Multi-Language Software Development: Issues, Challenges, and Solutions

Haoran Yang, *Graduate Student Member, IEEE*, Yu Nong, Shaowei Wang, *Member, IEEE*, and Haipeng Cai, *Senior Member, IEEE*

***Abstract*—Developing software projects that incorporate multiple languages has been a prevalent practice for many years. However, the *issues* encountered by developers during the development process, the underlying *challenges* causing these issues, and the *solutions* provided to developers remain unknown. In this paper, our objective is to provide answers to these questions by conducting a study on developer discussions on Stack Overflow (SO). Through a manual analysis of 586 highly relevant posts spanning 14 years, we revealed that multilingual development is a highly and sustainably active topic on SO, with older questions becoming inactive and newer ones getting first asked (and then mostly remaining active for more than one year). From these posts, we observed a diverse array of issues (11 categories), primarily centered around interfacing and data handling across different languages. Our analysis suggests that error/exception handling issues were the most difficult to resolve among those issue categories, while security related issues were most likely to receive an accepted answer. The primary challenge faced by developers was the complexity and diversity inherent in building multilingual code and ensuring interoperability. Additionally, developers often struggled due to a lack of technical expertise on the varied features of different programming languages (e.g., threading and memory management mechanisms). In addition, properly handling message passing across languages constituted a key challenge with using implicit language interfacing. Notably, Stack Overflow emerged as a crucial source of solutions to these challenges, with the majority (73%) of the posts receiving accepted answers, most within a week (36.5% within 24 hours and 25% in the following six days). Based on our analysis results, we have formulated actionable insights and recommendations that can be utilized by researchers and developers in this field.**

***Index Terms*—Multi-language software, Stack Overflow, developer discussion, software development issues, language interfacing, software build, data format, cross-language interoperability, error handling.**

Manuscript received 30 April 2023; revised 13 January 2024; accepted 22 January 2024. Date of publication 24 January 2024; date of current version 15 March 2024. This work was supported in part by the National Science Foundation (NSF) under Grant CCF-2146233 and in part by Office of Naval Research (ONR) under Grant N000142212111. Recommended for acceptance by A. Sarma. *(Corresponding author: Haipeng Cai.)*

Haoran Yang, Yu Nong, and Haipeng Cai are with the School of Electrical Engineering and Computer Science, Washington State University, Pullman 99163 USA (e-mail: haipeng.cai@wsu.edu).

Shaowei Wang is with the Department of Computer Science, University of Manitoba, Winnipeg, R3T 2N2, Canada.

This article has supplementary downloadable material available at https://doi.org/10.1109/TSE.2024.3358258, provided by the authors.

Digital Object Identifier 10.1109/TSE.2024.3358258

I. Introduction

MULTILINGUAL development, which involves the use of multiple languages within a single software project, is a widely recognized software practice [1], [2], [3], [4]. This approach has been commonplace for decades, with over 80% of projects sampled from both the industry and open-source communities utilizing two or more languages [1], [5], [6], [7]. Moreover, the practice has grown in popularity among developers [8], [9], [10]. The benefits of multilingual development are clear, as developers can leverage the strengths of different languages to increase productivity and flexibility [1], [2], [8], [11], [12], [13], [14].

Intuitively, developers who adopt multilingual development (i.e., *multilingual developers*) require techniques and tools to ensure the quality of their software. This need is particularly pressing given the heightened complexity of multilingual code [1]. Research has already revealed that using multiple languages can increase the likelihood of functionality defects [15], [16] and security vulnerabilities [17], [18], [19]. Addressing these concerns necessitates an understanding of the issues faced by multilingual developers and the underlying challenges that contribute to those issues. This knowledge can inform researchers and tool providers about what to prioritize and how best to meet the needs of multilingual developers. Furthermore, identifying current solutions to these challenges can provide valuable insights for designing automated tools and pinpoint areas where further development is needed.

Prior research has extensively examined the phenomenon of multilingual development, mainly focusing on the characteristics of the end product, such as prevalence [1], [3], [5], quality [15], [17], and cross-language links [7], [8], [13], [20]. However, to gain insight into the practical issues and challenges of multilingual development, a common approach is to analyze developers' discussions on relevant Q&A platforms such as Stack Overflow (SO) [21]. SO has enabled numerous informative studies on developers' challenges with software development on a broad range of topics [22], [23], [24]. Nonetheless, no prior research has specifically addressed the multilingual development issues that developers face in practice.

This paper aims to provide a systematic analysis of the *issues*, *challenges*, and *solutions* associated with multilingual development by examining relevant Stack Overflow (SO) posts. We selected SO as our primary source of information due to its significance as a platform where developers exchange

information about software development and as an educational resource that influences their practices [25], [26]. To conduct our analysis, we manually inspected 586 randomly sampled and highly relevant posts[1], covering the entire period of SO's existence until late 2021. For each post, we reviewed the entire discussion thread, including the original question, answers, and comments. Our study revolves around four key questions, and we provide our major findings as respective answers.

RQ1: How prevalent is multilingual development as discussed on SO? (i.e., How prevalent is multilingual development?) We examined the distribution, view counts, number of votes, and lifespan of multilingual development posts on SO from 2008 to 2021 to gain insights into the dynamics of discussions related to multilingual development. This investigation sheds light on the sustaining activeness and relevance of multilingual development discussions.

Starting with the existence of SO, there has been a considerable number of multilingual development questions initiated by developers each year, although this number did not monotonically increase ever since. There were questions becoming inactive (i.e., no follow-up answers or comments) in every year, but those asked earlier maintained their **active status for a non-trivial period** of time (i.e., mostly staying active for over one year). Additionally, they consistently garnered a high number of views and continuously received votes each year. Some of the questions have been being discussed for 10+ years, even up to the entire time span we examined (i.e., 2008 through 2021). Also, the total number of active posts on multilingual development generally follows a normal distribution over time and these posts continue to have a long active time, indicating a sustaining interest and high dynamics in this area. At any given year, there were on average 27% of the studied posts being actively discussed.

RQ2: What are the issues encountered and discussed by multilingual developers? (i.e., What are the issues encountered?) We utilized an open coding method to categorize posts, according to a codebook collaboratively developed and validated by three of the authors. This codebook-derivation process resulted in the creation of our multilingual development issue taxonomy. And the coding process ended up with the 586 posts being categorized by their issue categories.

Our analysis revealed that developers faced a diverse range of issues of 11 categories when developing multi-language software. The primary concerns were related to how to **interface different languages** (accounting for 38% of the 586 inspected posts), **handle data across languages** (30%), and **build the holistic multi-language system** (15%). It was observed that these issues were often associated with specific language combinations. The top three language combinations were found to be `PHP-JavaScript` (26% of the posts), `Python-C++` (10%), and `C++-C#` (9%), while nearly half of all the 586 issues were related to web applications. For example, 72% of embedding issues were mainly encountered in `PHP-JavaScript` projects, while 30% of error/exception handling issues were associated with the language combination of `Python-C++`. Although the majority of questions that received an *accepted* answer got it within a week, some questions never got one. For instance, among all the issue categories, error/exception handling issues appeared to be hardest (with only 50% of related questions having received an accepted answer), while security related issues seemed to be much less difficult to resolve (with 87% of the questions answered satisfactorily).

RQ3: Which challenges do multilingual developers face as the root causes behind the issues? (i.e., What challenges are behind the issues?) From the result of RQ2 (i.e., post categorization by issue types), we identified six most frequent issue categories (i.e., those with top-6 numbers of constituent posts). Subsequently, three of the authors independently analyzed each of these six categories to identify common challenges (i.e., across all of the posts in that category), leveraging the SO discussions and our prior knowledge about multilingual development.

Build issues were mainly about compilation failures, version conflicts arising from language evolution, and project maintenance problems. These problems may be a result of challenges such as insufficient documentation, uninformative compiler error messages, and inadequate tool support. **Data handling** issues mostly involved data conversion and differences in third-party library usage, which stemmed from challenges such as variations in library configurations, complexity in cross-language data structures, and diversity in the type systems of different languages. **Interoperability** issues were frequently observed as failures in memory management and inter-operations across languages, caused by inconsistencies in memory management mechanisms between languages and incompatibilities (and conflicts) in the data types of different languages. **Interfacing choice** issues primarily pertained to selecting appropriate language interfacing mechanisms due to developers' lack of familiarity with the various complex mechanisms. **Explicit interfacing** issues primarily concerned threading and foreign function calls caused by the complexity of multiple threading and varying multithreading requirements. **Implicit interfacing** issues primarily involved message passing and handling requests/responses due to the great complexity of message-passing configurations and the vast variety of ways of handling those requests/responses.

RQ4: How are the challenges being solved by multilingual developers? (i.e., What solutions to the challenges?) We analyzed posts with accepted answers from the six categories studied for RQ3, taking into account their responses and comments. We then distilled these into generalized solutions that address the common challenges outlined in RQ3 for each issue category.

To address the challenge of insufficient documentation underlying **Build** issues, solutions include using external links to relevant information and providing multilingual code examples on SO as alternative documentation. For the data

[1]We consider a post **relevant** if it (1) discusses *multilingual* software, (2) is related to software *development* issues, (3) has been manually confirmed as relevant in both regards (1) and (2)—i.e., related to *multilingual development*, because the automated process of validating (1) and (2) each alone suffers imprecision (false positives).

conversion-related challenges underlying **Data handling** issues, solutions include using a language-independent data format and verifying data conversion (particularly foreign function) calls for encoding/serialization errors. To overcome challenges with memory-management inconsistencies that lead to **Interoperability** issues, current solutions suggest avoiding pointer and memory operations across languages. For the challenge of language unfamiliarity underlying **Interfacing choice** issues, solutions include seeking documentation or alternative references about features of individual languages and/or interfaces across languages. The main solution to the challenge of threading-induced complications underlying **Explicit interfacing** issues is to avoid managing threads across languages and, if necessary, to use a mutex (i.e., a language-specific thread synchronization mechanism) to manage threads. To overcome challenges with complex message-passing configurations that lead to **Implicit interfacing** issues, current solutions suggest ensuring correct data configurations at both the server and client sides and using appropriate approaches to data transfer between them.

Based on our research findings, we propose practical recommendations for researchers and developers engaged in multi-language software development. For instance, we advise multilingual developers to avoid direct management of threads and memory across languages. Instead, they should manage threads and cross-language memory within individual languages and delegate the responsibility of handling low-level interoperations to the respective language's runtime. When deciding which languages to use, developers should be cognizant of the typical issues and challenges associated with multilingual development. This awareness can help them make informed decisions and mitigate potential issues. For researchers, our findings indicate a need for the development of tools that can detect data type/format conflicts across languages, provide better API/usage recommendations for multilingual development, and facilitate building multi-language projects. Our study suggests that the researchers can develop tooling support that recommends optimal options (e.g., concerning interfacing mechanisms and language choices) based on specific requirements (e.g., cross-language type compatibility and overall interoperability) for developers to make best decisions when developing multilingual projects.

This paper extends the preliminary version of our study, as presented in [27]. In comparison, this extended version (1) explores an additional research question (RQ1), (2) elaborates further on our study methodology, especially regarding the LDA-based filtering step in the data collection process, and (3) expands the scope of challenges and solutions pertaining to multilingual development issues. Importantly, in this extension, we have additionally examined challenges underlying a broader range of issues and delved further into the originally examined challenges explored in the initial study. Accordingly, we have also investigated solutions for the newly identified and dissected challenges. Further details on these extensions can be found in Section VI-C. Put together, this extended version represented a notably deeper and more thorough examination of the issues, challenges, and solutions pertinent to multilingual software development as revealed from developers' discussions on SO.

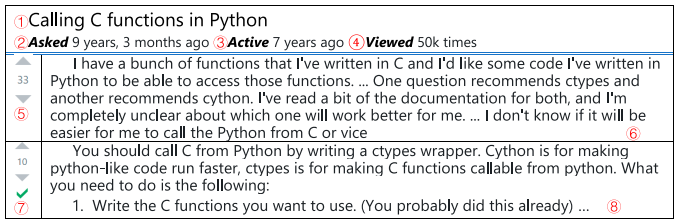

Fig. 1. An example Stack-Overflow post with an accepted answer illustrating that the `ctypes` is recommended to the questioner with advice on the usage of it [28].

II. Background and Motivation

In this section, we define crucial terms and concepts to aid in understanding the remainder of the paper. We also further stress the importance of our study to motivate this work.

A. Multi-Language Software Development

Multi-language software refers to software systems that use more than one computer language, without regard to the language's classification (e.g., programming, modeling, descriptive). Each distinct code unit written in a language, known as a **language unit**, is critical to the system's functionality. The process of developing multi-language software is known as **multilingual development**.

B. Language Interfacing Mechanism

In a multi-language software project, it is important for developers to carefully consider and decide on the mechanisms by which the used languages interface with each other, which is referred to as the **language interfacing mechanism** [17]. For instance, in programs that combine `Java` and `C`, a common approach for interfacing is for the Java code unit to call a `C` function via the Java Native Interface (`JNI`).

In multi-language software development, developers need to decide on the interfacing mechanism for how the different languages interact with each other. There are two types of interfacing mechanisms: explicit and implicit. In an **explicit interfacing mechanism**, the two languages communicate through function invocations. For example, in `Java-C` software, the `Java` code unit invokes a `C` function through Java Native Interface (JNI). There are two types of functions involved in explicit interfacing: *native functions* and *foreign functions*. A **native function** is written by the application developer in a language different from the caller's language. On the other hand, a **foreign function** is written by the language developer in the caller's language and is used to retrieve information from a different language. For instance, as illustrated in Fig. 1, the developer can use `ctypes` to call `C` functions from Python. In this case, the functions written in `C` are foreign functions, and the functions written in Python are native functions.

