How are Multilingual Systems Constructed: Characterizing Language Use and Selection in Open-Source Multilingual Software

WEN LI, Washington State University, USA
AUSTIN MARINO, Washington State University, USA
HAORAN YANG, Washington State University, USA
NA MENG, Virginia Tech, USA
LI LI, Monash University, Australia
HAIPENG CAI*, Washington State University, USA

For many years now, modern software is known to be developed in multiple languages (hence termed as *multilingual* or *multi-language* software). Yet to this date we still only have very limited knowledge about how multilingual software systems are constructed. For instance, it is not yet really clear how different languages are used, selected together, and why they have been so in multilingual software development. Given the fact that using multiple languages in a single software project has become a norm, understanding language use and selection (i.e, *language profile*) as a basic element of the *multilingual construction* in contemporary software engineering is an essential first step.

In this paper, we set out to fill this gap with a large-scale characterization study on language use and selection in open-source multilingual software. We start with presenting *an updated overview* of language use in 7,113 GitHub projects spanning five past years by characterizing overall statistics of language profiles, followed by *a deeper look* into the functionality relevance/justification of language selection in these projects through association rule mining. We proceed with an evolutionary characterization of 1,000 GitHub projects for each of 10 past years to provide *a longitudinal view* of how language use and selection have changed over the years, as well as how the association between functionality and language selection has been evolving.

Among many other findings, our study revealed a growing trend of using 3 to 5 languages in one multilingual software project and noticeable stableness of top language selections. We found a non-trivial association between language selection and certain functionality domains, which was less stable than that with individual languages over time. In a historical context, we also have observed major shifts in these characteristics of multilingual systems both in contrast to earlier peer studies and along the evolutionary timeline. Our findings offer essential knowledge on the multilingual construction in modern software development. Based on our results, we also provide insights and actionable suggestions for both researchers and developers of multilingual systems.

CCS Concepts: • **Software and its engineering → Maintaining software**.

*Haipeng Cai is the corresponding author.

Authors' addresses: Wen Li, Washington State University, Washington, 99163, Pullman, USA, li.wen@wsu.edu; Austin Marino, Washington State University, Washington, 99163, Pullman, USA, austin.marino@wsu.edu; Haoran Yang, Washington State University, Washington, 99163, Pullman, USA, haoran.yang2@wsu.edu; Na Meng, Virginia Tech, Virginia, 24061, Blacksburg, USA, nm8247@vt.edu; Li Li, Monash University, Clayton, Victoria, 3800, Australia, li.li@monash.edu; Haipeng Cai, Washington State University, Washington, 99163, Pullman, USA, haipeng.cai@wsu.edu.

1049-331X/2023/11-ART $15.00
https://doi.org/10.1145/3631967

Additional Key Words and Phrases: Multilingual software, language use, language selection, language profile, functionality relevance, evolutionary characterization, association mining

1 INTRODUCTION

A number of studies have suggested that software written in multiple languages[1] (i.e., *multilingual* software—as opposed to *single-language* software which is developed solely in one language) is prevalent. For instance, 20% of the 9,997 projects sampled on SourceForge [35] used two languages while 12% of them used three [13]. A later, industrial report found that most of applications developed by top companies were written in 2 to 15 languages [23]. This is consistent with more recent studies finding that more than half of the randomly sampled open-source projects on GitHub were developed in two or more languages, despite the largely varying sample sizes (e.g., 729 [43], 1,150 [33], and 15,000 [44]).

Intuitively, the prevalence and dominance of multilingual software is justifiable given the impetus (e.g., benefits or even necessity) of using multiple languages in a single software project [1, 49]. Indeed, different languages have their own peculiar strengths and weaknesses [10, 36]. Thus, combining various languages could be a natural consideration by developers for building software that requires capabilities each best offered by one of the selected languages. For example, a web application may use Python, a general-purpose language (GPL), and HTML, a domain-specific language (DSL), to combine their productivity and presentation merits, respectively. Similarly, an IoT software developer may use Java for plug-in development to exploit its portability advantage, along with C for implementing system-level features to leverage its efficiency advantages. It is also common that the different components of a distributed software system [8, 15], like in Android apps [6, 7], are developed in different languages to benefit from the decoupled design.

So much as the employment of multiple languages reduces software costs and improves software development productivity, the *multilingual construction* paradigm also leaves larger room for quality threats, including functionality defects and security vulnerabilities [28], in the resulting systems. Intuitively, the more languages used in a system, the harder the quality issues across different language units could be diagnosed effectively. However, before we focus on developing tool support for multilingual software quality assurance, we must first address the basic knowledge gap about *how multilingual systems are constructed*. The lack of this knowledge creates immediate barriers for building quality multilingual software. For instance, while developers recognize the general benefits of using multiple languages in one project, it may not be straightforward for a developer to make decisions in multilingual software development as regards how many and which languages should be used given the (e.g., functionality) requirements [34]. These decisions may be particularly challenging yet important to make given the large [23] and growing [27] number of languages as well as the constant evolution of the languages (e.g., in their features and capabilities) [45]. In this context, understanding how developers of existing multilingual software have selected languages, the rationale behind the selection, and the potential changes in the decision-making over time is a first yet essential step towards helping future developers deal with similar decisions and informing language designers about future language design considerations.

The use of programming languages in software development has been studied, concerning the factors that affect the success of a language [10], the popularity of different languages [4, 25, 36, 43], interactions/relationships (e.g., similarity) across languages [4, 25, 46], as well as evolution of languages in these regards [10, 25]. Relevant studies also have addressed the effects of language use on the defect-proneness [43], quality risk of the lack of maintenance [46], and bug resolution characteristics (e.g., bug fix size and time) [50] of the resulting software. However, these prior studies *mainly* targeted single-language software, focusing on how *individual* languages were used.

[1] In this paper, we refer to as ‘languages‘ any computer languages, including but *not* limited to *programming* languages.

Studies looking at language use in multilingual software do exist, which addressed the prevalence of such software [34, 44] as well as good/bad practices for various quality factors in developing software [1]. Few of these studies looked into the possible rationale behind developers' choice of languages and justifications of language combinations in multilingual projects [49]. Moreover, several of these studies were based on developers' perceptions through surveys [1, 34] rather than examining the actual artifacts of multilingual software projects. Two studies [13, 33] investigated the associations among chosen languages in multilingual software based on actual project data. Yet, like other peer studies, they did not examine how the language combinations in a multilingual project may be justified with respect to development decisions (e.g., functionality category/domain or project topic). In particular, the more recent study [33] (as performed in 2015) did not apply an evolutionary perspective but rather considered the dataset (1,150 projects) as one single collection from GitHub. The other considered the evolution of language use during the years of 2000—2005 [13]; since it was conducted over a decade ago, the relevance of the results might have significantly deteriorated.

In this paper, we conduct a large-scale *characterization study of multilingual software* with a focus on their *multilingual construction* in terms of language use and selection (i.e., *language profile*), sampling those in the open-source community while taking a longitudinal lens. The goal of our study is three-fold, subsuming **three specific aims**: (1) provide an updated, multifaceted *overview* of language use and selection in contemporary multilingual software in terms of overall prevalence, language distributions, and language-combination popularity, (2) take a *deeper look* into the functionality relevance of language selection in terms of the quantitative association between these two factors, and (3) offer a *longitudinal view* of (1) and (2) in terms of the evolution of both. The key motivation is that outcomes of these aims would lead to an understanding of how corresponding multilingual development decisions have been made, which will inform future decision-making in relevant regards. These aims and outcomes differentiate our study from, and make it complement to, previous peer studies on the use of programming languages. While multilingual software construction concerns many other aspects (e.g., data model/format, architecture, and human factors), we chose to only study the language selection/use aspect in order to keep a necessary focus in a single paper.

Around these objectives, our study revealed a range of novel findings about multilingual software construction as highlighted below.

Overall statistics/characteristics on language use/selection. Among 7,113 projects we sampled, despite the large number of (296) languages in use as available unique choices, most of the studied multilingual software projects only used 2 to 5 languages (mean=4.5 and median=3). Similar results were reported earlier [44] (mean=6 and median=5) and [33] (mean=5 and median=4). We also found that languages widely existed which were used frequently but only lightly (in terms of the associated code size) in multilingual software (e.g., `shell` and `cmake`), albeit mainstream languages tended to be used both frequently and contribute significantly to the software code size (e.g., `c/c++` and `javascript`). The combinations of these mainstream languages also tended to be popular for developers (i.e., the top combinations of languages were often those of top individual languages used). In terms of how the used languages interact with one another, implicit interfacing (e.g., two languages exchange data via interprocess communication) mechanisms were used notably more often than explicit ones (e.g., the code of one language explicitly calls a function written in another language).

