysis, Evolution and Reengineering. SANER, pp. 49–60.

Erfani Joorabchi, M., Mirzaaghaei, M., Mesbah, A., 2014. Works for me! characterizing non-reproducible bug reports. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 62–71.

Ford, D., Lustig, K., Banks, J., Parnin, C., 2018. "We don't do that here" how collaborative editing with mentors improves engagement in social Q&A communities. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems. pp. 1–12.

Fu, W., Menzies, T., 2017. Easy over hard: A case study on deep learning. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 49–60.

Gao, Z., Xia, X., Lo, D., Grundy, J., Li, Y.-F., 2021. Code2Que: A tool for improving question titles from mined code snippets in stack overflow. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1525–1529.

2010. Getting error on xml resultset in java. URL https://stackoverflow.com/questions/2018247, Online; (Accessed 10 January 2020).

Goldberger, J., Hinton, G.E., Roweis, S.T., Salakhutdinov, R.R., 2005. Neighbourhood components analysis. In: Advances in Neural Information Processing Systems. pp. 513–520.

Groves, R.M., Fowler, Jr., F.J., Couper, M.P., Lepkowski, J.M., Singer, E., Tourangeau, R., 2009. Survey Methodology. Vol. 561.

Hart, P.E., Stork, D.G., Duda, R.O., 2000. Pattern Classification. Wiley Hoboken.

Hastie, T., Tibshirani, R., Friedman, J.H., Friedman, J.H., 2009. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Vol. 2.

Horton, E., Parnin, C., 2018. Gistable: Evaluating the executability of python code snippets on github. In: International Conference on Software Maintenance and Evolution. ICSME, pp. 217–227.

Horton, E., Parnin, C., 2019. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In: 41st International Conference on Software Engineering. ICSE, pp. 328–338.

2009. Java: Resetting all values in the program. URL https://stackoverflow.com/questions/798184, Online; (Accessed 10 January 2020).

Joshi, A., Kale, S., Chandel, S., Pal, D.K., 2015. Likert scale: Explored and explained. Brit. J. Appl. Sci. Technol. 7 (4), 396–403.

Kitchenham, B.A., Pfleeger, S.L., 2008. Personal opinion surveys. In: Guide To Advanced Empirical Software Engineering. pp. 63–92.

Lin, J.-C., Wu, K.-C., 2008. Evaluation of software understandability based on fuzzy matrix. In: 2008 IEEE International Conference on Fuzzy Systems (IEEE World Congress on Computational Intelligence). pp. 887–892.

Lundberg, S.M., Erion, G., Chen, H., DeGrave, A., Prutkin, J.M., Nair, B., Katz, R., Himmelfarb, J., Bansal, N., Lee, S.-I., 2020. From local explanations to global understanding with explainable AI for trees. Nat. Mach. Intell. 2 (1), 56–67.

Lundberg, S.M., Lee, S.-I., 2017. A unified approach to interpreting model predictions. Adv. Neural Inf. Process. Syst. 30.

McHugh, M.L., 2013. The chi-square test of independence. Biochemia Med. 23 (2), 143–149.

Molnar, C., 2020. Interpretable Machine Learning: A Guide for Making Black Box Models Explainable. Lulu.com.

Mondal, S., Rahman, M.M., Roy, C.K., 2019. Can issues reported at stack overflow questions be reproduced? an exploratory study. In: 16th International Conference on Mining Software Repositories. MSR, pp. 479–489.

Mondal, S., Rahman, M.M., Roy, C.K., 2023a. Do subjectivity and objectivity always agree? A case study with stack overflow questions. arXiv preprint arXiv:2304.03563.

Mondal, S., Rahman, M.M., Roy, C.K., Schneider, K., 2022. The reproducibility of programming-related issues in stack overflow questions. Empir. Softw. Eng. 27 (3).

Mondal, S., Roy, B., 2022. Reproducibility challenges and their impacts on technical Q&A websites: The practitioners' perspectives. In: 15th Innovations in Software Engineering Conference. pp. 1–11.

Mondal, S., Roy, B., 2023. Replication package. URL https://github.com/saikatmondal/SO-Issue-Reproducibility-Estimation-Replication-Package.

Mondal, S., Uddin, G., Roy, C.K., 2021. Rollback edit inconsistencies in developer forum. In: 18th International Conference on Mining Software Repositories. MSR, pp. 380–391.

Mondal, S., Uddin, G., Roy, C., 2023b. Automatic prediction of rejected edits in stack overflow. Empir. Softw. Eng. 28 (1).

Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., Vendome, C., Poshyvanyk, D., 2016. Automatically discovering, reporting and reproducing android application crashes. In: International Conference on Software Testing, Verification and Validation. ICST, pp. 33–44.

Mu, D., Cuevas, A., Yang, L., Hu, H., Xing, X., Mao, B., Wang, G., 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In: 27th USENIX Security Symposium. USENIX Security 18, pp. 919–936.

Neshati, M., 2017. On early detection of high voted Q&A on stack overflow. Inf. Process. Manage. 53 (4), 780–798.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E., 2011. Scikit-learn: Machine learning in python. J. Mach. Learn. Res. 12, 2825–2830.

Polikar, R., 2006. Ensemble based systems in decision making. IEEE Circuits Syst. Mag. 6 (3), 21–45.

Ponzanelli, L., Mocci, A., Bacchelli, A., Lanza, M., 2014. Understanding and classifying the quality of technical forum questions. In: 2014 14th International Conference on Quality Software. IEEE, pp. 343–352.