Alternatively, in the case of **implicit interfacing**, two language units can interact through interprocess communication (IPC) methods such as shared memory, network-based message passing, or message queues. This method allows one unit to access another unit's functionalities indirectly. For instance, in `PHP-Java` applications, the PHP client communicates with the `Java` server through message passing via sockets.

C. Embedding

In multi-language projects, another approach for integrating units of different languages is through embedding one language unit within the code of another language. This technique, called **embedding**, is commonly used between two declarative languages, such as HTML embedding CSS, or between a declarative language and an imperative language, such as HTML embedding JavaScript.

D. Motivation

In recent years, real-world software projects have mostly and increasingly utilized two or more languages, as evidenced by several studies [1], [5], [6], [7], including the most recent one by ourselves [4]. Consequently, there has been a significant increase in the number of reported bugs and issues (e.g., security vulnerabilities) related to multilingual code [15], [17], [18], [19]. Intuitively, those bugs may be a result of *development* issues and challenges developers face when creating and maintaining multilingual software.

Yet currently multilingual code defects have primarily been studied at *code level* only, often through a few case studies of multilingual software projects and defects in them [15], [17], not from a development perspective. As it stands, there remains a lack of understanding of the underlying challenges behind the process of developing the multilingual code. Our study comes in to address this gap: we examine developers' discussions surrounding their experiences and difficulties with multilingual development, which helps gain insights into the development-wise issues and challenges they face that may have resulted in the code-level quality defects of multilingual software, hence informing future design of technique/tool support for facilitating multilingual software quality assurance.

III. Methodology

A standard SO post comprises several properties that we use in our analyses, as depicted in Fig. 1.

① **Title** summarizes the question of the asker (developer).
② **Created time** is the post creation time, in seconds.
③ **Active time** refers to the most recent time at which an individual participated in a discussion on a particular post by posting answers or comments. This time is accurate down to the second.
④ **View count** is the #times the post has been viewed.
⑤ **Vote count** is the sheer #times the post has been liked (i.e., #upvotes - #downvotes).

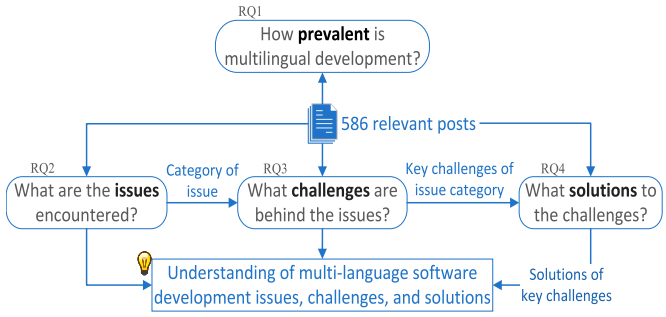

Fig. 2. Our research questions and their relationships.

⑥ **Question** is the description of the problem or issue that the post is addressing. It is typically presented in detail, and may include relevant code snippets, screenshots, or links to external resources.
⑦ **Accepted answer** indicates that the answer that has been chosen by the original poster of the question as the most helpful or satisfactory solution to their problem. The accepted answer is indicated by a green checkmark.
⑧ **Answer** describes a response from a user who attempts to provide a solution or answer to the question raised in the post. It may include code snippets, explanations, or links to external resources.

A. Research Questions

Using the relevant posts, we seek to answer four research questions in relation, as shown in Fig. 2. Below, we clarify the aim and justification for each question with respect to our study goal and describe our approach to answering it.

RQ1: Prevalence. First, we aimed to assess the prevalence of multilingual development. To that end, we intend to investigate the extent and activeness of discussions related to multilingual development on SO. Our approach is to track the yearly total views of SO to gauge any changes in the trend of multilingual development. Recognizing the popularity of multilingual programming can assist developers in determining its significance, and it can also encourage researchers to conduct more studies in this area.

To answer RQ1, we examined the distribution of a sample of highly topic-relevant posts over the span of the 14 years (from 2008 the birth year of SO until 2021) our study covered in terms of the post creation year and the year a post became inactive. This enables insights into the dynamics of SO discussions on multilingual development. To further assess the activeness of those discussions, we characterize the lifespan of each individual post, which informs the duration of attention to respective questions.

RQ2: Issues. Then, we aimed to investigate the issues faced by developers in multilingual development, which can hinder their productivity and affect the software quality. Identifying these problems is the initial step towards finding solutions to overcome these barriers. To achieve this, we developed a taxonomy of multilingual development issues that covers the entire software development life cycle, from analysis and design

to coding, testing, and maintenance/evolution. This taxonomy serves as a comprehensive guide for our further investigation into the challenges and solutions related to multilingual development. We also examined the language combinations associated with each category of issues.

To answer RQ2, we adopted an open coding approach to categorize each post, as we had no prior knowledge about it. Three authors created and validated the codebook with a common inter-agreement and consensus procedure, and then used it to categorize sample posts confirmed as highly relevant to our study, as elaborated in Section III-C. As a result of categorizing these posts, we obtained the issue taxonomy mentioned above.

RQ3: Challenges. Next, we aimed to gain an understanding of the root causes that underlie the issues identified in response to RQ2. This is crucial for developing effective and sustainable solutions to address the identified issues. To achieve this, we focused on the six most prevalent categories of issues and identified the major challenges associated with each of them. The resulting insights provide a valuable reference point for our investigation into the current solutions for addressing each of these categories of issues.

To address RQ3, we categorized the 586 posts obtained from RQ2 based on their identified issue categories. Then, we conducted an independent analysis of the posts in each category to identify the common underlying challenges across those posts. This analysis was performed by three authors and was based on the discussion in the posts, as well as our prior knowledge about multilingual development and relevant online resources. Any disagreements were resolved through meetings until a final consensus was reached for each post.

For each issue category, we initially read and analyzed each post, assigning relevant tags that indicated the content and problems discussed. These tags were then clustered based on semantic similarities, and we examined the posts in each cluster to identify the causal factors of the problems. Finally, we consolidated the primary and common factors for each cluster to identify the root causes or "challenges" for the issue. We identified multiple root causes/challenges for each issue category, although we will only present the primary challenges.

RQ4: Solutions. Finally, our goal was to identify the existing solutions that developers have used to tackle the multilingual development issues they faced. Answering RQ4 would enable us to create a comprehensive catalog of the current solutions, which could serve as a valuable reference for developers. Additionally, it would provide insights into the current practices and state of multilingual development and help us chart a course for future research. We focused on extracting the common solutions for the six most prevalent issue categories we discovered in response to RQ2 and their top underlying challenge identified in response to RQ3.

To answer RQ4, we first selected the posts belonging to the studied six categories which had accepted answers. We then proceeded to analyze the answers and comments provided in each selected post. Finally, we synthesized the specific solutions offered in individual posts into more general solutions that address the root causes or challenges identified in RQ3 for each category of issues.

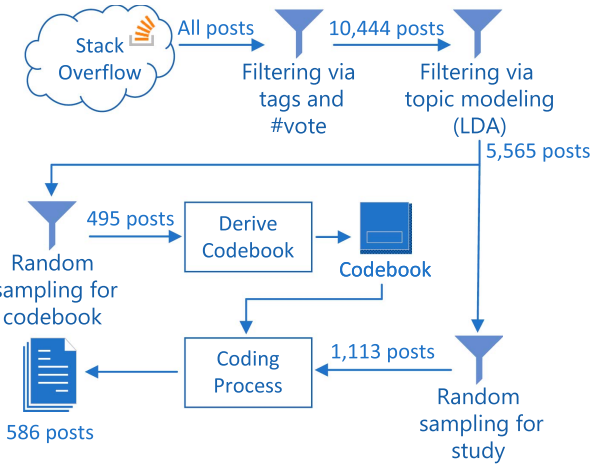

Fig. 3. The flow of our data collection process.

B. Data Collection

The complexity and diversity of each individual language and their combinations in multilingual software make it necessary to rely on manual inspection for content analysis of posts. Therefore, we need to determine an adequate but manageable number of relevant posts to investigate. As depicted in Fig. 3, we began by including all posts available on SO, followed by utilizing *two filters* to identify high-quality relevant posts. It is essential to note that we consider a post **potentially relevant** if it (1) pertains to *multi-language software* (Section II-A) and (2) discusses *software development* issues.

Filtering via language tags and #votes. To ensure manageability and relevancy of our study, we began by selecting the most popular programming languages as tags, which we determined through consulting multiple language popularity rankings [29]. Based on this research, we identified the top-7 popular languages in 2020, which were `JavaScript`, `Python`, `Java`, `C`, `C++`, `Shell`, and `PHP`, as our initial filter. These languages have been widely used for a long time [1], [2], [3]. We selected 7 as our threshold here, because this was the number of languages *commonly* found in the top-10 popularity lists we consulted.

To further narrow down our dataset, we used the number of votes a post received as a proxy for quality, a method previously used in similar studies [30], [31]. We only included posts that received a minimum of 6 votes, as we found this threshold provided a good balance between post coverage and manageable manual effort. Using the `Scrapy` tool [32], we crawled posts from SO that contained at least two of the 7 selected language tags. We chose to crawl the website rather than using a dump because it ensured our study would be up to date. After filtering, we ended up with 10,444 posts that met our relevancy and quality criteria.

Filtering via topic modeling (LDA). Given our study goals, it is essential to focus on posts that are highly relevant to multilingual development issues. However, out of the 10,444 posts, most are clearly irrelevant to any development issue as we have found during our quick sampling and inspection. Nevertheless,

TABLE I
PRECISION AND RECALL OF LDA-BASED POST RELEVANCE IDENTIFICATION WITH DIFFERENT NUMBERS (k) OF KEYWORDS A POST NEEDS TO ALL INCLUDE TO BE DEEMED RELEVANT

k	Precision	Recall
1	45.23%	95%
2	40.74%	55%
3	29.41%	25%
4	22.22%	10%
5	28.57%	10%
6	50.00%	10%

manually filtering out these irrelevant posts would be a tedious and costly process. To efficiently eliminate irrelevant posts, we employed a topic modeling technique called Latent Dirichlet Allocation (LDA) [33] as a filtering step. First, we used LDA to extract a list of generic topic words that are relevant to development issues. Given that not all posts pertain to development issues, the topic list (which results from applying LDA against all those posts) ended up containing many irrelevant words. We then manually removed noisy words (e.g., "hence," "solution") that were not indicative of development issues. Next, we kept a post if it contained k ($\geq$1) of the remaining topic words. Even if a post contained non-indicative or noisy words, we did not exclude the post from our study just because of that. Using the remaining topic words (e.g., "JNI," "SWIG," "socket") as keywords, we experimented with various potential sets of keywords, including different combinations of words as phrases. We randomly sampled 50 posts and manually labeled them as ground truth for validation purposes. For each potential set of keywords, we used it to detect relevant/irrelevant posts from the 50 and calculated the precision and recall based on the ground truth.

As shown in Table I, increasing the number (k) of keywords that a post must all include to be considered relevant results in a trade-off between precision and recall. When k is set to 1, recall is high but precision is low, indicating that many non-relevant posts are incorrectly included. When $k = 6$, precision reaches its maximum value, but recall also reaches its lowest point, indicating that many relevant posts are being excluded. It's important to note that the optimal value of k may vary depending on the specific research question and dataset being analyzed. In this case, a value of $k = 1$ may be sufficient for the current study goals. Thus, we eventually chose $k = 1$ in our LDA analysis.

After experimenting with different keyword sets, we arrived at a final set that provided a recall rate of 95% and precision rate of 45.23%, even though other sets delivered better precision rates but much lower recall rates. Our priority was to keep a high coverage of genuinely relevant posts, despite the higher manual effort required to eliminate false positives in the next stage. The recall rate of 95% is satisfactory for using the LDA-based filter as a criterion, as it ensures that we do not miss many true positives during the final post-relevancy verification. The filtering process utilizing LDA reduced the number of posts to 5,565.

C. Post Categorization

For the next step, we need to derive a codebook and then apply it for the categorization of posts.

Random sampling for codebook. To generate the codebook, we first randomly selected 495 out of the 5,565 posts for analysis. This sample size is statistically significant at 98% confidence level (CL) and 5% margin of error (ME).

Derive codebook. Three of the authors independently created an initial list of issue categories based on the 495 posts, and 262 were identified as highly relevant, followed by addressing any discrepancies to reach a consensus on the final codebook shown in Table II. Specifically, each author carefully read the posts, verified whether each post belonged to the existing issue categories, and created a new category if necessary. When creating a new category, they provided a label to describe the issue in the post, created descriptions for the category, listed some common issues that should belong to the category, and included the post as an example for the category. In certain instances, to enhance the accuracy and utility of the codebook, similar codes were merged and inappropriate ones eliminated. The detailed processes of code extraction, merging, and deletion, along with their rationales, are thoroughly documented in the supplementary document.

Random sampling for study. After filtering with the LDA, we were left with 5,565 posts. Due to the significant time required for manual inspection, we randomly selected 20% of these posts, resulting in 1,113 posts that require manual confirmation in the coding process. This sample size is statistically significant at 99.9% CL and 5% ME.

Coding process. In this final step of our post categorization process, the three authors participated in intensive discussions, working diligently to reach a consensus on the relevance and categorization of each post under review. Specifically, starting with an initial sample of 1,113 posts, the authors embarked on their thorough examination. Every post was individually assessed, taking into account its context, the intent behind it, and its pertinence to the study's goals. After comprehensive discussions and deliberations, the three authors collectively agreed that 586 posts were highly relevant to multilingual development.