Functionality relevance/justification of language selection. Our study revealed a variable yet generally strong association between functionality domains and main languages, between main languages and language interfacing mechanisms, and between the interfacing mechanisms and language selections, which justifies the overall association between functionality domains and language selections being strong as well, in the studied multi-language projects; in contrast, prior work only focused on the correlation between software categories and *individual* languages [4]. Meanwhile, some language combinations were more

strongly associated with certain domains than others. These results offer a lens to an in-depth understanding of how languages are selected in multi-language software, an empirical reference for developers when choosing a typical language combination for a common topic/domain, and potential insights for program analysis researchers on what language combinations to focus on and analyze.

Evolution of language use and selection. The number of languages and the number of projects using multiple languages have both been increasing every year. This observation brings to light a growing trend in which developers are choosing to use various combinations of multiple languages to construct software systems quickly to keep up with increasing demands. Meanwhile, over time, the lists of top individual languages and top language combinations used in the studied multilingual systems were stable, although the absolute ranking of top combinations has changed. For specific software categories, the language combinations used to implement corresponding kinds of software changed from year to year; yet the primary language commonly kept stable (e.g., our study indicated that albeit the language combinations for the category of `End-user application` changed every year, `javascript` was consistently included in the language combinations for constructing software in that domain). In all, over the 10 past years, some language combinations were more stably associated with certain domains than others, and the association was generally less stable than that between individual languages and the domains. Moreover, language interfacing mechanisms had been being adopted in an increasingly diverse manner, signaling the growing construction complexity of multi-language software; meanwhile, implicit interfacing has generally maintained a consistent dominance over other language interfacing mechanisms over time during the studied 10-year span.

We have released all of the source code and data sets used in our study, as found here[2] and here[3]. We will turn them into archived open data with detailed documentation.

Paper organization. The rest of this paper is structured as follows. We start with our study design in Section 2, elaborating data collection, filtering, development of study toolkit, and experimental procedures. We then present our empirical results and major findings in Section 3, followed by an in-depth discussion of the insights and implications behind our results in Section 4, along with the various threats that may affect the validity of the results. After that, Section 5 relates our study to relevant prior works, right before we offer concluding remarks in Section 6.

2 METHODOLOGY

We first outline our research questions, which provide the overarching guideline for our study. Next, we describe our approach to answering these questions, starting with an overview of our study process and followed by elaborations on the datasets and data analyses used in our study.

2.1 Research Questions

In accordance with our study goal and specific aims (Section 1), we seek to answer the following three main research questions, for which the scope, rationale (justification), and approach are also outlined below.

- **RQ1** ***What statistical properties describe the overall characteristics of language use and selection in multilingual software?***
  Scope. We start with a basic empirical analysis of the overall language use in multilingual software, including the prevalence of individual languages versus that of various language combinations, as well as the distribution of languages within and across projects.

[2]https://www.dropbox.com/s/h515kgfufyi2mr1/Multilanguage_Tool.zip?dl=0
[3]https://bitbucket.org/wsucailab/multilangstudy/

Rationale. These overall statistical properties provide a recent, multifaceted overview about the language profile as a basic construction characteristic of multilingual software, fulfilling the *aim (1)* of our study. Also, given the age of closest prior studies and the evolution of languages [45], the general statistical properties also reveal an updated view of the characteristics of multilingual software regarding language use.
Approach. We started with a random sampling of 10,000 open-source projects on GitHub that has been active throughout the five past years (2015 through 2019) and received at least 1,000 stars. This initial process ended up with 7,113 projects that come with meta data necessary for our empirical analysis (e.g., language profile information and meaningful project descriptions). We calculated basic statistics of the dataset to compute the metrics and measures within the scope of this question (e.g., #languages used per project, ranking of top language combinations, language distribution in terms of the occurrences of unique languages and the size of code written in different languages, and mechanisms in which the selected languages interface with each other).

- **RQ2** ***How is language selection related to the functionality domain/topic in multilingual software?***
  Scope. We further characterize multilingual software by examining whether developer decisions in choosing which languages to use in these software projects may have been associated with the project topic or software domain in terms of their functionality categories. If such associations exist, we proceed to quantify the extent.
  Rationale. Intuitively, an essential milestone in understanding the practice of multilingual software construction is to understand *why developers select the particular languages they choose to use*. One avenue toward the milestone would be to measure the relationship between language selection and functionality domain/topic—prior work based on developer opinions suggested that one reason that developers chose to use different languages is their perception that each language offers the best features for certain functionalities of the software [1, 34]. While such a quantitative analysis as in our study focusing on this single aspect may not suffice for fully answering the *why* question, studying the functionality relevance of language selection should be a useful step, which fulfills the *aim (2)* of our study.
  Approach. With the same dataset used for RQ1, we computed the functionality topic from the natural-language project descriptions through topic modeling. We then used the extracted topics to form common domain names for the studied projects after manual normalization and calibration, following a principled approach (i.e., inductive and axial coding processes) to label the functionality domain of each project. Next, we discovered and quantified the correlation between functionality domains/categories and language combinations in the (7,113) projects through association rule mining. We also took a deep dive into these overall associations through the selection of main languages and language interfacing mechanisms, while examining the effects of *non-programming* languages (as opposed to programming languages) on the associations.
- **RQ3** ***How has multilingual software evolved in terms of language use and selection?***
  Scope. We lastly characterize the evolutionary dynamics of multilingual software, via a time-aware empirical analysis of how the overall language use and the association between language selection and project functionality category (i.e., the answers to RQ1 and RQ2, respectively) have changed over time.
  Rationale. One widely recognized norm of modern software is that in general they constantly evolve. For multilingual software in particular, it is reasonable to assume that the evolution of language use and selection plays a significant underlying role in the overall evolutionary dynamics of these software projects. Understanding the dynamics in the past would naturally help make informed decisions for future multilingual development and language design. For instance, studying how the functionality relevance of language selection has changed over time can provide insights on the same into the future. In addition, peer studies in the last decade [4, 33, 43, 50] provide rich insights into the language use in contemporary

software development yet without incorporating a longitudinal view, a gap we intend to fill so as to fulfill the *aim (3)* of our study.

Approach. We randomly sampled GitHub for open-source projects in the 10 past years (2010 through 2019) and used 1,000 projects for each year in our evolution study. With the per-year datasets, we performed the same empirical analysis for each year as used on the dataset used for RQ1 and RQ2. We then characterized the evolutionary traits of multilingual software in terms of language use (e.g., top language combinations and language distribution), selection (i.e., association between language combination and project functionality category), and language interfacing mechanisms from the per-year characterization results.

2.2 Study Overview

To answer the above three research questions, we propose an orchestrated characterization study whose overall process flow is depicted in Figure 1. As its primary input, the process takes the Git repositories of open-source projects on GitHub [17]. From this data source, we mined the repositories of different numbers of projects for two complementary characterization studies. The first considers all projects to be from a single period (hence referred to as *single-period characterization (SPC)*), which aims to answer RQ1 and RQ2 based on the projects from the five past years (2015-2019) as a whole. The second characterization considers projects per year (referred to as *evolutionary characterization (EVC)*), which aims to answer RQ3 based on projects from each of the 10 past years (2010-2019) separately.

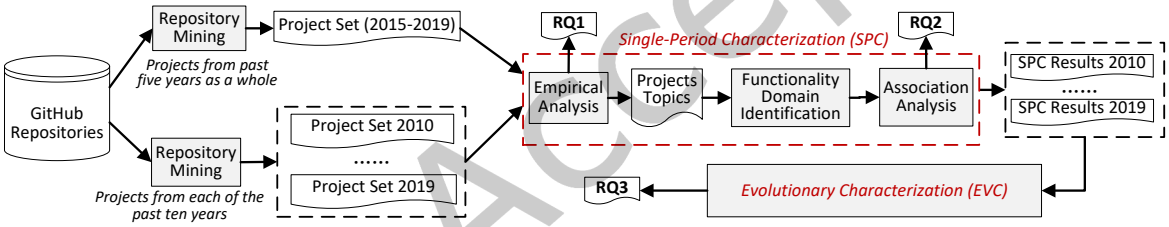

Fig. 1. Overview of the process flow of our multilingual software characterization study.