Posnett, D., Hindle, A., Devanbu, P., 2011. A simpler model of software readability. In: Proceedings of the 8th Working Conference on Mining Software Repositories. pp. 73–82.

Rahman, M.M., Khomh, F., Castelluccio, M., 2020. Why are some bugs non-reproducible?:–An empirical investigation using data fusion–. In: International Conference on Software Maintenance and Evolution. ICSME, pp. 605–616.

Rahman, M.M., Khomh, F., Castelluccio, M., 2022. Works for me! cannot reproduce–A large scale empirical study of non-reproducible bugs. Empir. Softw. Eng. 27 (5), 111.

Rahman, M.M., Roy, C.K., 2015. An insight into the unresolved questions at stack overflow. In: Proceedings of the 12th Working Conference on Mining Software Repositories. IEEE Press, pp. 426–429.

Saha, R.K., Saha, A.K., Perry, D.E., 2013. Toward understanding the causes of unanswered questions in software information sites: a case study of stack overflow. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, pp. 663–666.

Scalabrino, S., Bavota, G., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D., Oliveto, R., 2017. Automatically assessing code understandability: How far are we? In: 32nd International Conference on Automated Software Engineering. ASE, pp. 417–427.

Scalabrino, S., Linares-Vasquez, M., Poshyvanyk, D., Oliveto, R., 2016. Improving code readability models with textual features. In: 24th International Conference on Program Comprehension. ICPC, pp. 1–10.

2013. Scanner doesn't see after space. URL https://stackoverflow.com/questions/19509647, Online; (Accessed 10 January 2020).

Singer, J., Vinson, N.G., 2002. Ethical issues in empirical studies of software engineering. IEEE Trans. Softw. Eng. 28 (12), 1171–1180.

Soltani, M., Panichella, A., Van Deursen, A., 2017. A guided genetic algorithm for automated crash reproduction. In: 39th International Conference on Software Engineering. ICSE, pp. 209–220.

2009. Use sounds in java?. URL https://stackoverflow.com/questions/1264770, Online; (Accessed 10 January 2020).

2019. StackExchange API. URL http://data.stackexchange.com/stackoverflow.

2010. Get resource is null. URL https://stackoverflow.com/questions/2012643, Online; (Accessed 10 January 2020).

Tahaei, M., Vaniea, K., Saphra, N., 2020. Understanding privacy-related questions on stack overflow. In: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems. pp. 1–14.

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.

Tian, Y., Lo, D., Lawall, J., 2014. Automated construction of a software-specific word similarity database. In: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering. CSMR-WCRE, pp. 44–53.

Treude, C., Barzilay, O., Storey, M.-A., 2011. How do programmers ask and answer questions on the web?(nier track). In: Proceedings of the 33rd International Conference on Software Engineering. pp. 804–807.

Treude, C., Robillard, M.P., 2017. Understanding stack overflow code fragments. In: 2017 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 509–513.

Trockman, A., Cates, K., Mozina, M., Nguyen, T., Kästner, C., Vasilescu, B., 2018. "Automatically assessing code understandability" reanalyzed: combined metrics matter. In: Proceedings of the 15th International Conference on Mining Software Repositories. pp. 314–318.

Vagias, W.M., 2006. Likert-Type Scale Response Anchors. Clemson International Institute for Tourism & Research Development, Department of Parks, Recreation and Tourism Management. Clemson University.

Vincent, N., Johnson, I., Hecht, B., 2018. Examining wikipedia with a broader lens: Quantifying the value of wikipedia's relationships with other large-scale online communities. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems. pp. 1–13.

White, M., Linares-Vásquez, M., Johnson, P., Bernal-Cárdenas, C., Poshyvanyk, D., 2015. Generating reproducible and replayable bug reports from android application crashes. In: 23rd International Conference on Program Comprehension. pp. 48–59.

Yamashita, A., Moonen, L., 2013. Do developers care about code smells? An exploratory survey. In: 20th Working Conference on Reverse Engineering. WCRE, pp. 242–251.

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.

Yao, Y., Tong, H., Xie, T., Akoglu, L., Xu, F., Lu, J., 2015. Detecting high-quality posts in community question answering sites. Inform. Sci. 302, 70–82.

Saikat Mondal is a doctoral researcher in the Department of Computer Science at the University of Saskatchewan, Canada. His research interests include Empirical Software Engineering, Software Repository Mining, Data Analytics, and Explainable AI. Saikat has been investigating several crucial quality aspects of online programming Q&A websites. These websites curate invaluable programming knowledge and disseminate it among millions of software developers daily. He has published over twenty high-quality research papers (e.g., ICSE, ICSME, MSR) and articles (e.g., EMSE). He received several prestigious awards, including the USASK Graduate Thesis Award, Research Excellence Award, GSA Research Excellence in STEM Award, Geddes Award, Carl McCrosky Innovation Award, and Best Paper Award.

Banani Roy is an Assistant Professor and Director of the Interactive Software Engineering and Analytics Lab at the Department of Computer Science, University of Saskatchewan. Her research interests include Engineering Interactive Systems, Software Reengineering, Big Data Analytics, Provenance, and Empirical Software Engineering. She is a co-applicant of the NSERC CREATE program on Software Analytics Research and two CFREF projects. She has been conducting research on human-centric software development & renovation and software analytics. She has worked or has been working with over 20 graduate and undergraduate students with high-quality publications and research awards (e.g., Best Paper Award, Research Excellence Award).