After this manual confirmation, the subsequent task was to systematically organize these 586 posts. Using the codebook that had been previously developed, the authors categorized these posts into different categories. This codebook, detailed in Table II, consisted of specific issue categories, each tailored to encapsulate the core essence and concerns of the posts. The three authors distributed the posts across these categories, ensuring each post was appropriately matched to its respective place in the codebook. To ensure the reliability of the coding procedure, we utilized the *negotiated agreement* method, which is particularly useful in research aimed at providing new insights [34]. The three authors who developed the codebook performed the final coding, reaching a consensus on each of the 586 posts. These posts, hereafter referred to as relevant posts[1], were further analyzed manually to answer our four RQs.

TABLE II
KEY CODES USED TO CATEGORIZE SO POSTS ON MULTILINGUAL DEVELOPMENT ISSUES

Code	Summary Description
Language choice	Developers ask/discuss about choosing the languages to use for their multi-language project.
Interfacing	Developers ask/discuss about calling interface (implicit or explicit) between two languages.
Embedding	Developers ask/discuss about embedding one language unit within another language unit.
Build	Developers discuss how to build (e.g., compile, install, configure) the multilingual code.
Efficiency/Performance	Developers are concerned about computing/storage efficiency of their multi-language systems.
Security	Developers have security/privacy/cryptography-related concerns with multilingual development.
Data handling	Developers discuss cross-language data processing and/or transferring data between languages.
Error/exception handling	Developers ask/discuss about handling errors and/or exceptions in multilingual code.

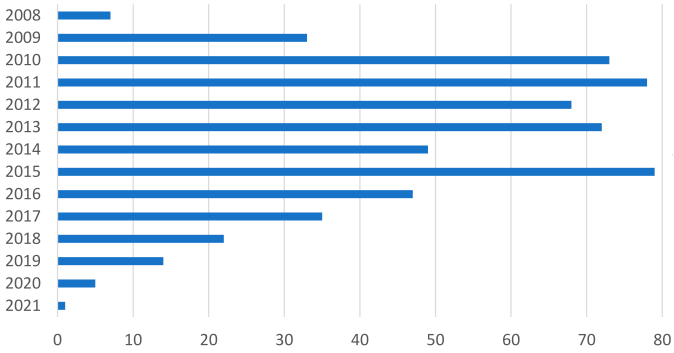

Fig. 4. The number of posts (x-axis) *created* within each of the 14 years studied (y-axis).

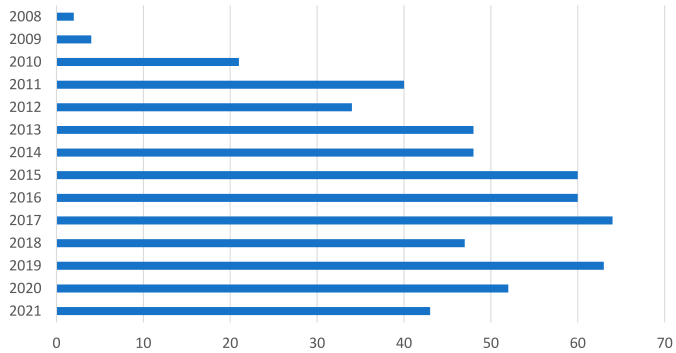

Fig. 5. The number of posts (x-axis) that *became inactive* within each of the 14 years studied (y-axis).

IV. RESULTS

A. RQ1: Prevalence

We started by looking into the dynamics of SO discussions on multilingual development in terms of the number of new posts created each year—that is, new questions that were first asked during that year. Intuitively, as developers who have questions that have been asked earlier by others would be satisfied by just following up the existing posts via new comments or simply viewing the current threads of discussion, they would not need to create new posts for those questions—in fact, SO prevents redundant questions from being asked. As a result, with existing questions accumulating to a (saturation) point, new questions to be asked may not be expected to keep increasing.

This extrapolation is generally validated by our result. As shown in Fig. 4, the number of new questions hit a saturation point at 2015 and it appears that attention to multilingual development has decreased monotonically since 2016. However, we note that two factors need to be kept in mind when explaining this seemingly declining trend. Firstly, during data collection, we established such a threshold that led us to have only collected posts with a vote count greater than five. This measure was taken to ensure the quality of the posts and filter out low-quality content through the voting of the Stack Overflow community. Nevertheless, this threshold means that some newly released posts may not have had sufficient time to accumulate enough votes. The number of views and votes a post receives generally increases as it is active on the site for a longer period. Secondly, the redundancy of questions also affects the trend, as noted above.

To understand the evolution of developers' attention to the topic of multilingual development, we also analyzed the (in)activeness of the posts according to their last active time. Fig. 5 shows how many of the studied posts became inactive (i.e., no followup/answer and no comment anymore) over the years. The result indicates that there were consistently a large number (>40) of posts becoming inactive. Considering that a significant proportion (>80%) of the posts we analyzed were created before 2016 (Fig. 4), it is noteworthy from Fig. 5 that these posts have maintained an active status for a non-trivial period of time, with the majority of them remaining active until 2015-2021. This highlights the enduring relevance of previously asked questions. It is worth noting that the data collection process concluded on August 30, 2021, and thus the data for 2021 is not fully comprehensive. However, this shall not have a significant impact on the overall findings with respect to our goal for RQ1.

To gain deeper insight into developers' interest in multilingual development, we also assessed the total and average number of views for posts created annually. The bottom chart of Fig. 6 depicts the cumulative views of posts created each year within the periods studied. From 2009 to 2013, there's a

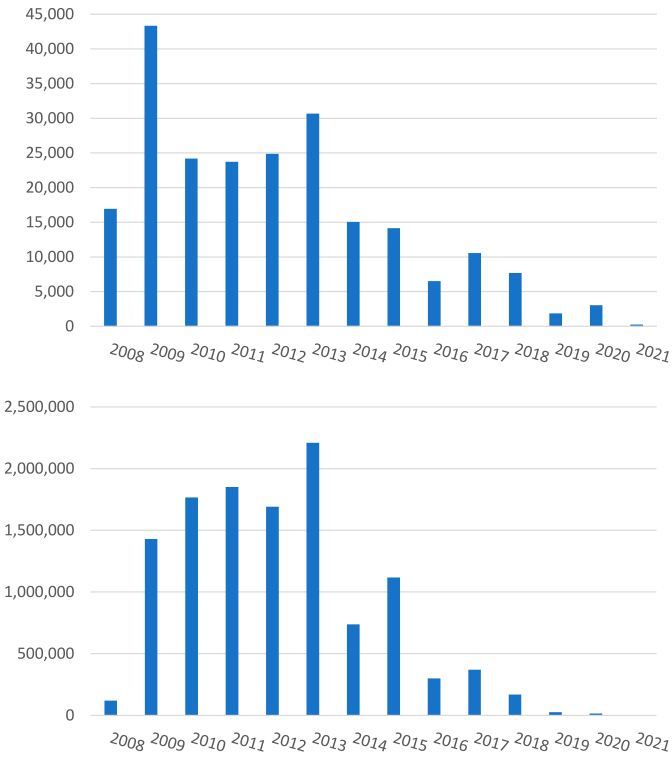

Fig. 6. The distribution of average view counts of each post created in each year (top chart) and total view counts of the posts created in each year (bottom chart).

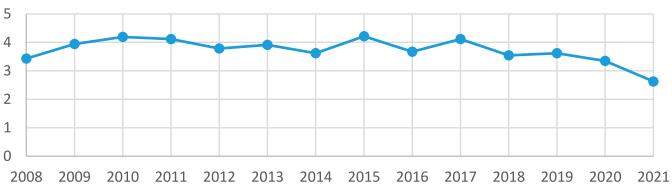

Fig. 7. Average #votes received per post within each year.

notable spike in the total view count. The subsequent years, 2014 and 2015, also maintained a considerable view count. The top chart of Fig. 6 presents the average views per post for each year, highlighting a pronounced peak in views from 2009-2013. The averages for 2014-2015 are also notably higher compared to those from 2016-2021. It's worth noting that SO commenced its operations in 2008, making data from that year less representative due to the platform's infancy and smaller user base.

From Fig. 7, it's evident that posts concerning multilingual development average more than 3 votes annually. It's important to note that the 2021 data only extends to August of that year and wasn't fully considered, as the posts from that period hadn't had ample time to accumulate votes. Such data suggests that multilingual development topics consistently garner significant attention from the community. On SO, the creation of duplicate or highly similar questions is discouraged. Therefore, when a post on a particular question already exists, developers are more inclined to view or participate in the discussions of the pre-existing questions. This behavior is reinforced by Fig. 4,

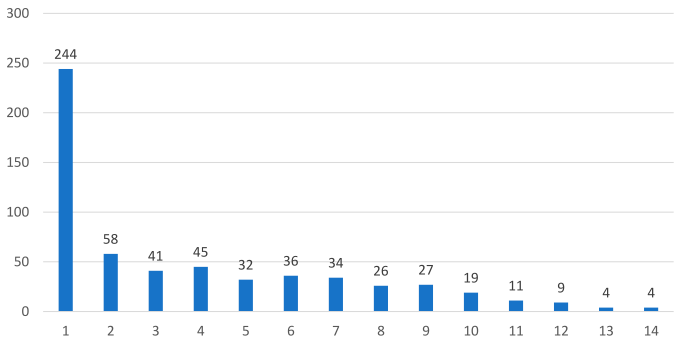

Fig. 8. The count distribution of the 586 analyzed posts (y-axis) over different life spans in #years (x-axis).

which displays a declining trend in the number of new posts created after 2016. Yet, combined with the findings from Fig. 5, which show that the majority of posts remained active through 2015-2021, it is clear that interest in multilingual development persists. This suggests that the topic of multilingual development remains active on SO. Now that developers primarily rely on existing posts for their queries, the platform's discouragement of duplicate content reduces their inclination to create new posts on the same topics. Consequently, this boosts the view counts for existing posts.

While the number of new questions on multilingual development did not always monotonically increase, prior posts have consistently attracted high views and continuously received average votes each year. This highlights the dynamic nature of SO discussions centered on multilingual development.

Given the general observation on the duration of activeness, it is natural to see how long each individual post has exactly been active. The data presented in Fig. 8 indicates that the lifespan of posts related to multilingual development is not short, corroborating the trends observed in Fig. 4 and Fig. 5 combined. In fact, over half of the posts have remained active for more than one year, with approximately 8% of them still being actively discussed after more than ten years. This suggests that the prevalence of multilingual development is not a fleeting trend, but rather a sustaining phenomenon.

Finally, we examined the activeness of multilingual development discussions on SO from another perspective—the total volume of active discussions per year. Fig. 9 depicts the number of posts that were active within each year, The bell shape suggests a normal distribution of these numbers. Using this information, we can deduce that the mean (μ) of the distribution is 156 and the standard deviation (σ) is 82. In other words, on average, 27% of the questions were being actively discussed in any given year, indicating multilingual development topic remained overall active over the years.

The community of developers involved in multilingual development has been kept highly active—relevant questions tend to remain active for a non-trivial period of time since first asked, indicating that multilingual development is sustainably popular.

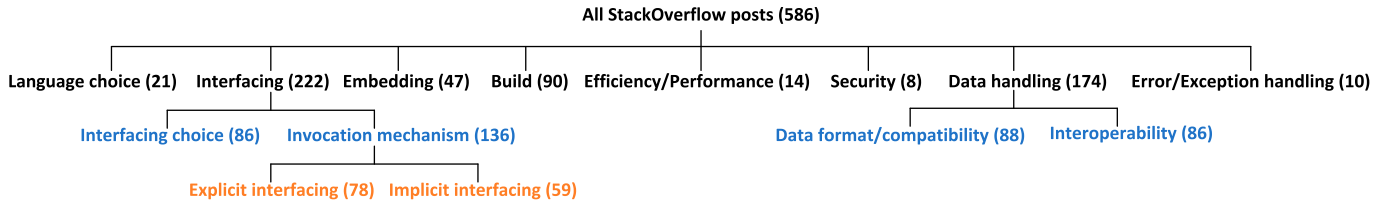

Fig. 9. Our proposed *taxonomy of multilingual development issues* derived on the basis of the studied SO posts.

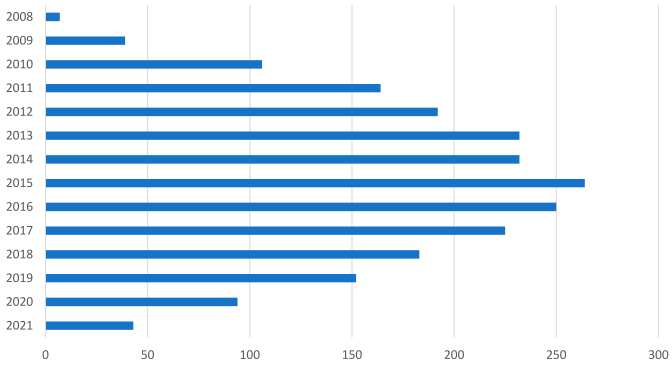

Fig. 10. The number of posts (x-axis) that were active within each of the 14 years (y-axis).

B. RQ2: Issues

We present our issue taxonomy and then examine the correlation between issue categories and language combinations. We also assess the difficulty level of each category of issues through the ratio of answers that were accepted by the question poster (to the total #questions in that category).

1) Issue Taxonomy (Categories): In Fig. 10, we have presented a categorization of the 586 posts. They were categorized into 11 separate and distinct categories, which are located at the leaf nodes of the tree. This hierarchy also provides information on the distribution of posts across the various categories. At the highest level, there are 8 categories derived from the codebook presented in Table II. The two categories with the highest number of posts (222 and 174) were further divided to form the second level: **Interfacing choice** and **Invocation mechanism** under Interfacing, and **Data format/compatibility** and **Interoperability** under Data handling. Additionally, the *Invocation mechanism* category was broken down further into two subcategories corresponding to the two types of interfacing mechanisms (Section II-B) at the third level. We constructed the taxonomy such that the number of posts in all leaf categories does not exceed 100, which makes deeper analyses for RQ3 and RQ4 more feasible.