More specifically, in the SPC study, an empirical analysis is performed to compute basic statistical properties of the dataset to answer RQ1. This analysis also extracts project topics, from which project functionality domains (i.e., categories) are identified through topic modeling. Then, we computed the relationships between the resulting categories and top language selections via an association analysis to answer RQ2. In particular, we first look at the overall association between functionality domain (FD) and language selection (LS), followed by a closer look into such associations through two key underlying factors of LS in relation to FD: main language (MaL) and language interfacing mechanism (LIM). The rationale here is that when developers select languages for a target functionality domain, it is intuitive for them to first select MaL, then LIM according to the MaL chosen, and finally the other languages hence the entire language combination (i.e., language selection).

In the EVC study, per-year characterization results are computed first by running the SPC on the projects from each year. The resulting per-year SPC results are then used to characterize the evolutionary dynamics of all the projects across the 10-year span hence answer RQ3.

Next, we elaborate on the two characterizations separately, after describing the common *repository mining* step that produced the dataset used for each characterization.

2.3 Repository Mining

To obtain the datasets needed for our characterization studies, we retrieved raw project repositories from GitHub followed by two data processing steps, as described below. These three steps constitute the data collection procedure that we applied for *each project*.

Raw repository retrieval. We accessed each original project repository using the GitHub's Version 3 Python API [18]. In particular, we retrieved repositories that meet three criteria as listed and justified below, each corresponding to a project property specified when invoking the repository query via the API.

(1) *Popularity.* We used the property "star" as the popularity indicator of a project as in prior work [38, 43]: the larger this property's value, the more popular the project. We only chose projects that had at least 1,000 stars as these projects were considered popular ones [38]. Given the huge number of projects on GitHub, we intended to focus only on characterizing software that likely represents influential development practices, with respect to the goal of our study. While high popularity may not necessarily indicate high quality, it is reasonable to assume that the more popular projects more likely represent the practices that have greater influence.
(2) *Creation time.* We used the property "created" as the indicator of when the project was created. We only chose projects that were created in no more than 10 years. The rationale is that a project likely does not represent contemporary software characteristics if it is too (over 10-years) old. Also, none of our two characterizations would look beyond the span of 10 past years (2010 through 2019).
(3) *Update time.* We used the property "pushed" as the indicator of the latest time the project was updated. We only chose projects that have been updated in the latest six months (relative to July 2022 when we started sample project crawling). The rationale is that more active projects tend to more likely represent *ongoing* software development practices.

While GitHub offers a valuable source for software data mining, there are multiple perils in conducting studies based on GitHub mining [24]. For example, most of the projects on GitHub were found inactive. It is for this reason that we applied multiple selection criteria as described above. But we have additionally applied a basic filter in light of another peril: many projects on GitHub are not used for software development activities [24]. Thus, we have skipped repositories that are not for software development projects (e.g., personal/course websites and tutorials)—assisted by automated, conservative checking, we have spent extensive effort to apply this filtering step. In particular, we manually defined a list of heuristic words that might be indicative of common kinds of non-software-development projects on GitHub based on our experience (e.g., "tutorial", "course", "website", "book", and "list"). We then applied this conservative list to automatically check against project descriptions to obtain a rough list of potentially non-software-development projects. Lastly, we manually validate each project in this rough list as software development versus non-software-development projects by reading any documentation available and the source code as much as necessary.

Mining key project properties. Given a project's URL, detailed project information can be obtained (using the GitHub API). Among the over 70 different project properties available, most were irrelevant for our current study. The relevant ones are listed in Table 1.

These four fields contained pertinent information for our study. Accordingly, we set two constraints: (1) a project must contain at least one valid topic, which should not be just a programming language name; (2) a project's description must contain at least 5 characters. These minimal requirements were set to ensure that we have valid information to derive the project's functionality category.

Mining language information. Information on the languages used in a project is key to our study. With the language URL (Field 2 of Table 1) retrieved for a project, the GitHub API enables us to query the detailed language information for the project. The information we gathered includes the number of languages and the number

Table 1. Key project properties retrieved during the software project repository mining for our study

No.	Field	Description
1	`id`	Repository Id
2	`language URL`	the hyperlink for querying the project's languages
3	`topics`	List of topic tags
4	`description`	Simple description of the project

of bytes written in each language. As an example, the query result for a project was {`'css': 71539, 'html': 17627, 'javascript': 992797, 'shell': 340`}. We refer to this information as the ***language profile*** of the project, where the numbers indicate the number of bytes written in each language (referred to as the ***language code size*** of the language). When a project's profile size (i.e., #languages used) is greater than 1 (i.e., *potentially multi-language* project), we detect/classify the interaction between the languages using our multi-language software characterization tool PolyFax [29]. We only keep the project if is a *really multi-language* project (i.e., there exists an interaction between the languages).

2.4 Single-Period Characterization (SPC)

SPC is the core component of our study process. It takes as input a set of projects without considering any time information about each project, and then computes basic statistics about these projects as a whole via an *empirical analysis*, followed by *functionality domain identification* and then an *association analysis*.

Dataset. We started with randomly sampling 10,000 projects from GitHub that met the basic criteria for our raw repository retrieval step but limited the creation time to the five past years. The rationale for this time length is that through the SPC study we intended to take one sizable, single-period sample from GitHub to characterize the multilingual software in the sample as a whole, for which five years represent a reasonable length of period. And looking at the particular window of the *past* five years (2015-2019) ensures the recency of this single sample. From the 10,000 initial projects, we ruled out those that did not satisfy the two constraints set against the key project properties (Table 1), which left us 7,113 projects. The data mined from the corresponding 7,113 repositories by following our data collection procedure (Section 2.3) formed the basis for SPC.

2.4.1 Empirical Analysis. This analysis computes the following measures over the given input project set. The rationale of computing these measures is that they constitute a basic overview of language use in multilingual software.

- *Average Language Profile Size (ALPS)*: we first retrieve per-project language profile sizes (i.e., the number of languages used in a project), and then compute the mean and standard deviation (*stdev*) of language profile sizes across all projects in the given set.
- *Language Profile Size Distribution (LPSD)*: this is the percentage distribution of projects in the given set over different numbers of languages used. For example, if we have a set of 10 projects, of which 3, 5, 2 projects use 1, 2, 3 languages, respectively, then this measure would be {1: 0.3, 2: 0.5, 3: 0.2}.
- *Language Distribution By Frequency/code Size (LDBF/LDBS)*: we measure the popularity of each language through two measures: frequency (number of occurrences) and code size (number of bytes). From the language profile of each project, we compute these measures for all the projects in the given project set.
- *Average Language Code size Percentage (ALCP)*: we first compute the percentage distribution of code size of all the languages used in each project from its language profile. For example, given a language profile {`'c#': 569869, 'javascript': 198348, 'shell': 317`}, the distribution is {`'c#': 0.74, 'javascript':`

`0.25, 'shell': 0.01`}. Then, by averaging these percentage distributions over all projects in the given set, we compute the ALCP per language for the project set.

- *Top Language Combinations (TLCO)*: we compute the top combinations from the given project set according to the number of occurrences of each combination. Yet we only consider mainstream languages which have the greatest influence on the functionality of software [4, 43], given that the number of language combinations is an exponential of the number of unique languages.
- *Language Interfacing Mechanisms (LIM)*: Intuitively, looking at the structure of a multi-language software project helps understand the construction of the software. And we believe the most essential and unique aspect of this structure, as opposed to that of single-language software, is the language interfacing mechanism. Hence, examining the association between language selection and language interfacing mechanism offers a useful angle into the rationale of language selection/use in multi-language software. To that end, we compute the language interfacing mechanisms for the studied projects with our multi-language software characterization tool *PolyFax* [29]. *PolyFax* classifies the *LIM* of a given project into four categories: *Foreign Function Invocation* (FFI)—one language provides a foreign function interface to match its semantics and calling conventions with those of another language, *IMplicit Invocation* (IMI)—one language interacts implicitly with another language via interprocess communication (IPC), *EmBoDiment* (EBD)—the involved languages interact via one embodying the other, and *Hidden InTeraction* (HIT)—there are no any code-level evidence of connection, even implicit ones, between the languages; the interaction is often realized through external data sharing. More detailed descriptions about these interfacing mechanisms can be found in relevant earlier works [28, 29]. In this study, we only consider the interfacing between mainstream languages as in the *TLCO* computation for the same reason (i.e., the reason why we only considered the mainstream languages in computing that metric).