Approximately 38% and 30% of all relevant posts are categorized under Interfacing and Data handling respectively, indicating that developers require further assistance on these issues.

2) Issue Categories Versus Language Combinations: Fig. 10 (top chart) depicts the primary language combinations linked with each of the 8 top-level issue categories, while the distribution of all 586 posts over the involved language combinations is provided as a reference in the bottom chart. The outcomes in the bottom chart indicate that `PHP-JavaScript` (26% of the posts), `Python-C++` (10%), and `C++-C#` (9%) were the top 3 language combinations used across all issue categories. However, this does not necessarily imply that `PHP-JavaScript` projects are more prone to multilingual development issues. It could just be a result of the higher popularity of this combination among the ones analyzed.

Although the distribution of language combinations can affect the overall percentage of posts associated with each combination, our findings suggest that specific types of issues tend to be associated with certain language combinations. For example, the majority (72%) of Embedding issues were encountered in `PHP-JavaScript` projects, while 50% of the Efficiency/Performance issues were associated with the same language combination. Additionally, 38% of Security issues were related to `Java-JavaScript` projects. On the other hand, Data handling issues were found to be evenly distributed across different language combinations, as evidenced by the height of the bars in the top chart of Fig. 10, where taller bars represent a greater dominance of the top three language combinations associated with a particular category of issues.

The issue categories of Embedding and Efficiency/Performance were mainly associated with a small number of highly-dominant language combinations, whereas other issue categories exhibited a more diverse range of language combinations.

3) Issue Difficulty: Intuitively, examining the answer acceptance status of questions after they are posted can help assess the degree of difficulty of resolving respective issues. Accordingly, we can gauge the level of difficulty for each issue category using the percentage of questions (posts) in that category that had an accepted answer. Similarly, such percentages of posts with respect to other answer status—*undetermined* (i.e., there are answers of which none were marked as accepted) and *undiscussed* (i.e. there is no answer to the question received)—can help further understand the difficulty levels.

Fig. 11 shows that all categories have more than 50% of their posts having received an accepted answer. Out of the 586 relevant posts, a significant portion (73%) of them received an accepted answer, which is higher than the 57% answer acceptance rate reported for all posts on Stack Overflow in an earlier study [35]. Interestingly, only 1% of the multilingual posts never received any answer (i.e., undiscussed). The **Security** category boasts the highest answer acceptance rate at approximately 87%. As depicted in Fig. 12, questions within the 'security' category average 3.88 answers, placing them among the top

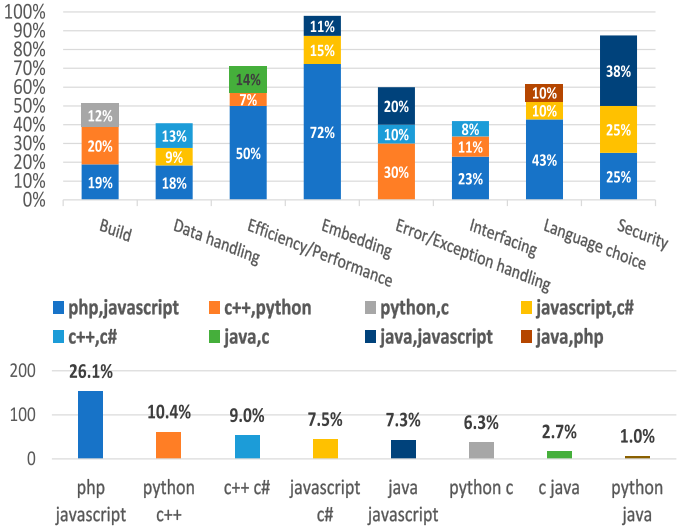

Fig. 11. The percentage distribution of top three language combinations involved in the posts of each issue category (top chart), and the percentage distribution of all the 586 posts over all such language combinations (bottom chart). The bottom chart serves as a reference for facilitating understanding the top chart.

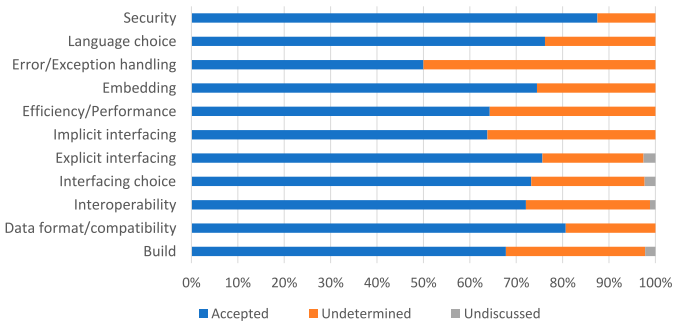

Fig. 12. The distribution of posts across answer status (accepted, undetermined, and undiscussed) for different issue categories.

across all categories. Such a trend for 'security' questions could be attributed to several factors: they might attract greater interest and attention, or perhaps questions in this category are relatively straightforward, eliciting more responses.

Conversely, Fig. 11 shows that the **Error/Exception** category is likely to be the most challenging (or least interesting to the post viewers) as it has the lowest answer acceptance rate of only 50%.As illustrated in Fig. 12, the average number of answers for the Error/Exception category stands at just 1.5, markedly trailing all other categories. We surmise that this could be due to the category receiving limited attention from developers or because questions within this category pose unique challenges, resulting in fewer responses and diminished accuracy in answers. Furthermore, we observed no statistically significant differences in the answer acceptance rate among the other issue categories (rates ranging from 63% to 81%).

As illustrated in Fig. 13, it is notable that the majority of these accepted answers were provided within 7 days of the question being posted, with nearly half of them being answered within just 24 hours. These findings support our initial speculation that multilingual projects receive significant attention from the

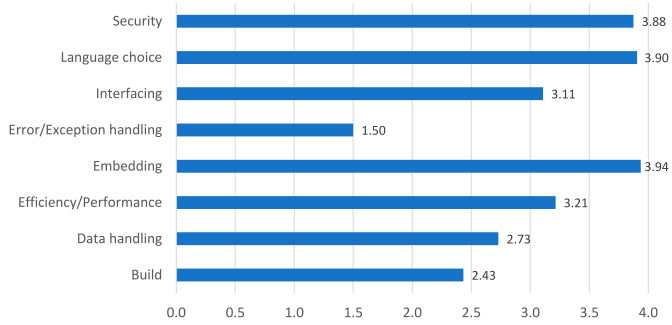

Fig. 13. The average number of answers to each category.

community and that most multilingual development issues are not intractable, with half of the issues being resolved in just one day. Even the most challenging questions can be answered within several days. These results suggest that the multilingual development community on SO is highly responsive to developers' questions, providing timely and effective assistance.

Using the answer acceptance rate of each issue category as an indicator of its difficulty level, Security related issues appeared to be relatively more resolvable than other issue categories (with an 87% acceptance rate) and Error/Exception handling issues the least resolvable (with the acceptance rate of 50%). The majority of all the issues received an accepted answer within 7 days.

C. RQ3: Challenges

We have chosen to concentrate on the 6 most frequent issue categories, which together account for 84% of all posts, in order to identify the shared challenges behind each category.

1) Build: We found that build issues mainly involved problems related to installation, compilation, configuration, and packaging, and we identified 3 common challenges that summarize the root causes of these issues. These challenges were observed in 15 out of the total 90 posts on multilingual software build issues.

Challenge 1: *Documentation Insufficiency: Insufficient or non-existent documentation was observed in the context of the discussed build-related topic.* Many developers expressed concerns about inadequate or missing documentation in their questions or comments. Specifically, among the relevant posts, developers encountered two types of situations related to documentation:

- *Missing documentation:* Missing documentation on the discussed topic is a recurring problem highlighted by many developers in their questions and comments. For instance, one relevant post [36] exemplifies this issue, where the `QWebChannel` JS API failed to set up in a `QWebEngineView` due to incomplete documentation about Qt. Specifically, the documentation related to the `QWebChannel` component was entirely missing, resulting in difficulties for the developer in resolving the issue.
- *Insufficient documentation*: Relevant documentation exists, but it only provides reference information or general

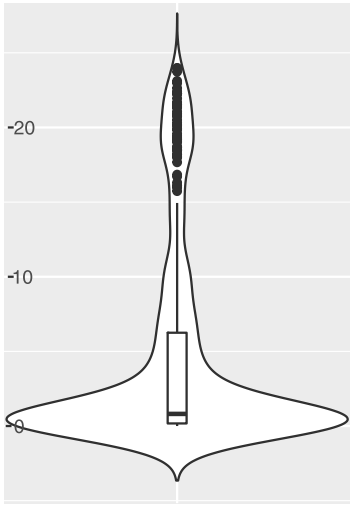

Fig. 14. Distribution of the number of days since a post was created until it received an accepted answer.

ideas for solving respective problems, as shown in the post cited as [37]. In this post, the developer faced an issue with splitting code into multiple modules in a general situation using `Pybind11`, but the documentation only addressed a special case and did not provide a complete method.

Challenge 2: *Insufficient Support: Support for multi-language code build is severely lacking.* Developers often face the challenge of inadequate compiler log messages and, as a result, they seek to build tools to enhance their work efficiency. However, the existing tools do not fully support multilingual projects. As depicted in Fig. 14, the developer was unable to remove unnecessary resources from a multilingual project during the build process due to inadequate build support. This was because the project was written in `C++-C#`, but there was no tool available to remove resources at runtime.

Challenge 3: *Language Evolution: As languages continue to evolve, certain language versions may become incompatible with previous versions. The use of multiple languages can exacerbate this issue, leading to even more compatibility challenges.* Computer languages undergo frequent version updates, leading to changes or abandonment of some functions or modules. Multilingual projects, which involve at least two languages, are particularly affected by this evolution. When one of the languages undergoes changes, it can impact the entire project environment. This problem arises due to the inherent complexity of multilingual environments. As illustrated in Fig. 15, C++ can interact with Python 2.7 using `boost`, but fails to do so with Python 3.2, highlighting the challenge of evolving languages and their incompatibility with current versions.

The posts related to Build issues primarily pertain to failures encountered during the build process, such as compilation errors, conflicts with language versions, and challenges with project maintenance. These difficulties arise due to various reasons, including inadequate documentation, insufficient support, and evolution of computer languages.

How do I remove unnecessary resources from my project?

Question: I am working with a very big project (a solution that contains 16 projects and each project contains about 100 files). It is written in C++/C# with Visual Studio 2005. One of the projects has around 2000 resources out of which only 400 are actually used. How do I remove those unused resources? I tried to accomplish the task by searching for used ones. It worked and I was able to build the solution, but it broke at runtime. I guess because enums are used. (IMPORTANT)

How can I make sure that it doesn't break at runtime? ...

Fig. 15. An example illustrating that tool support for building multilingual code is lacking, as a challenge causing the issue here: unnecessary resources were not successfully removed during the build process [38].

Hello world with boost python and python 3.2

Question: So I'm trying to interface python 3.2 and c++ using boost python, and have come across many many issues. I've finally gotten it to compile using the 2.7 libraries and it works, but I can't seem to make it work with python 3.2. ...

If I compile it using the 2.7 libraries it works just fine, but when I use the 3.2 libraries I get tons of undefined references from libboost_python.so ...

And the error from the python 3 interpreter is

```
File "<stdin>", line 1, in <module>
ImportError: /usr/local/lib/libboost_python.so.1.47.0: undefined symbol: PyClass_Type
```

Fig. 16. An example where language evolution causes compatibility issues between languages, as a challenge under build issues that the build succeeded with one version but failed with another one of the languages (Python) [39].

2) Data Format/Compatibility: Data format and compatibility issues often arise during data exchange, leading to program defects. These issues can occur due to incompatible data formats between languages. The underlying challenges of these issues can be classified into two main categories. These challenges are the primary causes of data format and compatibility issues discussed in 27 out of 88 posts.

Challenge 1: *Error-Prone Data Conversion: Multi-language software development projects can involve languages with different typing strengths and type systems, making conversions across such languages error-prone.* In addition, developers may face difficulties in comprehending the data type validity requirements and data conversion rules across different languages due to their distinctive typing strengths and type systems. The issue of type conversion, which is commonly observed in multilingual software development projects, is often associated with these challenges. The rules for type conversion vary among programming languages, with some languages having strong typing while others having weak typing. Weakly-typed languages often perform implicit conversions between data types and permit the compiler to interpret data items as having different representations arbitrarily. While this feature is convenient for developers, it can introduce errors in data format for multilingual development projects. For instance, as depicted in Fig. 16, converting data of the `dictionary` type in Python to a JS hash table is challenging, as Python and JavaScript use different systems to represent a dictionary internally.

Challenge 2: *Differences in Library Configurations: Different languages may have different third-party libraries, even if they aim to implement the same algorithm. Consequently,*

How can I convert Python dictionary to JavaScript hash table?

Question: I have passed to template regular Python dictionary and I need to inside $(document).ready(function() {.. } to convert that Python dictionary to JavaScript dictionary. I tried like var js_dict={{parameters}}; but I got errors (' instead of ' and all strings start with u'). How can I convert Python dictionary to JavaScript hash table?