2.4.2 Functionality Domain Identification. To examine the functionality relevance of language use (as one way to justify language selection) in multilingual software, we needed to identify the functionality domain of each project. To that end, we categorized the studied projects based on their functional features through inductive and axial coding analysis [12, 37]. In the inductive coding, we manually labeled a set of randomly sampled projects as per their corresponding non-code artifacts (i.e., README, project description, and topics) and collected a set of codes identified during the manual analysis, hence forming a codebook. Then, in axial coding, we categorized the studied projects by functionality domains according to the codebook derived (i.e., labeling each project with one of the codes that best represents its primary functionality domain).

Specifically, our manual analysis for project functionality domain categorization includes two main steps, *codebook creation* and *project categorization*, as elaborated below.

(1) *Codebook creation.* For our study, the codebook is a set of rules that defines how to assign a specific code to a project. To create this codebook, we randomly selected 1,500 projects, a sample size that was statistically significant at a 95% confidence level (CL) and 5% margin of error (ME). The sampled projects' documents were then analyzed by three of the authors to create and iteratively refine the codebook, addressing disagreements through meetings/discussions until reaching a consensus. Specifically, each project was evaluated by the authors via (1) carefully reading its documents, (2) checking if it fit into an existing category, and (3) creating a new category for those that did not fit into any existing categories.

To create a new category, the authors first defined a label for it and then created a detailed description of the category. To aid in labeling future projects, the authors also summarized the descriptions of projects of that category as typical examples.

The result of this analysis was a codebook that consisted of 20 codes, along with their summary descriptions, which are presented in Table 2. The codebook has two levels of categorization: *level 0*

encodes the codes corresponding to the different layers of the common software stack, ranging from drivers to end-user applications, while *level 1* is coded to cover the diverse kinds of application software.

It is important to note that a well-designed codebook plays a critical role in ensuring consistency and accuracy in coding the projects. This consistency allows for comparing and analyzing the functionality domains of the studied projects.

(2) *Project categorization.* Based on the codebook, the five authors analyzed and coded the entire set of the studied projects. During the coding process, some projects were assigned multiple labels as they were related to different functionality domains. Furthermore, we employed negotiated agreement to address the reliability of coding [9]. As a result, a project was only categorized when all of the authors reached a consensus (through discussion/meetings if disagreement arose initially). A summary of the categorization results, concerning the overall distribution of projects in the SPC dataset over different functionality domains, is presented in Table 3.

Table 2. Codes used to categorize functionality domains of projects

id	Level	Category	Description
1	0	driver	a software component connecting the operating system and hardware devices
2	0	system	the interface between hardware and user applications
3	0	programming	tools for software development (e.g., programming, build)
4	0	middleware	software providing services to applications beyond those available in the OS
5	0	application library	libraries providing data and functions to other client applications
6	0	end-user application	applications providing data and functions to end users
7	1	word process	software for manipulating and designing text
8	1	database	software for creating, editing, and maintaining database files and records
9	1	spreadsheet	software for capturing, displaying, and manipulating data arranged in rows and columns
10	1	multimedia	software for playing, recording or editing audio/video files
11	1	presentation	software for creating a presentation of ideas via texts, images, and/or audios/videos
12	1	enterprise	an integral part of an information system for organizations
13	1	information worker	software for users (individuals) to create and manage information
14	1	communication	software for passing information from one entity to another
15	1	education	software for educational purposes
16	1	simulation	software modeling a real phenomenon with a set of mathematical formulas
17	1	content access	software for accessing content without editing
18	1	application suite	a group of different but inter-related software applications
19	1	email	software for using electronic mail
20	1	engineering/development	integrated software systems supporting development tasks

2.4.3 Association Analysis. As noted earlier, we aim to understand the process and rationale of language selection in multi-language software construction from the perspective of functionality domains. The motivation is that

Table 3. Distribution of projects in SPC over functionality domains, with both levels considered but only notable (>1%) ones shown

Software functionality domain	Percentage of projects in SPC
end-user application	28.67%
application libraries	14.38%
middleware	13.34%
content access	8.75%
engineering/development	6.11%
education	5.16%
database	4.30%
programming	3.82%
multimedia	2.94%
word process	2.82%
system	2.10%
communication	1.60%
email	1.57%
presentation	1.13%
application suites	1.07%

practically different language combinations are chosen typically in relation to what kind/category of functionalities is targeted by the software project. Thus, we compute the association between such categories and language combinations over the studied projects. In addition, to dissect these associations and understand their presence and strength, we further decompose them through in-depth examination of (1) how the functionality domains (FD) are associated with the main languages (MaL) selected—the motivation is that intuitively developers would start language selection with choosing the main languages (i.e., the primary languages to use), which are usually *mainstream programming* languages as we consistently observed in both of our SPC and EVC datasets, (2) how the main languages are further associated with language interfacing mechanisms (LIM)—the motivation is that once the main languages are selected, the next necessary step is to consider how those main languages would interface with other languages (i.e., which interfacing mechanisms are suitable for the chosen main languages)—also, the selection of such interfacing mechanisms is indeed an integral part of the holistic language selection process, and finally (3) how the language interfacing mechanisms are eventually associated with language combinations/selections (LS)—the motivation is that the choice of language interfacing mechanism, given the main languages chosen, immediately affects the choice of other languages in the ultimate language selection. In short, we dissect the overall association between FD and LS through a series of analyses of the association along the following chain: FD→MaL→LIM→LS.

Accordingly, after we obtained/computed the relevant properties (i.e., FD, MaL, LIM, and LS) of each project, we conducted four kinds of association analysis: (1) *Overall association between FD and LS*, (2) *Association between FD and MaL*, (3) *Association between FD and LIM*, and (4) *Association between MaL and LIM*.

Specifically, for any of these kinds of association analysis, we followed an association rule mining process against the given project set. In particular, we identify frequent *if-then* associations which consist of an antecedent (*if*, e.g., FD here) and a consequent (*then*, e.g., LS here), using the Apriori algorithm [41] implemented in the Mlxtend library [42]. This association rule mining process consists of the following three general steps:

(a) *Data formatting*. We represent the inputs as a data frame where one column stores the antecedent variable while the other stores the consequent variable.

(b) *1-hot encoding.* We transform the data frame as follows: first, form all unique data items (i.e., words) in the data frame as a set of size n; then, each cell of the data frame is encoded as n bits by setting each item of the cell as 1 if it is in that set, followed by zero padding.
(c) *Association computation.* With the encoded data frame, we first calculate the *support* for each row, and then obtain the association rules (i.e., the *if-then* association matrix) for the given project set.

For the four kinds of association analysis, in (1)–(3), we consider *FD* as the variable antecedent and others as the variable consequents, while in (4) *MaL* is considered the variable antecedent and *LIM* the variable consequent.

2.5 Evolutionary Characterization (EVC)

As shown in Figure 1, for the per-year project sets as inputs, the EVC works by first computing the per-year SPC results. Then, the EVC examines the evolutionary dynamics of multilingual software based on what the SPC results inform about, in terms of the six statistics listed in Table 4. Each of these statistics indicates an evolutionary characteristic we focused on in our EVC.

Table 4. Evolutionary characteristics we focused on in EVC

No.	Statistics
1	number of unique languages (*language diversity*)
2	percentage (*prevalence*) of multilingual projects
3	language profile size distribution (LPSD)
4	average language code size percentage (ALCP)
5	top language combinations (TLCO)
6	functionality relevance of language use
7	language interfacing mechanisms (LIM)

The rationale of focusing on these particular statistics in the EVC is that the first five offer an intuitive evolutionary *overview* of language use while the last would reveal how the *deeper look* into the correlation between functionality domain/topc and language selection have changed over the years. This is in line with the goal and corresponding specific aims of our study.

Datasets. For EVC, we need multiple yearly datasets each for one of the years within our targeted history—the 10 past years (2010-2019). To that end, we crawled GitHub extensively and obtained 1,000 projects for each year that met all the criteria (i.e, *popularity*, *creation time*, *update time*) and constraints (i.e., *valid topic*, *valid description*) as we set for the repository mining step (Section 2.3). We also aimed to ensure that there is no overlap between any two yearly datasets, for which we considered a project belonging to a specific year only when its last update time is within that year and it is not a fork [33] of another project.

In order to get 1,000 projects per year, we randomly sampled a greater number of projects for that year. Specifically, for each year, we (1) randomly sampled more (than 1,000) projects, (2) dismissed those that failed to satisfy any of the criteria/constraints/requirements stated above, and repeated (1) and (2) until we had 1,000 projects left.

We believe that 10 years represent a reasonably long span for anticipating that multi-language software potentially undergoes noticeable changes in language use and selection. Thus, a span of this length should be suitable for a study with a focus on an evolutionary angle. And 1,000 projects can be considered a sizable dataset for each year. Another reason for choosing 1,000 as the per-year dataset size was that it was more difficult to get a lot more projects from earlier years that met all the criteria/constraints/requirements on GitHub.

2.6 Language Scoping

As a hindsight, we found that there are nearly 300 unique languages used throughout the projects in our study datasets. The sheer total of combinations among this large set of languages turned out to be even more overwhelming. For the ease of presentation and the need to enable in-depth investigation towards our study aims, we have to prioritize and be more focused instead of trying to report results about all the languages and their combinations in one single paper.

Therefore, for the rest of the paper, whenever holistic language profiles/selections are involved (e.g., language profile prevalence for RQ1 and associations with language selections for RQ2), we consider those consisting of some in the *top 50 languages* according to language popularity seen in our datasets—the popularity of a language is measured as the percentage of all of the studied projects that use that language. These 50 languages include both programming languages (e.g., Python and Java) and non-programming languages (e.g., CSS and HTML).

Moreover, given the consistently primary roles played by *programming* languages in all of our studied projects—compared to those of non-programming languages according to the percentage of entire project code size that is attributed to individual languages, the associations computed for RQ2 and RQ3 would be intuitively different between considering all of the 50 languages and considering programming languages only. Thus, we separately examined those associations for the *top 20 programming languages* (among the top 50) only. To determine whether a language is a programming language or not, we referred to the Github Language Stats [16].

For brevity, hereafter, we refer to the top 50 languages when we note that *all languages* are considered, and we refer to the top 20 programming languages when we note that only *programming languages* are considered.

3 RESULTS

In this section, we present and discuss main results and findings obtained according to our study methodology as answers to our research questions.

3.1 RQ1: Language Use/Selection Overview

We start with a basic empirical analysis of the overall language use in multilingual software, including the prevalence of various language selections and the significance of each selected language in a language profile. Specifically, we aimed to understand the language use in terms of the size and composition of language profiles of the studied projects. Again, given that the total number of language selections is an exponential of the number of unique languages, we focus on the combinations of mainstream languages as they have the greatest influence on software quality and functionality as in prior work [4, 43].

As we defined earlier, the overview of language use and selection in multilingual software consists of five measures (Section 2.4.1). We report the measurement results for these measures below.

3.1.1 Language Profile Size. We examine the language profile size in terms of its average and distribution (i.e., *ALPS/LPSD*). Figure 2 depicts the percentage distribution of the 7,113 projects studied (y axis) over different language profile sizes (x axis). Size one was included here to (1) assess the prevalence of multilingual projects overall and (2) make our results more comparable to those of prior peer studies which commonly included single-language projects also when reporting language profile size statistics. We did not differentiate languages of different types (e.g., GPL versus DSL) in order to assess the entire language profiles in multilingual software.

Across these 7,113 projects, 296 unique languages were used, including the well-known languages such as `c`, `c++`, `java`, `python`, and `javascript` and some unusual/much less known ones such as `renderscript`, `hcl`, `processing`, `mako`, `tcl`, `plsql`, `xs`, `gap`, `qmake`, `meson`, `angelscript`, `zenscript`, and `hlsl`—the full list can be found in our artifact. The average language profile size was 4.5 (median 3, stdev 4.8). Excluding a few outlier projects that used more (up to 149) languages, the maximum profile size was 9. The majority (over 75%) of our subject projects used more than 2 languages, while 25% of all used 5 or more languages. There appears to be

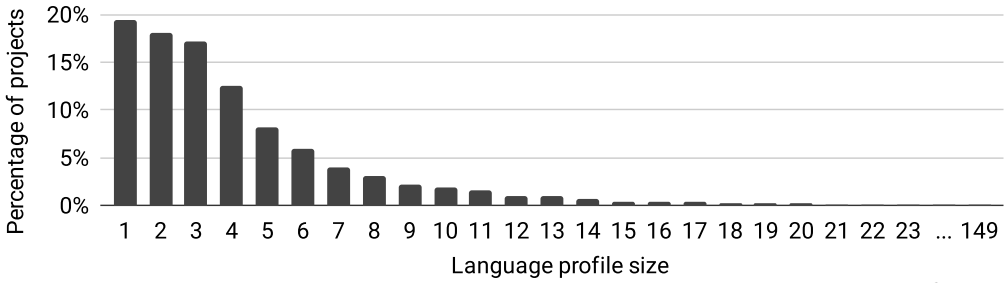

Fig. 2. Distribution of language profile sizes in SPC.

a long tail in the chart because we chose to cover the entire range of profile sizes whereas there were only a tiny/negligible portion of projects that have a profile size greater than 9. Also, the profile sizes are not continuous after 35 (e.g., there were no projects in our datasets with profile sizes of 36, 42, between 44 and 50, between 52 and 64, between 66 and 77, between 79 and 87, or between 89 and 148). And there were no more than 3 projects having a profile size greater than 25. Our inspection revealed that, in those outlier projects (profile size above 9), the majority of the language units are parallel/alternative to each other in terms of functionalities—for example, demonstrating the implementation of the same function (e.g., a Hello-World program) in different languages and providing the capabilities (e.g., syntax highlighting) for (hence in) different languages. That is, most of the languages in these outlier projects do not interact with each other—per our prioritization as described earlier (Section 2.6), these outlier projects were not eventually part of our results for RQ2 or RQ3.

All in all, these numbers show that the studied subjects did not use a very large number of languages in one project, despite the large variety of (296) language choices available. The results are generally close to those from recent peer studies in GitHub (e.g., mean and median language profile size of 6 and 5 in one study [44], and of 5 and 4 in another study [33], respectively). On the other hand, the profile sizes tend to be considerably larger in our study than as reported a decade ago in an average case: according to [13], back during 2000-2005, over 90% of the studied projects (albeit from a different source—SourceForge [35]) used no more than 2 languages. Meanwhile, the largest (outlier) language profiles were much smaller five years or longer ago (e.g., up to 36 [33] or 52 [44] languages in one project) than now (149).

Despite the large number of (296) languages in use, most of the studied multi-language software projects only used 2 to 5 languages (mean=4.5), similar to what were reported over five years ago but noticeably larger than a decade back.

3.1.2 Language Prevalence. After looking at the size of language profiles, we now look inside and across the profiles to examine language distribution by frequency/code size (LDBF/LDBS). Figure 3 shows how frequently different languages were used in the studied multilingual software, where the x axis lists the top-30 most frequently used languages and the y axis indicates the frequency. For instance, `objective-c` was used in 10% (versus `html` in almost 40%) of our projects.

Our results show that languages such as `shell`, `javascript`, `python`, `java`, `c`, and `c++` were the most popularly used languages in the studied multilingual projects. These are not drastically different from what prior studies found about popular languages, whether in single-language projects (e.g., `java` and `javascript` were growing in popularity and `python` was staying popular [13]) or multilingual ones (e.g., `javascript` and `c/c++` were the top two most frequently used languages [33]).

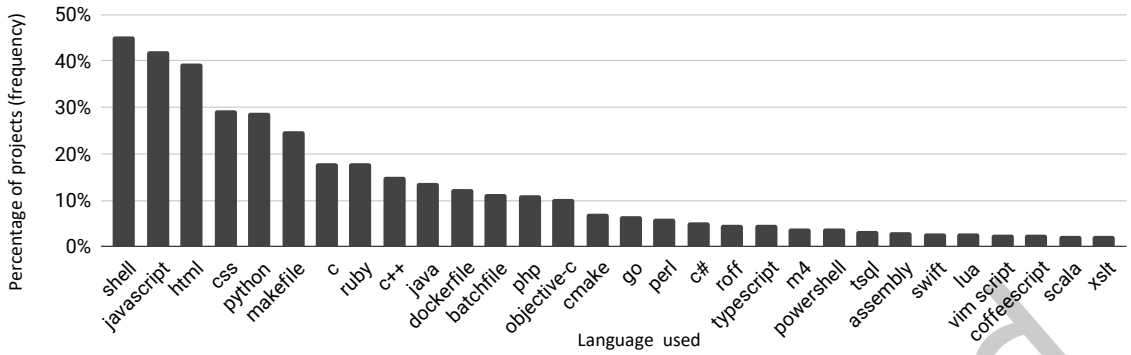

Fig. 3. Language distribution by frequency across all language profiles in SPC.