Fig. 17. An example showing that data conversion is difficult in multilingual coding, as a challenge causing the issue: converting the Python dictionary to JavaScript hash table failed in this particular case [40].

libraries providing functions for these algorithms may differ for distinct languages. It is a common issue in multilingual software development that the same algorithm used in different languages can produce inconsistent results, despite the expectation that it should be identical. The root cause of this problem is that different languages employ different algorithm parameters, which results in data generated in one language being incompatible with another, even though both use the same algorithm. Third-party libraries provide different functions with varying parameters to implement the same purpose, and it can be challenging for developers to select the correct function when they are unfamiliar with it. Fig. 17 provides an example where the encryption result in the C# unit differs from that in the Java unit. This issue highlights the challenge of producing identical results in different languages, even when using the same algorithm, due to varying parameters.

The posts related to the Data Format/Compatibility issue primarily discussed the difficulties encountered during data conversion and the usage of third-party libraries. These issues stem from the variances in the data type systems used in different programming languages, as well as the differences in algorithm configurations implemented by various libraries.

3) Interoperability: This is a relatively complex problem, where issues related to data often arise during the interfacing phase, in addition to formatting problems. These issues include parameter configuration or type problems that occur when data is transmitted. The two most significant challenges underlying these interoperability issues were identified in 29 out of a total of 86 posts on the subject.

Challenge 1: *Memory-Access-Mechanism Discrepancy: Inconsistencies in memory management mechanisms, such as allocation and recycling, can lead to interoperation faults, such as buffer overflow and memory leaks, when working with different languages.* Some languages, such as C and C++, offer developers the ability to manage their own memory. While this may appear to be a convenient approach, it can pose risks in a multilingual environment. When multiple languages are used in a project, differences in the garbage collection mechanisms of each language can create development challenges for programmers. Additionally, passing data types like pointers in the code may result in memory management issues. For instance, as shown in Fig. 18, retrieving strings from Java using JNI can lead to memory leaks. This is because the distinct memory management mechanisms of different languages make interoperation between them difficult.

Encrypt AES with C# to match Java encryption

Question: I have been given a Java implementation for encryption but unfortunately we are a .net shop and I have no way of incorporating the Java into our solution. Sadly, I'm also not a Java guy so I've been fighting with this for a few days and thought I'd finally turn here for help.

I've searched high and low for a way to match the way Java Encryption is working and I've come to the resolution that I need to use Rijndael-Managed in c#. I'm actually really close. The strings that I'm returning in c# are matching the first half, but the second half are different. ...

Fig. 18. An example illustrating difficulties with encryption and decryption between different languages, as a challenge causing the data format/compatibility issue that the AES encryption result is different between C# and Java [41].

Memory leak using JNI to retrieve String's value from Java code

Question: I'm using GetStringUTFChars to retrieve a string's value from the java code using JNI and releasing the string using ReleaseStringUTFChars. When the code is running on JRE 1.4 there is no memory leak but if the same code is running with a JRE 1.5 or higher version the memory increases. This is a part of the code

```
msg_id=(*env)->GetStringUTFChars(env, msgid,NULL);
opcdata_set_str(opc_msg_id, OPCDATA_MSGID, msg_id);
(*env)->ReleaseStringUTFChars(env, msgid,msg_id);
```

I'm unable to understand the reason for leak. Can someone help? ...

Fig. 19. An example of garbage collection mechanism divergences across different languages leading to interoperation faults, as a challenge causing the interoperability issue that JNI retrieving strings resulted in a memory leak [42].

Challenge 2: *Incompatible Data Types: Incompatibility or conflicts in data types between languages can result in failed interoperation between the languages.* When two languages are unable to communicate directly, they rely on language interaction APIs to interact. Poor semantic interoperability often leads to issues in the data transmission process, resulting in buggy APIs. As shown in Fig. 19, the string-type data in C# is incompatible with the `wchar_t *` parameter type returned by the C++ function. This challenge is caused by the disparity in data type systems between the two languages.

The majority of posts regarding Interoperability issues were related to two main challenges: memory management discrepancies and interoperation failures between different languages. These challenges were caused by differences in memory management mechanisms and incompatible data types across languages.

4) Interfacing Choice: This problem is often encountered during the design phase of multilingual projects, especially among developers who are not familiar with this type of development. This lack of experience may result in inadequate knowledge of the effectiveness and reliability of language interactions. Overcoming this challenge could enable developers to gain a thorough understanding of the topic, thereby providing useful insights and references for their own multilingual projects. The two main challenges that underlie the Interfacing Choice issues, as identified in 42 out of 86 posts on this issue category, are highlighted below.

C# calling native C++ all functions: what types to use?

***Question*:** I want to make a native C++ all that can be used from a C# project. **1**. If I want to pass a string from C# to the function in the C++ all, what parameter should I use? **2**. I know that C# strings use Unicode, so I tried wchar_t * for the function but it didn't work; I tried catching any exceptions raised from the called function, but no exception was thrown. **3**. I also want to return a string so I can test it.

... What type should I use for the C++ function's return type, so that I can call it from C# with a return type of string[]? the same Q but for the parameter of the function to be string[] in C#?

Fig. 20. An example illustrating parameter type conflicts or incompatibilities, as a challenge causing the interoperability issue that `C#` failed to save the data of the `wchar_t *` type, which returns from the `C++` function, as a string [43].

Calling C functions in Python

***Question*:** I have a bunch of functions that I've written in C and I'd like some code I've written in Python to be able to access those functions.

I've read several questions on here that deal with a similar problem (here and here for example) but I'm confused about which approach I need to take. One question recommends ctypes and another recommends cython. I've read a bit of the documentation for both, and I'm completely unclear about which one will work better for me. Basically I've written some python code to do some two dimensional FFTs and I'd like the C code to be able to see that result and then process it through the various C functions I've written. I don't know if it will be easier for me to call the Python from C or vice versa.

Fig. 21. An example illustrating that the developer cannot make a choice on the interfacing mechanism between `ctypes` and `cpython`, as a challenge causing the interfacing choice issue that the developer is not familiar with the interaction across `C` and `Python` [28].

Challenge 1: *Language (Interfacing) Unfamiliarity: Developers often encounter difficulties with selecting proper language interfacing mechanisms when designing multilingual development projects, particularly when they lack familiarity with individual languages or the ways in which different languages interact.* Given the multitude of development languages and the continuous emergence of new language combinations, it is unrealistic for developers to attain mastery of every language. Therefore, developers face a unique set of challenges when dealing with multilingual development projects, including uncertainty surrounding language interfacing mechanisms, data transmission methods, and functions, which can cause hesitation when making choices. As shown in Fig. 20, developers may struggle to choose the right interfacing method choice existing options (e.g., between `ctypes` and `cython`) due to their limited familiarity with the various kinds of ways in which language interactions may be achieved.

Challenge 2: *Complex Interaction: Integrating different languages poses a significant challenge due to the complexity of language interactions and integration methods.* Each language has its own strengths and weaknesses, making it difficult to balance the trade-offs and choose the optimal method. Accordingly, different interfacing mechanisms have their pros and cons as well. As a result, developers often struggle to determine the most appropriate method to integrate the different language units for their multilingual projects. Fig. 21 illustrates the dilemma faced by developers when presented with multiple choices for integrating C and Python. Despite proposing three interfacing methods, the developer is uncertain about which one to use, since each of these options would come with varying advantages and limitations.

Wrapping a C library in Python: C, Cython or ctypes?

***Question*:** I want to call a C library from a Python application. I don't want to wrap the whole API, only the functions and datatypes that are relevant to my case. As I see it, I have three choices: 1. Create an actual extension module in C. Probably overkill, and I'd also like to avoid the overhead of learning extension writing. 2. Use Cython to expose the relevant parts from the C library to Python. 3. Do the whole thing in Python, using ctypes to communicate with the external library. I'm not sure whether 2) or 3) is the better choice. The advantage of 3) is that ctypes is part of the standard library, and the resulting code would be pure Python – although I'm not sure how big that advantage actually is. Are there more advantages / disadvantages with either choice? Which approach do you recommend?

Fig. 22. An example illustrating that the developer hesitates among the three C/Python interfacing methods, as a challenge causing the interfacing choice issue that it is complex to integrate different languages [44].

The posts related to the Interfacing Choice issues primarily centered around the selection of appropriate language interfacing mechanisms. The challenges arose from the complexity and diversity of these mechanisms, which can be difficult to choose for developers who lack familiarity with them.

5) Explicit Interfacing: Explicit interfacing refers to using a foreign function interface (FFI) such as JNI, CPython, etc., to integrate different languages. When developers choose a specific interfacing mechanism, they may encounter issues related to that mechanism. The most common challenge in Explicit Interfacing issues, as seen in 6 out of 78 posts on this topic, is related to errors occurring in the selected interface.

Challenge 1: *Threading-Induced Complication: The correct usage of explicit interfaces can be further complicated by multiple threading, as there may be special or additional requirements for sharing data or values across threads through these interfaces.* Thread control has been a persistent challenge for developers, and it is already complex in a single-language development environment. In a multilingual environment, this challenge is exacerbated by the fact that multithreading mechanisms may differ significantly across languages. For instance, the multithreading feature is not natively supported in the C++ language, and developers need to invoke low-level functions of the operating system to implement multithreading. As illustrated in Fig. 22, the developer intends to use the C# thread to retrieve a value from the C side. However, the developer is uncertain whether the variable in C needs to be declared as "volatile" in the multilingual project. Hence, the challenge stems from the additional requirement placed on the C side variables.

The posts on Explicit Interfacing issues mostly focused on threading and foreign function calls, which presented several challenges for developers. These challenges were primarily caused by the complexities of multiple threading in a multilingual environment and the varying multithreading requirements.

Does managed languages lock flush and reload variables of native libraries?

***Question*:** When we use locks in managed languages like C# and Java, we can always be sure we are dealing with the latest data. Specifically in Java memory model, they have a guarantee called Happens-before relationship. But I'm not sure what will happen with native libraries. ... As you see, if sharedData from C side is not declared as volatile, then is there still a guarantee that Thread 2 can always get the latest value set by Thread 1? Does the same apply to Java using JNI too?

Fig. 23. An example illustrating that multiple threading leads to special interfacing requirements in the multilingual project, as a challenge causing the explicit interfacing issue that the developer was confused about declaring the 'volatile' data type in multithreading between `C` and `C#` [45].

Content-Transfer-Encoding in file uploading request

***Question*:** I'm trying to upload file, using XMLHTTPRequest, and sending this headers: ... Content-Transfer-Encoding: base64 ... But on server side PHP ignore header "Content-Transfer-Encoding: base64" and write base64 undecoded data directly into the file! Is there any way to fix it?

Fig. 24. An example illustrating that the configuration is complicated for the developer in an HTTP-Request, as a challenge causing the implicit interfacing issue that the configuration in the header that is aimed to encode the content to the `base64`-type data on the server side does not work [46].

6) Implicit Interfacing: In implicit interfacing, methods of an interface are applied without explicitly specifying the interface name. This type of interfacing is commonly used in web-based multilingual projects where HTTP-Request (including Ajax) is the most frequently used method, followed by Socket (using Socket directly), pipe, and memory sharing. We will elaborate on the top two challenges that were mentioned in 20 out of 59 entries on these problems.

Challenge 1: *Complex Message-Passing Configuration: Implicit interfacing, which is frequently utilized in multilingual projects for message passing between languages through high-level protocols like HTTP request or Socket, can be a complex and error-prone process due to its intricate configurations.* Developers often face challenges when programming sockets, especially when dealing with header and token components. Similarly, working with headers and forms in HTTP requests can be difficult. Even when utilizing Ajax within a `jQuery` wrapper, developers may encounter configuration challenges. For instance, as depicted in Fig. 23, a developer misconfigured the interface between JavaScript (which sends the request) and PHP (which receives the request) through HTTP requests. The data encoding was wrongly specified in the request header, and as a result, the receiver used undecoded data, which is incorrect.

Challenge 2: *Diversity of Message Passing: Cross-language message passing through implicit interfacing involves handling message responses in different ways, often requiring the use of various frameworks.* Cross-language message passing through implicit interfacing involves handling message responses in different ways, often requiring the use of various frameworks. As depicted in Fig. 24, developers may encounter obstacles when using Ajax to transfer data via a specific framework, such as `Krajee Bootstrap`. In this case, catching the response may be difficult for developers who are not familiar with the differences in handling responses between frameworks, especially when compared to using `jQuery`. This highlights the challenge of dealing with the various ways in which message responses can be handled, which may lead to difficulties in their proper handling.

Krajee Bootstrap File Input, catching AJAX success response

***Question*:** I'm using Krajee the Bootstrap File Input plugin to perform an upload via AJAX call. Here is the link to the Krajee plugin AJAX section: Krajee plugin AJAX.

Right now I get a response from PHP whatever it is an error or a success as JSON, I have went through the plugin documentation and I still can't find how to catch the AJAX response and act according to that response as we do in jQuery with the ajax success function: success: function (response) {} How can I do this?

Fig. 25. An example illustrating that the response of Ajax is complicated for the developer, as a challenge causing the implicit interfacing issue that the developer is unaware of catching the response since he only knows the function that can work in `jQuery` to catch the response [47].

The issues related to Implicit Interfacing primarily involved managing message passing and handling requests and responses. These challenges arose from the intricate nature of configuring message passing and the wide range of methods available for handling requests and responses.

D. RQ4: Solutions

After identifying the challenges, we proceeded with looking into the solutions currently available for addressing the primary challenge in each of the 6 major issue categories we identified as common challenges in RQ3.

1) Documentation Insufficiency Challenge: We did not find any universal solutions for the documentation insufficiency challenge underlying the *Build* issues. However, we did come across two helpful suggestions.

Solution 1: According to the accepted answers, including external links can be helpful for the asker in resolving their issue. In some cases, posts even provide direct excerpts from the relevant documentation, which can be valuable for understanding and addressing the problem.