Yet neither `c` nor `c++` has dropped out the top list as yet despite there were found to decline in popularity over 15 years ago [13]. Also, the greatest frequency of use (as high as 45%) of `shell` over other languages has not been reported before (e.g., previous studies found that the most popular language was `xml` [25] or `c` [4]). This prominent popularity of `shell` in multilingual software may be partly attributed to its good interoperability with other languages [4].

Intuitively, higher (lower) frequency of use of a language may not always indicate a greater (lower) *extent* of use of the language in terms of the language code size. For example, a language may be used commonly but mostly only for writing very little code. Indeed, in contrast to the frequency results of Figure 3, Figure 4 reveals that the most popularly used languages (e.g., `shell` and `javascript`) were not exactly the ones in which most of the software code was written (e.g, `c` and `c++`). Here the y axis shows the percentage of code size attributed to each language (listed on the x axis) across all the 7,113 projects (i.e., treating all these projects as one project). Note that language code sizes here can be largely affected by the nature/type of different languages (e.g., code of a certain number of lines/bytes in higher-level languages would be written in much more lines/bytes in lower-level ones). Nevertheless, the contrast still suggests that neither frequency nor code size alone can fully characterize language use extent in multilingual software. An earlier study [4] found that `c`, `javascript`, and `c++` were top 3 languages in terms of lines of code written in various languages. That is quite similar to our finding here.

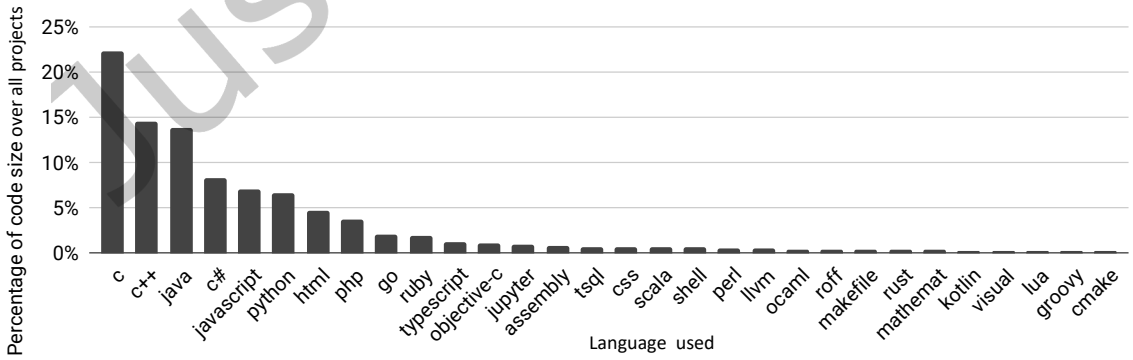

Fig. 4. Language distribution by code size across all language profiles in SPC.

Meanwhile, despite the differences in measurement results between frequency and code size, we also observed that some languages were among the dominant ones in terms of both measures (e.g., `javascript`, `c`, `c++`, `python`, and `java`). These languages are all mainstream programming languages in modern software development in general [36, 43, 50].

> *Popular individual languages used in multilingual systems were not quite different from those in single-language software, but the languages of prevalent use were not necessarily used extensively (in terms of language code size) across the studied projects.*

3.1.3 Language Significance. In light of the results of Figures 3 and 4 on the LDBF/LDBS measures, we extrapolate that languages within a language profile were not evenly significantly used in terms of the code size of each constituent language. To validate this hypothesis, we examine the significance of language use in average language profiles in terms of the average language code size percentage (ALCP). Figure 5 shows what percentage of code (bytes) was written in each language *within a project (i.e., a language profile)* in an average case. The languages are listed on the x axis in the same order as that of Figure 4 to facilitate contrasting between these two results. Thus, as opposed to LDBS measuring the percentage of the total amount of code of all the (7,133) projects in different languages, this figure characterizes the average contribution (in terms of code size) of different languages in multilingual projects—the differences in the total code sizes of different projects were thus considered. For instance, on average 32% of the code in a project was written in `c` among the projects whose language profile included `c`, while among the projects that used `go` over 70% of the code in each project was written in `go` on average. The bases (denominators) from which the averages were computed may not be the same since a language was not necessarily used in every project. Thus, the result should be interpreted together with the frequency of each language among all the (7,133) projects (Figure 3).

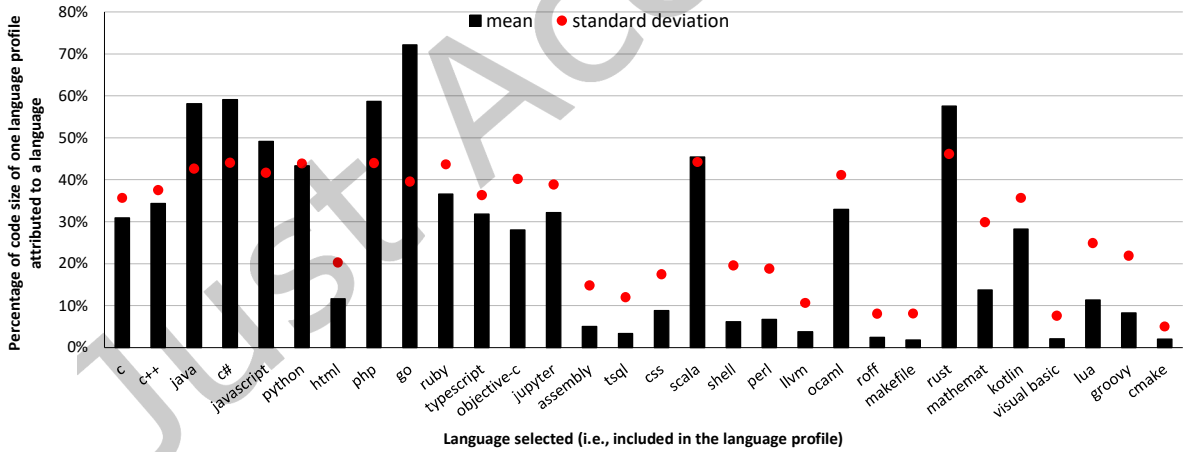

Fig. 5. Average code size percentage attributed to different languages in SPC.

As shown, some languages (e.g., `go`, `rust`, `php`, `java`, and `c#`) contributed to more than half of the code of a project when the project used them. In contrast, certain languages (e.g., `perl`, `html`, `css`, and `cmake`) only contributed minimally to the projects they participated in. Considering the frequency of these languages, the contrast reveals that quite a few languages were widely adopted in multilingual software development but only used very lightly (e.g., `shell`). This can be explained by the intuitive observation that those languages (e.g.,

makefile) were best for certain functionalities that are commonly needed (e.g., project building) but not much code is needed for such functionalities (e.g., a few lines of shell commands for building a project).

> *Languages widely existed which were used frequently but only lightly in the studied multilingual software (e.g.,* shell *and* cmake*), albeit mainstream programming languages tended to be used both frequently and contributed significantly to the software code size (e.g.,* c/c++ *and* javascript*).*

3.1.4 Language Profile Prevalence. Our another perspective into language use/selection concerned the prevalence of language profiles, for which we examine the top (most frequently used) language combinations (TLCO) with *all languages* considered.