The example in Fig. 25 illustrates a scenario where a developer faced a challenge due to insufficient documentation. This challenge prevented the developer from successfully setting up the `QWebChannel` JavaScript API for use in `QWebEngineView`. In addition, the developer is confused about the <script> tag usage of `QWebChannel` on the HTML page. To tackle this issue, the developer was offered some alternative sources to resolve the problem with Qt's `QWebChannel` JavaScript API and the confusion around the usage of the <script> tag. In essence, the developer can explore alternative sources on this Q&A platform (i.e., SO) or other forums where developers discuss such issues.

Solution 2: The accepted answers to the analyzed posts indicate that code examples were sometimes provided, which helped developers immediately comprehend how to solve their problems.

As depicted in Fig. 25, in addition to the alternative sources, the developer was also given a code example that illustrates how to set up the `QWebChannel` JavaScript API using a <script> tag in the web page, which helped them solve the issue. In sum, developers may find it helpful to seek out code examples when facing challenges related to insufficient documentation.

How to setup QWebChannel JS API for use in a QWebEngineView?

Question: ... As Qt's documentation is far from complete, which are the ways to setup QWebChannel and are there different ones from adding a <script> tag in your HTML page like recommended in documentation?

Answer: ... Using runJavaScript() to execute the code in Qt's QWebChannel JS API: ...<code snippet>...

Sources: - QT QWebEnginePage::setWebChannel() transport object
- How do I include a JavaScript file in another JavaScript file?
- How to use Qt WebEngine and QWebChannel?
- http://doc.qt.io/qt-5/qtwebchannel-javascript.html

Fig. 26. An example illustrating that external references and code example can provide useful information for developers to overcome documentation insufficiency [36].

To address the challenge of Insufficient Documentation in the Build issues category, the suggested solutions involved providing alternative sources such as external links or code examples that can offer relevant information.

2) Error-Prone Data Conversion Challenge: We found three generic solutions for the challenge with the error-proneness of data conversion underlying the *Data format* issues.

Solution 1: Based on the accepted answers, it was found that incorrect usage of data conversion functions can cause issues for questioners. Developers need to verify whether the function being used is appropriate and check for potential issues such as function call parameter mismatch.

One example of this challenge is shown in post [48], where a developer encountered difficulties in converting a PHP array to a JavaScript object. The attempted conversion using the `implode()` function resulted in a syntactic error. The suggested solution was to use the `json_encode()` function instead. Therefore, developers should carefully check the functions they use for data conversion.

Solution 2: When dealing with two languages, developers can utilize foreign functions, such as those found in `SWIG` and `ctypes`, to facilitate data type conversion. It is crucial for developers to search for specialized foreign functions that are suitable for accurate data conversion across the different languages being used.

One example of such a challenge is shown in the post [49], where the developer encountered an issue while converting data types from `jobject` to `jstring` using JNI code. The unsuccessful conversion led to a compile-time error. To resolve this issue, the accepted answer suggested using the correct foreign function and modifying the parameters as required. Therefore, developers should carefully choose and utilize dedicated foreign functions for data conversion to avoid errors.

Solution 3: Multi-language projects that rely on network transmission often use language-independent data-interchange formats such as JSON or XML for data conversion. These formats have good compatibility with many languages, making data conversion simple and efficient while reducing the likelihood of conversion-related issues.

As depicted in Fig. 26, a developer faced a challenge when attempting to convert a Python dictionary to its equivalent JavaScript dictionary. The direct conversion failed due to the difference in data types between the two languages.

How can I convert Python dictionary to JavaScript hash table?

Question: I have passed to template regular Python dictionary and I need to inside $(document).ready(function() {.. } to convert that Python dictionary to JavaScript dictionary. I tried like var js_dict={{parameters}}; but I got errors (' instead of ' and all strings start with u'). How can I convert Python dictionary to JavaScript hash table?

Answer: Python and javascript both have different ideas about how to represent a dictionary, which means that you need an intermediate representation to pass data between them. The most common way to do this is JSON, which is a simple lightweight data-interchange format.

Use the python json library to convert (or dump) your python dict into a JSON string. Then in the javascript parse the JSON string into a javascript dict. (If you are using JQuery, then use jQuery.parseJSON)

Fig. 27. An example illustrating that there is a common way for the dictionary conversion between Python and JavaScript, which is using JSON [40].

To overcome this issue, the recommended solution was to use JSON, a language-independent data-interchange format, to transmit data between Python and JavaScript. By doing so, data conversion problems can be avoided while achieving simplicity and efficiency. In summary, using a data-interchange format that is independent of specific languages is a viable solution to data-conversion challenges.

For the challenge of Data Conversion in the context of Data Format/Compatibility issues, we observed that solutions included verifying the correctness of conversion function calls (such as foreign functions for that purpose) and utilizing language-independent data-interchange formats.

3) Memory-Access-Mechanism Discrepancy Challenge: Two generic solutions were identified for the memory-access-mechanism discrepancy challenge underlying the *Interoperability* issue category.

Solution 1: When dealing with complex memory management in a multilingual environment, developers can try to avoid using pointers across languages or minimize the usage of data types that require manual memory allocation and deallocation.

Fig. 27 illustrates a situation where storing C++ pointers in C# can pose a challenge for developers. The developer was concerned about the stability and safety of storing pointer addresses from C++ code in C# units since C++ programming requires the pointer address to remain unchanged. To address this problem, the suggested solution was to use strings in `p/invoke` and convert them to C-style `char*` using the `MarshalAs()` function. In sum, the solution was to avoid using the pointers in C# to keep the pointer address unchanged in C++.

Solution 2: Improper memory allocation and release is another frequent cause of memory issues [51]. Due to the differing memory management mechanisms of various languages, developers often encounter errors when attempting to allocate or release memory across languages. For instance, when memory is allocated in one language and released from another, limited memory access across languages caused by permission issues may lead to errors.

In Fig. 28, the developer faced an issue where the application exits without throwing any exceptions while calling a DLL from C#, and the developer believed that the DLL file was not the issue. Upon investigation, it was determined that the problem was

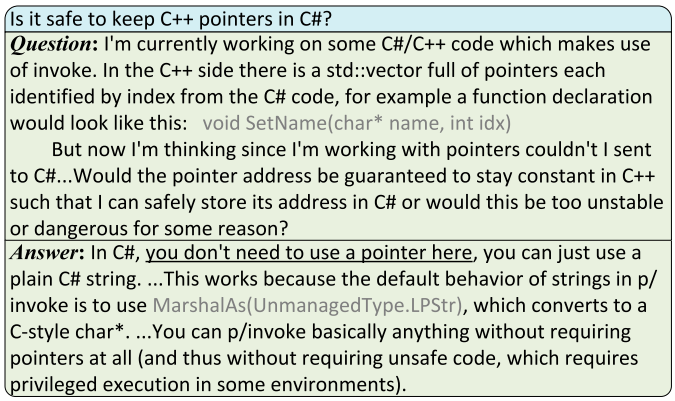

Fig. 28. An example illustrating that it is not necessary to use the *pointer* date type in C#; instead, the *string* date type may be used to solve the problem [50].

related to memory management. The developer attempted to release memory using the C# function `CoTaskMemFree()`, but because of the stricter memory manager in Vista and Windows 7, C# was unable to release the memory that was allocated in the C++ DLL. To address this problem, the suggested solution was to prevent the marshaller from attempting to release the string. Thus, it is important to avoid releasing memory from another language to prevent memory-related issues caused by different memory management mechanisms.

For the Memory-Access-Mechanism Discrepancy challenge underlying the Interoperability issue category, the current main solutions were to exercise caution when dealing with pointers and to minimize or even entirely eliminate memory operations across language boundaries.

4) Language (Interfacing) Unfamiliarity Challenge: We have identified from accepted answers a generic solution for the challenge of not being familiar enough with individual languages and/or language interfacing mechanisms in the *Interfacing choice* category of issues.

Solution 1: For developers who are not familiar with the interaction of different languages, it is recommended that they refer to official documentation or alternative references such as developer discussion or Q&A forums. This helps in understanding the characteristics of individual languages used and the availability and pros/cons of different language interfacing mechanisms.

As illustrated in Fig. 29, a developer faced the challenge of choosing between `Cython` and `ctypes` when attempting to write Python code to access C functions. After consulting some documentation, the developer became confused and sought advice on how to make the appropriate choice. The suggested solution was to compare the pros and cons of `ctypes` and `Cython` and determine which was better suited to the specific project requirements. Thus, by seeking appropriate references, developers can make informed decisions on selecting the appropriate interfacing mechanism.

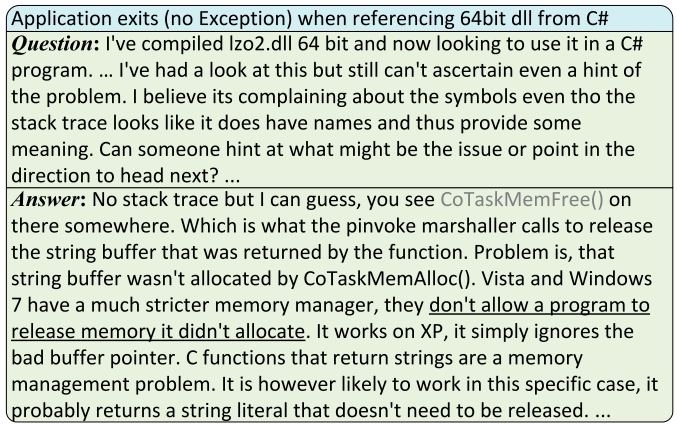

Fig. 29. An example illustrating that C# code cannot release the memory allocated by C++ code using the function `CoTaskMemFree()` [52].

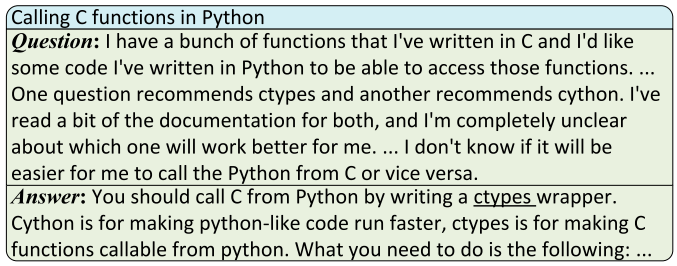

Fig. 30. An example illustrating that the `ctypes` interfacing mechanism is recommended to questioners, and some usage steps are provided [28].

For the Language/Language-Interaction Unfamiliarity challenge underlying the Interfacing Choice issues, the solution was to look for documentation or alternative references about features of individual languages and/or interfaces across languages.

5) Threading-Induced Complication Challenge: We have identified two general solutions for addressing the challenges posed by threading-induced complications that underlie the *Explicit interfacing* issue.

Solution 1: In the context of multithreading, it is important to note that the threading mechanism can differ across programming languages. For simpler multi-threaded interactions that involve calling functions without data exchange or mutual exclusion between threads, it is recommended that developers manage threads within the same language and avoid managing the same thread across different languages. This includes avoiding the creation of a thread in one language from another language, whenever possible.

The developer in Fig. 30 faced a challenge regarding calling a foreign function (written in Java) from multiple native threads (created in C) and concluded that it was not possible or that it would require the JVM to create multiple (Java) threads accordingly. This challenge arose due to the developer's unfamiliarity with Java's *native threading* mechanisms, which involve threading in native code via JNI.

The recommended solution was to use JNI and Java's *native threading* [53] mechanisms. The solution involves executing

concurrent calls in the native threads (not Java/JVM threads) without requiring the JVM to create its own threads. JNI also allows for managing these native threads by wrapping them in Java thread objects or attaching/detaching them using relevant APIs such as `AttachCurrentThread()` and `DetachCurrentThread()`.

Solution 2: Multilingual projects that use threads may encounter more complex thread activities [55], [56], such as mutual exclusion between threads, leading to challenges with explicit cross-language interfacing. If threading management across languages is necessary, developers can use mutex, a language-specific thread synchronization mechanism, such as Python's global interpreter lock (`GIL`), to properly manage the threads across the involved languages.

As an example, the post [57] describes a situation where a developer faced challenges in multithreading between C++ and Python. Specifically, the developer encountered an issue where the Python thread would stop working when returning to the main thread of the C++ program. This was due to the lack of proper thread synchronization, such as the use of `GIL`, to manage these threads across languages.

The suggested solution was for the main thread to acquire GIL and for the developer to use it to properly manage each thread. In summary, using GIL can assist the developer in managing threads between C++ and Python.

To address the challenge of Threading-Induced Complication that underlies the Explicit Interfacing issues, there were two suggested solutions. The first solution was to avoid managing threads across languages. If it is inevitable to manage threads across languages, the second solution was to use a mutex (for thread synchronization).

6) Complex Message-Passing Configuration Challenge: We found two generic solutions for challenges with complex message-passing configurations that often led to *Implicit Interfacing* issues.

Solution 1: Developers often encounter challenges when sending requests, such as HTTP requests, in implicit interfacing. To avoid related errors, developers should pay close attention to the configurations at both the client and server sides, including HTTP request headers from the client side, and follow appropriate configuration approaches. Relevant documentation can be consulted for guidance in this regard.

In Fig. 31, the developer attempted to configure the header "content transfer encoding: Base64" to make the server-side `PHP` automatically decode the base64 data. However, this did not work because the header "content transfer encoding" is only used if the value does not conform to the default encoding, which is "7BIT". To resolve this issue, it is recommended that the developer consult the relevant documentation for proper header configuration. In this case, the documentation would indicate that the header is not appropriate for this situation. Therefore, it is important for developers to check the usage of headers in relevant documentation to avoid similar errors.