Table 5. Top language combinations in SPC

Rank	Language Combination	%Occurrences
1	css-html-javascript	10.4%
2	c-c++-shell	4.8%
3	python-shell	3.6%
4	javascript-typescript	3.1%
5	html-python	2.7%
6	html-ruby	2.4%
7	css-html-javascript-python	2.3%
8	javascript-python	2.2%
9	css-html-javascript-shell	1.9%
10	css-html-javascript-ruby	1.9%
11	c-python	1.9%
12	html-javascript-python	1.8%
13	html-java	1.8%
14	makefile-python	1.6%
15	html-php	1.6%
16	objective-c-ruby	1.5%
17	go-shell	1.5%
18	c++-java-shell	1.5%
19	javascript-php	1.5%
20	css-html-javascript-php	1.4%
21	objective-c-ruby-swift	1.4%
22	javascript-shell	1.4%
23	java-shell	1.4%
24	c-c++-python	1.4%
25	html-javascript-java-c	1.4%
26	c-c++-cmake	1.4%
27	css-javascript-php	1.3%
28	java-javascript	1.3%
29	css-html-javascript-python-shell	1.3%
30	c++-python	1.2%

Table 5 lists the top-30 most frequently appeared language combinations in the language profiles of the multilingual projects studied in SPC. Intuitively, languages `css`, `html`, and `javascript` were most widely used together, as found earlier [44], plausibly because of the popularity of Web and mobile applications in recent years—these languages were indeed common choices for such applications. This suggests that in the era of mobile Internet, front-end applications are a point of interest for most multilingual software developers. In relation to that, combinations of languages `c`, `c++`, `shell`, and `python` were also relatively popular, likely due to the popularity of back-end services which were commonly developed using these languages. Bissyande et al. [4] found that `javascript`, `shell` and `ruby` appeared to be most used together (having the best interoperability) with other languages, which is consistent with what we found here.

In light of other earlier studies, our results also indicate certain shifts of language selection preferences and dominating language profiles in multilingual software construction. For example, `java-xml` and `java-sql` were found to be the most common language pairs [34], which is related to another prior finding that `java` and `xml` files were the top dominating co-evolving code units [25]. And `c` and `perl` were most commonly used together for Web development [13].

> *Top popular language combinations tended to be the combinations of top popular individual languages used in the studied multilingual software projects.*

3.1.5 Language Interfacing Mechanisms. Our final perspective on language use/selection concerned how the languages selected interact with each other. For each project, we used PolyFax [29] to detect all of the interfacing mechanisms among the top language selections (i.e., those among the top-30 list of Table 5). The tool may return *hybrid* mechanisms (i.e., multiple interfacing mechanisms) for a single project if its top language selections do involve different interfacing mechanisms. This is reasonable because a project can indeed involve multiple functionality domains.

As shown in Table 6, eight types of LIMs (including single and hybrid mechanisms) were discovered to be used across the projects in the *SPC* dataset. Implicit interfacing (`IMI`) was the most prevalent among all LIMs, with 69% of projects utilizing it. In contrast, only 7% of the projects used an explicit interfacing via foreign function invocation (`FFI`). This may be due in part to the fact that only a limited number of language pairs support `FFI` between the top languages. Specifically, of the 66 pairs of top 12 languages analyzed in our study, only 30% were found to support interaction via `FFI`. As a result, indirect interaction is frequently the only feasible option for most language pairs, such as `javascript-python`. Additionally, indirect interaction reduces coupling between language components and is relatively straightforward to implement using a mature communication framework. For example, popular languages such as `c`, `python`, `java`, and `ruby` can easily communicate with one another through gRPC [21] in any setting.

Aside from `FFI` and `IMI`, inter-dependence (EBD) is another significant category, mostly found in projects that use the language selection `javascript-css-html`. This trend reflects the common usage of both general-purpose programming languages (GPLs) and domain-specific languages (DSLs) [34].

> *Implicit interfacing (`IMI`) was dominantly used over explicit mechanisms such as `FFI` (e.g., JNI). A significant portion (43.57%) of the projects used hybrid interfacing mechanisms, mostly including `IMI` or `FFI`.*

3.2 RQ2: Functionality Relevance of Language Selection

We examined whether, and quantified how, developer decisions in choosing which languages to use in the studied software projects are associated with their functionality domains in terms of the project topics. Specifically, we first computed an overall associations between functionality domains and language selections. Then, to achieve a

Table 6. Distribution of *LIM* over the SPC dataset

LIM	Percentage
IMI_EBD	22.04%
FFI_IMI	20.14%
IMI	25.72%
EBD	12.17%
FFI	7.04%
HIT	11.49%
FFI_IMI_EBD	1.03%
FFI_EBD	0.36%

deeper understanding of the functionality relevance of language selection, we look into the hidden connections under those associations (i.e., underlying associations) with respect to language interfacing mechanisms, a crucial and *unique* (relative to single-language software) factor in multilingual software construction, hence the total of four kinds of associations as described in Section 2.4.3. By default, we computed these associations with *all languages* considered; to examine the effects of non-programming languages, we additionally examined the underlying associations with non-programming languages excluded (i.e., focusing on programming languages only). Concerning the functionality domains in these association analyses, we consider level-0 and level-1 domains (as listed in Table 2) separately. We further characterized the evolution of such associations, as described later in Section 3.3.

3.2.1 Overall Associations. Tables 7 and 8 list the results of our association analysis on the overall functionality relevance of language selection using the SPC dataset in terms of level-0 and level-1 functionality domains, respectively. The selections shown are part of the top language selection ranking obtained for RQ1. For each pair of (*software domain*, *language combination*), the *support* indicates how frequently the pair appears in the dataset, and the *confidence* indicates the conditional probability of the occurrence of the language selection given the domain. We only report the pairs for which *support*≥1% and *confidence*≥50%. These two thresholds were determined empirically with respect to our SPC dataset: continuing to lower these thresholds would not produce more pairs of at least *weak* association [22] (i.e., *lift*≥1). The strength of association is indicated by the *lift* factor: *lift*<1 indicates the selection and domain are mutually exclusive; *lift*==1 indicates no association; and *lift*>1 indicates the selection and domain are associated, with a greater *lift* value for a stronger association.

Our results revealed that there is a generally quite notable tie between language selection and the functionality domains examined, although the association was relatively weaker (e.g., between `css-html-javascript-php` and `middleware` as well as between `c-c++-cmake` and `end-user application` at level 0 as shown in Table 7) in a few cases than others (e.g., between `css-html-javascript` and `multimedia` at level 1 as shown in Table 8). Overall, with half (3) of the level-0 and half (7) of the level-1 domains, there were strongly associated language selections.

More particularly, when considering the domains at level 0 only, the majority of the positive associations are with *applications* (e.g., `application library` or `end-user application`). Also, for a specific domain, language selections tend to be somewhat diverse—multiple selections are associated with one domain. For example, to develop end-user applications, some may select `php-shell` while others may choose `java-kotlin` or `objective c-ruby-swift`. One possible reason is software of the same domain is naturally developed with different language combinations when in different software ecosystem or on different platforms (e.g., apps on Android often use `java` or `kotlin` and other languages while the apps of same functionality domains on iOS use `objective c` and `swift`

Table 7. Association between level-0 functionality domains and top language combinations in SPC with *all languages* considered

Functionality Domain	Top Language Selection	Support	Confidience	Lift
application library	`css-javascript-php`	2.20%	7.11%	1.51
middleware	`c-c++-python`	1.66%	12.65%	1.41
end-user application	`php-shell`	1.50%	2.98%	1.39
end-user application	`css-html-ruby`	1.82%	3.62%	1.38
application library	`css-html-javascript`	8.80%	28.42%	1.33
end-user application	`css-html-javascript`	1.89%	14.56%	1.29
end-user application	`java-kotlin`	1.02%	2.02%	1.18
application library	`objective c-ruby-swift`	1.72%	5.55%	1.10
middleware	`css-html-javascript-php`	3.59%	27.35%	1.03
middleware	`css-html-javascript-python`	3.38%	25.71%	1.01
end-user application	`c-c++-cmake`	3.97%	7.88%	1.01

Table 8. Association between level-1 functionality domains and top language combinations in SPC with *all languages* considered

Functionality Domain	Top Language Selection	Support	Confidience	Lift
simulation	`go-shell`	1.28%	7.95%	3.24
multimedia	`css-html-javascript`	1.28%	48.00%	2.59
multimedia	`css-html-javascript-php`	1.38%	52.00%	2.19
multimedia	`css-html-javascript-ruby`	1.38%	52.00%	2.18
multimedia	`css-html-javascript-python`	1.28%	48.00%	2.10
simulation	`makefile-python-shell`	1.92%	11.92%	1.90
application suites	`css-html-javascript-ruby`	1.49%	45.16%	1.89
engineering/development	`c-c++-cmake`	3.83%	14.63%	1.86
spreadsheet	`css-html-javascript`	1.70%	30.19%	1.63
spreadsheet	`css-html-javascript-php`	1.92%	33.96%	1.43
spreadsheet	`css-html-javascript-ruby`	1.92%	33.96%	1.42
engineering/development	`html-javascript-typescript`	3.83%	14.63%	1.37
spreadsheet	`css-html-javascript-python`	1.70%	30.19%	1.32
engineering/development	`objective c-ruby-swift`	1.49%	5.69%	1.27
communication	`css-html-javascript-python`	2.66%	28.74%	1.26
engineering/development	`html-javascript-python`	2.56%	9.76%	1.19
email	`css-html-javascript-php`	3.30%	25.41%	1.07
application suites	`css-html-javascript-python-shell`	1.28%	38.71%	1.07
communication	`css-html-javascript-ruby`	2.34%	25.29%	1.06
email	`c-c++-cmake`	1.06%	8.20%	1.04

more often). For the other (i.e., non-application) domain, `middleware`, developers tend to select `c-c++-python` or languages (e.g., `php` and `python`) combined with `css-html-javascript`.