Solution 2: Message-passing can pose a risk of errors when it comes to receiving data. To mitigate this risk, developers should take the necessary steps to identify the appropriate data format and transfer approach for accurate data reception. Additionally, it is important to thoroughly scrutinize the received data to detect and rectify any errors in the data format. Furthermore, it is imperative to ensure that function calls in response to the received data are made correctly to avoid any incorrect usage.

For instance, as illustrated in Fig. 32, a developer sends a post request to a Flask URL to retrieve a JSON data dictionary. Despite the post request successfully contacting the appropriate route[2], the function `request.get_json()` returns a None value. The suggested solution was that the developer should

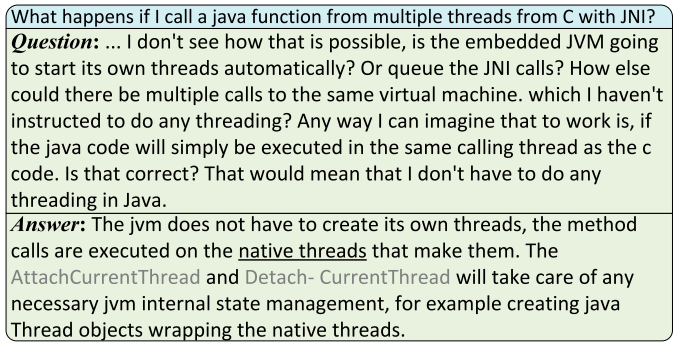

Fig. 31. An example illustrates that the `JVM` can use its own thread mechanism to handle multi-threading in JNI [54].

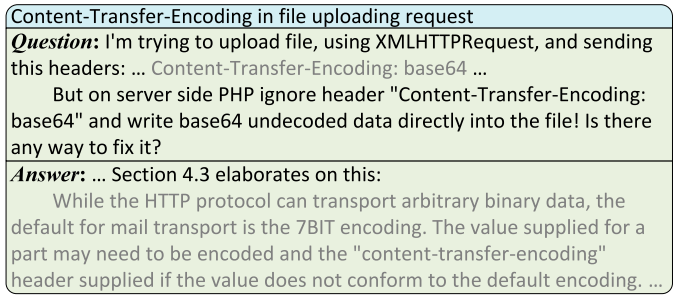

Fig. 32. An example illustrating developers misunderstanding usage of the header "content transfer encoding" [46].

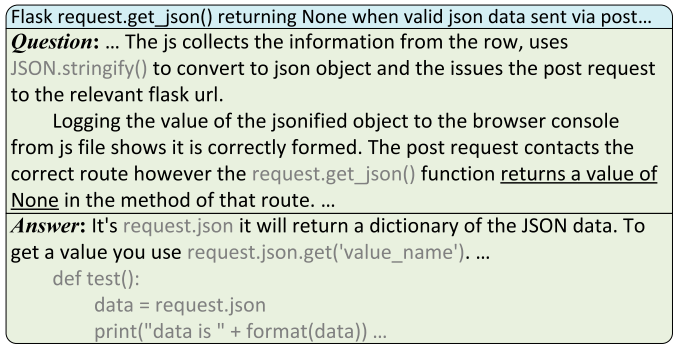

Fig. 33. An example illustrates the developer misusing the function to return and receive the data [57].

[2]In the Flask framework, routing is used to map an URL to function handling the task associated with the URL.

change the function used for retrieving the data since the correct function `request.json` will return the desired data (i.e., the JSON directory). In summary, developers must carefully check the data format and function used to retrieve the data to avoid errors in message passing.

To address the Complex Message-Passing Configuration challenge underlying the Implicit Interfacing issues, developers should ensure that the data configurations are correct on both the server and client sides. They should also use the appropriate approach to transferring data between the server and the client.

Note that none of the solutions presented for RQ4 were provided by the authors themselves. Rather, these solutions were provided by developers in a specific manner. When extracting common solutions, we attempted to make them as specific as necessary to be actionable. Additionally, the identified solutions were found to address challenges in different language combinations. Each solution was summarized from a number of posts, which often involved various language combinations, making the solutions applicable to a wide range of scenarios. Although the illustrating/example posts provided in the paper typically involved only one language combination, the solutions identified were rarely tied to a few specific language combinations.

V. Discussion

A. Implications of Our Findings

1) Actionable Suggestions for Developers: **The results of our investigation indicate that there are significant risks and challenges associated with multilingual software development. Developers should take these factors into consideration when making decisions about their projects.** In RQ2, our study identified common issues faced by developers in multilingual development, including challenges related to interfacing, data handling, and building a cohesive multi-language system. We also found that certain types of issues were more prevalent in specific language combinations, such as 72% of Embedding issues being mainly encountered in `PHP-JavaScript` projects. In RQ3, we revealed the root causes of these challenges for multilingual development. We found that issues with data handling and interfacing across different languages were caused by incompatibilities in data type systems. Furthermore, we identified difficulties in memory management and multi-threading operations as additional challenges for developers working with multiple languages.

The findings from RQ2 and RQ3 highlight the potential risks and challenges involved in multilingual development, and provide valuable insights for developers to make informed decisions based on their specific context. For instance, developers should consider these risks and challenges when deciding whether to adopt a single-language or multilingual development approach, and if they choose the latter, which languages to use and how to design the system across languages to mitigate potential risks. Additionally, certain issues are more commonly associated with specific language combinations, and developers should take this into account when selecting languages based on their system design and requirements.

Our findings offer practical guidelines to address the challenges in multilingual development. In RQ4, we distilled the solutions proposed in SO answers to tackle the common challenges that arise in multilingual development, particularly with regards to data handling and interfacing across languages. For example, for the two challenges discussed in RQ4, the current solutions involve managing threads/pointers/memory within each language unit, as attempting to do so across languages could potentially cause issues due to language semantics disparities. As for the other two challenges, the suggested solutions involve avoiding isolation and instead using a more universal, language-agnostic data format such as JSON. We advise developers to keep these common solutions in mind when working on multilingual projects.

2) Actionable Suggestions for Researchers: **Further research is recommended to explore and develop methods and tools for detecting conflicts and incompatibilities of data types and formats across different languages.** In Section IV.C.2, we have highlighted that data format and compatibility issues often result in error-prone data conversions. Therefore, we encourage future research to focus on developing techniques and tools for detecting such issues. One possible direction for projects using strongly-typed languages is to perform static analysis on all of such languages to infer the types and formats of converted data and subsequently detect any conflicts. On the other hand, detecting conflicts for weakly-typed languages is more challenging since the type is only determined during runtime. In this case, one possible approach is to use machine learning techniques to infer the type based on the surrounding code, as has been explored in prior studies [58], [59].

It is common for developers to face challenges when utilizing or selecting suitable APIs for handling data across different languages. To address this issue, further research is needed to enhance the provision of guidance and support for identifying and implementing appropriate APIs and their corresponding usage. In Section IV.C.3, it was observed that lack of familiarity with multilingual APIs and erroneous API selection [60] poses a significant challenge. To address this, a potential solution is the development of an API recommender system for multilingual projects. However, current research on API recommendations is not adequately tailored for multilingual development scenarios. Most of the existing state-of-the-art approaches rely heavily on machine learning algorithms and training data [61], [62], while a dearth of multilingual code corpus hinders progress [63]. Therefore, future studies should focus on creating a multilingual project dataset and train the API recommendation model accordingly, building on the approaches used in previous studies [61], [64].

It is recommended that future research endeavors to create tools that facilitate the development of multilingual projects. As highlighted in Section IV.C.1, the absence of tool support for building multilingual code poses a significant challenge. Current building tools are unable to adequately support the diverse build environments and configurations of different

languages, making it difficult to build multilingual projects. This is particularly problematic in the DevOps environment, where all build and deployment processes need to be automated and continuous [65]. The lack of adequate build tools may have impeded the DevOps workflow. Therefore, we recommend that future research should focus on developing multilingual software-compatible build solutions that incorporate more comprehensive multilingual build support tools. Additionally, integrating data type/format incompatibility/conflict detection and cross-language API misuse detection tools into such build support would further enhance the utility of the toolset.

Developers frequently encounter difficulties in selecting the most suitable interfacing mechanism. To mitigate this challenge, future research could explore different interfacing mechanisms and develop a tool capable of recommending optimal options to developers based on their specific requirements. As discussed in Section IV.D.4, our research findings indicate that developers working on multilingual projects often encounter challenges in selecting an appropriate interfacing mechanism. The wide range of available options can make it difficult for developers to have a comprehensive understanding of each mechanism. Furthermore, the lack of research in this area exacerbates the problem. Therefore, it is imperative to conduct a thorough analysis and evaluation of the unique characteristics of each interfacing mechanism [63]. Based on these insights, researchers could develop a tool capable of recommending the most suitable interfacing mechanism for a given scenario, taking into account various factors such as cross-language compatibility and code size based on the project requirements and language selection [3].

B. Threat to Validity

Threats to internal validity. Our study heavily relied on manual analysis, which raises concerns regarding potential human bias in the labeling process. To address this issue, two authors independently examined each post and labeled the corresponding challenge and solution. Any discrepancies were discussed until a consensus was reached. In RQ2, we utilized the LDA model to filter out irrelevant posts, thereby reducing the number of posts requiring manual labeling. While this approach may have resulted in some relevant posts being missed, our approach achieved a 95% recall rate, indicating that we covered the majority of relevant posts. Additionally, we filtered out posts with fewer votes, which may have caused us to overlook some useful posts and introduced bias to our results. However, we believe that posts with low votes tend to have lower value and relevance, justifying our decision to exclude them.

Threats to external validity. Our study relied solely on analyzing posts from Stack Overflow (SO), which raises concerns about the generalizability of our findings to other programming-related Q&A websites. This limitation poses a validity threat to our study as it questions the extent to which SO accurately reflects the challenges multilingual developers face in the field.

However, SO is currently the most accessible and widely used data source for our study. It is a well-known repository where developers post questions and receive answers, and has been frequently utilized in prior software engineering studies [25], [66], [67]. We thus assume that SO reasonably reflects the issues and challenges faced by developers, including those related to multilingual development. Nevertheless, this assumption introduces a validity threat. Therefore, we recommend that future research investigate this problem through other forms and platforms, such as surveys or GitHub issues.

Threats to construct validity. As described in Section III-B, the language-choice-based filter we used during data collection in this study was applied to make the study more manageable, as the manual effort required would have been much greater without this filter. However, this filter may have limited the comprehensiveness of our taxonomy. To address this, future research could consider dropping this filter and collecting data on a wider range of languages. It should be noted that our current language choices were based on the popularity of the languages, as they are commonly listed among the most popular languages in various rankings, including one from Stack Overflow itself. However, it is possible that making alternative or different choices may have resulted in a different taxonomy. Therefore, the language-choice-based filter constitutes a construct validity threat to our study.

Given the manual nature of our study, we could not track the evolution of developers' discussion on multilingual development over time. Analyzing the over 500 posts took over a year, and focusing on yearly evolution would limit us to about 50 posts annually if we keep the same our study scale overall, which is insufficient for us to draw reasonably useful and valid conclusions. On the other hand, examining over 500 posts per year manually while covering multiple years would be too tedious and costly to be practical. Nonetheless, from a holistic perspective, future studies may benefit from tracing the evolution of challenges and issues in multilingual development across different years. Such an approach would offer insights that more accurately mirror the actual landscape, particularly in light of the evolving nature of multilingual software and the changing trends in its development practices.

VI. Related Work

A. Studies on Multilingual Software Development

A number of studies have been conducted on the subject of multi-language software development, as evidenced by the presence of relevant prior works [1], [2], [3], [10], [11], [13], [68], [69], [70], [71]. Einarsson and Gentleman [71] were the first to introduce the concept of mixed language programming, which is now known as multi-language software development. Abidi et al. [2] surveyed 93 developers to assess the impact of multi-language design practices on software quality, while we analyzed Stack Overflow posts to investigate the same issue.

Mayer et al. [13] mined 1,150 open-source projects on GitHub and found that multilingual programming is prevalent in such projects. They also identified cross-language links as a common issue that can cause problems for developers [1]. Our analysis of Stack Overflow posts also revealed similar issues, such as difficulties in data handling and interfacing across different languages.

In contrast to previous studies that focused on understanding the practice of multi-language software development by analyzing open-source projects and conducting surveys, our approach involves examining issues, challenges, and solutions by analyzing real-world problems and solutions discussed in Stack Overflow posts.

B. Studies on Software Development Issues Using Stack Overflow

Stack Overflow has been used as a data source in several studies that investigated developers' challenges across various domains [22], [23], [24], [66], [67], [72]. For example, Abdalkareem et al. analyzed the impact of reusing code from Stack Overflow on Android development and observed that code reuse often leads to development issues [22]. Similarly, Meng et al. manually examined Stack Overflow posts to identify security risks associated with using code snippets from the platform and to assess the gap between specification and implementation of secure coding practices [23]. Yang et al. explored the trends and evolution of security-related topics by analyzing Stack Overflow posts over time [66]. Wang et al. studied the challenges faced in developing big data applications by analyzing Stack Overflow posts related to Apache Spark [67]. In contrast, our study investigates a novel topic: the challenges, issues, and solutions related to multilingual software development.

C. Comparing With Preliminary Study

In the preliminary investigation [27], we undertook the task of characterizing 586 posts and manually analyzing the issues, challenges, and solutions highlighted in these posts. The present paper represents a significant expansion of that study in several respects. Firstly, we have introduced and examined a new research question that provides more comprehensive support for our motivation to explore multilingual development by demonstrating its prevalence. Secondly, we have provided a more detailed account of the LDA filter employed in the data collection, including the assessment of alternative keyword sets as the filtering condition to improve recall in post relevancy identification. Thirdly, to better understand the help developers received for various kinds of issues, we added an analysis on the difficulty level of each issue category in terms of the ratio of posts having received an accepted answer in each category. Fourth, to provide further inspiration for researchers and developers, this paper includes investigations and results on one additional challenge underlying Build issues and two key challenges underlying each of two additionally examined categories of issues (Interfacing Choice and Implicit Interfacing issues), as well as (a total of 3) main solutions to the primary challenge underlying each of the two additionally examined issue categories. Finally, we propose an additional set of actionable insights for researchers, namely, to develop a recommendation tool for interfacing mechanisms that can assist developers in selecting the most appropriate mechanism based on their specific requirements. In accordance with these additional results, the entire paper has been thoroughly updated to reflect the new holistic study.

VII. Conclusion

While previous research has examined multilingual development, there has been a lack of comprehensive studies on the challenges that developers encounter during this process and existing solutions available to them. In this paper, we conduct a manual analysis of developer discussions on Stack Overflow to investigate the issues, challenges, and solutions encountered in multilingual software development. Through our analysis, we identify and categorize the various challenges faced by developers during multilingual development. We then summarize and present the primary solutions to each dominant issue's root cause. Our study provides actionable insights and recommendations to researchers and developers of multilingual software through the consolidation of empirical findings.

VIII. Data Availability

Open science. Source code and datasets are all available in our artifact package and has been made publicly accessible.

Acknowledgment

We thank the reviewers for their constructive comments which helped us improve our original manuscript.

References

[1] P. Mayer, M. Kirsch, and M. A. Le, "On multi-language software development, cross-language links and accompanying tools: A survey of professional software developers," *J. Softw. Eng. Res. Develop.*, vol. 5, no. 1, pp. 1–33, 2017.

[2] M. Abidi, M. Grichi, and F. Khomh, "Behind the scenes: Developers' perception of multi-language practices," in *Proc. 29th Annu. Int. Conf. Comput. Sci. Softw. Eng.*, 2019, pp. 72–81.

[3] W. Li, N. Meng, L. Li, and H. Cai, "Understanding language selection in multi-language software projects on GitHub," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. Companion*, 2021, pp. 256–257.

[4] W. Li, A. Marino, H. Yang, N. Meng, L. Li, and H. Cai, "How are multilingual systems constructed: Characterizing language use and selection in open-source multilingual software," *ACM Trans. Softw. Eng. Methodol.*, early access, 2023, https://dl.acm.org/doi/abs/10.1145/3631967

[5] F. Tomassetti and M. Torchiano, "An empirical assessment of polyglot-ISM in GitHub," in *Proc. 18th Int. Conf. Eval. Assessment Softw. Eng.*, 2014, pp. 1–4.

[6] C. Jones, *Software Engineering Best Practices: Lessons From Successful Projects in the Top Companies*. New York, NY, USA: McGraw-Hill, 2010.

[7] D. P. Delorey, C. D. Knutson, and C. Giraud-Carrier, "Programming language trends in open source development: An evaluation using data from all production phase sourceforge projects," in *Proc. 2nd Int. Workshop Public Data Softw. Develop.*, 2007, pp. 1–5.

[8] P. S. Kochhar, D. Wijedasa, and D. Lo, "A large scale study of multiple programming languages and code quality," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng.*, vol. 1, 2016, pp. 563–573.

[9] K. Kontogiannis, P. Linos, and K. Wong, "Comprehension and maintenance of large-scale multi-language software applications," in *Proc. 22nd IEEE Int. Conf. Softw. Maintenance*, 2006, pp. 497–500.

[10] M. Abidi, M. S. Rahman, M. Openja, and F. Khomh, "Are multi-language design smells fault-prone? An empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 3, pp. 1–56, 2021.

[11] M. Grichi, E. E. Eghan, and B. Adams, "On the impact of multi-language development in machine learning frameworks," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2020, pp. 546–556.

[12] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in GitHub," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 155–165.

[13] P. Mayer and A. Bauer, "An empirical analysis of the utilization of multiple programming languages in open source projects," in *Proc. 19th Int. Conf. Eval. Assessment Softw. Eng.*, 2015, pp. 1–10.

[14] P. Mayer, "A taxonomy of cross-language linking mechanisms in open source frameworks," *Computing*, vol. 99, no. 7, pp. 701–724, 2017.

[15] M. Grichi, M. Abidi, F. Jaafar, E. E. Eghan, and B. Adams, "On the impact of interlanguage dependencies in multilanguage systems empirical case study on Java native interface applications (JNI)," *IEEE Trans. Rel.*, vol. 70, no. 1, pp. 428–440, Mar. 2021.

[16] H. Yang, W. Li, and H. Cai, "Language-agnostic dynamic analysis of multilingual code: Promises, pitfalls, and prospects," in *Proc. ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE), Ideas, Vis. Reflections*, 2022, pp. 1621–1626.

[17] W. Li, M. Jiang, X. Luo, and H. Cai, "PolyCruise: A cross-language dynamic information flow analysis," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 2513–2530.

[18] W. Li, L. Li, and H. Cai, "On the vulnerability proneness of multilingual code," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 847–859.

[19] W. Li, J. Ruan, G. Yi, L. Cheng, X. Luo, and H. Cai, "PolyFuzz: Holistic greybox fuzzing of multi-language systems," in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur.)*, 2023, pp. 1379–1396.

[20] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillere, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects," in *Proc. IEEE 37th Annu. Comput. Softw. Appl. Conf.*, 2013, pp. 303–312.

[21] "Stack overflow." Accesssed: Mar. 17, 2022. [Online]. Available: https://stackoverflow.com/

[22] R. Abdalkareem, E. Shihab, and J. Rilling, "On code reuse from StackOverflow: An exploratory study on android apps," *Inf. Softw. Technol.*, vol. 88, pp. 148–158, 2017.

[23] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty, "Secure coding practices in Java: Challenges and vulnerabilities," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 372–383.

[24] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? An analysis of topics and trends in Stack Overflow," *Empirical Softw. Eng.*, vol. 19, no. 3, pp. 619–654, 2014.

[25] Y. Wu, S. Wang, C.-P. Bezemer, and K. Inoue, "How do developers utilize source code from Stack Overflow?" *Empirical Softw. Eng.*, vol. 24, no. 2, pp. 637–673, 2019.

[26] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing, "What do developers search for on the web?" *Empirical Softw. Eng.*, vol. 22, no. 6, pp. 3149–3185, 2017.

[27] H. Yang, W. Lian, S. Wang, and H. Cai, "Demystifying issues, challenges, and solutions for multilingual software development," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE),*Piscataway, NJ, USA: IEEE Press, 2023, pp. 1840–1852.

[28] "Calling C functions in Python." Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/16647186/calling-c-functions-in-python

[29] V. Puzhevich, "Top programming languages to use in 2020." Scand. Accessed: Jan. 12, 2024. [Online]. Available: https://scand.com/company/blog/top-programming-languages-to-use-in-2020/

[30] S. Wang, T.-H. Chen, and A. E. Hassan, "Understanding the factors for fast answers in technical Q&A websites," *Empirical Softw. Eng.*, vol. 23, no. 3, pp. 1552–1593, 2018.

[31] A. Bhatia, S. Wang, M. Asaduzzaman, and A. E. Hassan, "A study of bug management using the Stack Exchange question and answering platform," *IEEE Trans. Softw. Eng.*, vol. 48, no. 2, pp. 502–518, Feb. 2022.

[32] "Scrapy." Accesssed: Mar. 17, 2022. [Online]. Available: https://scrapy.org

[33] D. M. Blei, "Probabilistic topic models," *Commun. ACM*, vol. 55, no. 4, pp. 77–84, 2012.

[34] E. R. Morrissey, "Sources of error in the coding of questionnaire data," *Sociol. Methods Res.*, vol. 3, no. 2, pp. 209–232, 1974.

[35] N. Gantayat, P. Dhoolia, R. Padhye, S. Mani, and V. S. Sinha, "The synergy between voting and acceptance of answers on StackOverflow-or the lack thereof," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, 2015, pp. 406–409.

[36] "How to setup QWebChannel JS API for use in a QWebEngineView?" Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/39649807/how-to-setup-qwebchannel-js-api-for-use-in-a-qwebengineview

[37] "With pybind11, how to split my code into multiple modules/files?" Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/53762552/with-pybind11-how-to-split-my-code-into-multiple-modules-files

[38] "How do I remove unnecessary resources from my project?" Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/1496731/how-do-i-remove-unnecessary-resources-from-my-project

[39] "Hello world with boost Python and Python 3.2." Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/7195959/hello-world-with-boost-python-and-python-3-2

[40] "How can I convert Python dictionary to JavaScript hash table?" Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/10073564/how-can-i-convert-python-dictionary-to-javascript-hash-table

[41] "Encrypt AES with C# to match Java encryption." Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/21890805/encrypt-aes-with-c-sharp-to-match-java-encryption

[42] "Memory leak using JNI to retrieve string's value from Java code." Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/915790/memory-leak-using-jni-to-retrieve-strings-value-from-java-code

[43] "C# calling native C++ all functions: What types to use?" Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/5368720/c-sharp-calling-native-c-all-functions-what-types-to-use

[44] "Wrapping a C library in Python: C, Cython or ctypes?" Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/1942298/wrapping-a-c-library-in-python-c-cython-or-ctypes

[45] "Does managed languages lock flush and reload variables of native libraries?" Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/56787106/does-managed-languages-lock-flush-and-reload-variables-of-native-libraries

[46] "Content-transfer-encoding in file uploading request." Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/5169434/content-transfer-encoding-in-file-uploading-request

[47] "Krajee bootstrap file input, catching AJAX success response." Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/29626410/krajee-bootstrap-file-input-catching-ajax-success-response

[48] "Convert PHP associative array into JavaScript object." 2014. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/21153805/convert-php-associative-array-into-javascript-object

[49] "How to convert JObject to JString." Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/14036004/how-to-convert-jobject-to-jstring

[50] "Is it safe to keep C++ pointers in C#?" Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/7057022/is-it-safe-to-keep-c-pointers-in-c

[51] W. Li, H. Cai, Y. Sui, and D. Manz, "PCA: Memory leak detection using partial call-path analysis," in *Proc. ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2020, pp. 1621–1625.

[52] "Application exits (no exception) when referencing 64bit DLL from C#." Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/8241732/application-exits-no-exception-when-referencing-64bit-dll-from-c-sharp

[53] "Understanding Java and native thread details." IBM. Accessed: Jan. 12, 2024. [Online]. Available: https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=threads-understanding-java-native-thread-details, 2019.

[54] "Multithreading with Python and C api." Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/29595222/multithreading-with-python-and-c-api

[55] X. Fu, H. Cai, W. Li, and L. LI, "Seads: Scalable and cost-effective dynamic dependence analysis of distributed systems via reinforcement learning," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, pp. 1–45, 2020.

[56] H. Cai and X. Fu, "$D^2$2Abs: A framework for dynamic dependence analysis of distributed programs," *IEEE Trans. Softw. Eng.*, vol. 48, no. 12, pp. 4733–4761, Dec. 2022.

[57] "What happens if I call a Java function from multiple threads from C with JNI?" Stack Overflow. Accessed: Jan. 12, 2024. [Online]. Available: https://stackoverflow.com/questions/8654519/what-happens-if-i-call-a-java-function-from-multiple-threads-from-c-with-jni

[58] A. M. Mir, E. Latoskinas, S. Proksch, and G. Gousios, "Type4Py: Practical deep similarity learning-based type inference for Python," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 2241–2252.

[59] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 607–618.

[60] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated python library APIS are (not) handled," in *Proc. ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2020, pp. 233–244.

[61] C. Chen et al., "Holistic combination of structural and textual code information for context based API recommendation," *IEEE Trans. Softw. Eng.*, vol. 48, no. 8, pp. 2987–3009, Aug. 2022.

[62] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 631–642.

[63] W. Li, L. Li, and H. Cai, "PolyFax: A toolkit for characterizing multi-language software," in *Proc. ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2022, pp. 1662–1666.

[64] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, 2015, pp. 858–868.

[65] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "DevOps," *IEEE Softw.*, vol. 33, no. 3, pp. 94–100, 2016.

[66] X.-L. Yang, D. Lo, X. Xia, Z.-Y. Wan, and J.-L. Sun, "What security questions do developers ask? A large-scale study of Stack Overflow posts," *J. Comput. Sci. Technol.*, vol. 31, no. 5, pp. 910–924, 2016.

[67] Z. Wang, T.-H. P. Chen, H. Zhang, and S. Wang, "An empirical study on the challenges that developers encounter when developing Apache Spark applications," *J. Syst. Softw.*, vol. 194, Aug. 2022, Art. no. 111488.

[68] M. Grichi, "Towards understanding modern multi-language software systems," Ph.D. dissertation, Ecole Polytechnique, Montreal, QC, Canada, 2020.

[69] M. Lopes and A. Hora, "How and why we end up with complex methods: A multi-language study," *Empirical Softw. Eng.*, vol. 27, no. 5, pp. 1–42, 2022.

[70] S. Buro, R. L. Crole, and I. Mastroeni, "On multi-language abstraction: Towards a static analysis of multi-language programs," in *Proc. 27th Int. Static Anal. Symp. (SAS)*, Virtual Event, 2020, pp. 310–332.

[71] B. Einarsson and W. M. Gentleman, "Mixed language programming," *Softw. Pract. Exp.*, vol. 14, no. 4, pp. 383–395, 1984.

[72] M. Bagherzadeh and R. Khatchadourian, "Going big: A large-scale study on what big data developers ask," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 432–442.