Regarding the domains at level 1 (i.e., specific kinds of end-user applications), `multimedia` is strongly associated with languages combined with `css-html-javascript`—this kind of language selection is also popularly seen in constructing other kinds of end-user applications, such as `spreadsheet`, `communication`, and `email`. These associations indicate that the language selection `css-html-javascript` is most widely used in developing application software. Referring to the results in Table 7, we note that `end-user application` is strongly associated with `css-html-javascript`, indicating a consistency in the association analysis results between the two levels of functionality domains we examined.

On the other hand, between the two functionality domain levels, the associations at level-1 are generally stronger. This is because when we look at the higher-level (level-0) domains, the greater diversity of language selections within each domain (compared to the lesser diversity within each domain at the lower level, i.e., level 1) tend to make the association with a particular language selection relatively weaker.

> *Language selection was considerably relevant to the functionality domain in multilingual software construction, and some language combinations were more strongly associated with certain functionality domains than others.*

3.2.2 Underlying Associations. To further mine the hidden connections (as potential interpretations) underlying the overall associations identified in Section 3.2.1, we progressively computed associations between *FD* and *MaL*, then between *MaL* and *LIM*, and finally between *LIM* and *LS*. In this way, we may potentially understand why functionality domains are associated with some specific language selections along this association chain. We then separately look at this association chain with programming languages considered only to further assess how non-programming languages may have impacted the functionality relevance of language selection. Given the generally stronger associations with functionality domains at level 1, to avoid verbosity of this paper we will only report results on the association chain with respect to level-1 domains.

With all languages considered. Table 9 shows the associations between *FD* and *MaL*, Table 10 shows the associations between *MaL* and *LIM*, and Table 11 shows the associations between *LIM* and *SL*.

As summarized in Table 9, most of (8 out of the total of 14) the level-1 functionality domains were found positively associated with one or more main languages. For instance, the `simulation` domain is associated with four main languages `shell`, `go`, `java`, and `python`, while the *engineering/development* domain is associated with `c++`, `c`, `php`, and `c#`. That is, the main languages associated with a functionality domain can also be diverse, consistent with the diversity in this regard seen in the overall associations between functionality domains and language selections.

Further along the association chain, we found substantive associations between the main language and language interfacing mechanisms, as shown in Table 10. Specifically, main languages `c` and `c++` are both strongly associated with the interfacing mechanisms involving FFI (i.e., FFI, FFI_IMI, and FFI_IMI_EBD). One reason is that current mainstream languages all have FFI interfacing with `c/c++` [28]. For example, `python` interacts with `c` through Python extension, `java` interacts with `c` through JNI, and `go` interacts with `c` via cgo. As a result, most of the main languages (which are also mainstream languages) listed in the table are found associated with FFI or interfacing mechanisms involving FFI.

Finally, between language interfacing mechanisms and language selections, we also found generally quite strong associations, as listed in Table 11. For instance, FFI is associated with `c-c++`, while FFI_IMI is associated with `c-c++-python`, `c-c++-objective c`, `c-python`, and so on. These associations are also consistent with the results of Table 10, where the main languages are always part of language selections shown in Table 11.

Overall, along the holistic association chain, the results on overall associations are generally quite well aligned with, hence explained/justified by, the corresponding results on underlying associations. To illustrate, let us consider the `simulation` domain. From Table 8, we see that one of strongly associated language selections with

this domain is `makefile-python-shell`. To understand how this overall association came about rationally, let us follow the association chain as follows. First, from Table 9, we can see the most strongly associated main language with `simulation` applications is `shell`. Then, from Table 10, we notice that the interfacing mechanism most strongly associated with `shell` is IMI. Finally, as shown in Table 10, we see that IMI is strongly associated with the language selection `makefile-python-shell`. In this way, the overall association between the `simulation` domain and the `makefile-python-shell` selection is justified by the illustrated chain of underlying associations.

More generally, from a developer's perspective, given a functionality domain targeted along with a specific software ecosystem concerned with, selecting the main language can be the crucial first step during the multilingual software construction process. For example, to develop a `communication` application, `python` can be a good choice as the main language, as shown in Table 9. Then, the developer can try to find languages that work well with the main language to satisfy the development requirements. In this step, the language interfacing mechanism is a primary decision factor since the interfacing is knowingly associated with the quality (e.g., security in terms of vulnerability proneness) of multilingual software [28]. Following the example, if the developer chose the interfacing of IMI to construct the software, then according to Table 11, the language selection `css-html-javascript-python` can be a good choice since it is widely used in this domain. Moreover, the domain of `communication` is also associated with `css-html-javascript-python` as shown in Table 8. Alternatively, the developer could choose other language selections associated with IMI that include `python` (e.g., `c++-python`).

Table 9. Association between (level-1) functionality domains and main languages in SPC with *all languages* considered

Functionality Domain	Main Language	Support	Confidience	Lift
simulation	shell	1.32%	7.80%	2.72
multimedia	javascript	1.03%	47.37%	2.13
end user application	javascript	2.12%	38.95%	1.75
simulation	go	1.49%	8.81%	1.60
email	c	1.09%	9.22%	1.58
communication	python	1.32%	13.22%	1.51
engineering/development	c++	2.80%	10.10%	1.51
spreadsheet	javascript	1.89%	33.33%	1.50
engineering/development	c	2.35%	8.45%	1.45
engineering/development	php	2.75%	9.90%	1.20
word process	java	1.03%	12.41%	1.20
engineering/development	c#	1.26%	4.54%	1.18
communication	java	1.14%	11.49%	1.11
email	php	1.03%	8.74%	1.06
simulation	java	1.83%	10.85%	1.05
simulation	python	1.49%	8.81%	1.01

With only *programming* languages considered. Table 12 shows associations between *FD* and *MaL*, Table 13 shows associations between *MaL* and *LIM*, and Table 14 shows associations between *LIM* and *SL*, with non-programming languages dismissed.

Like those with all languages considered, the underlying associations computed with only *programming* languages considered are similarly strong, supporting generally similar conclusions as well. One major difference is that EBD as an interfacing mechanism is now absent in the underlying associations of Table 14. The reason is

Table 10. Association between main languages and language interfacing mechanisms in SPC with *all languages* considered

Main Language	Language Interfacing Type	Support	Confidience	Lift
css	EBD	1.31%	55.56%	4.56
html	EBD	1.33%	34.74%	2.85
c++	FFI	1.59%	19.90%	2.82
shell	IMI	1.13%	69.14%	2.69
c	FFI	1.49%	18.18%	2.58
php	HIT	1.67%	29.64%	2.57
java	FFI	1.37%	15.63%	2.22
javascript	EBD	6.06%	26.76%	2.20
ruby	HIT	1.41%	23.49%	2.04
go	IMI	1.93%	48.98%	1.91
html	FFI_EBD	1.39%	36.32%	1.85
c	FFI_IMI	7.65%	93.37%	1.77
c++	FFI_IMI	7.29%	91.18%	1.73
python	IMI	5.23%	41.47%	1.61
css	IMI_EBD	2.27%	96.58%	1.61
typescript	IMI_EBD	1.93%	96.00%	1.60
javascript	IMI_EBD	21.60%	95.38%	1.59
shell	FFI_IMI	1.31%	80.25%	1.52
java	FFI_IMI	6.96%	79.54%	1.50
python	FFI	1.31%	10.37%	1.47
go	FFI_IMI	3.00%	76.02%	1.44
html	IMI_EBD	3.28%	85.79%	1.43
shell	IMI_EBD	1.37%	83.95%	1.40
javascript	FFI_EBD	6.20%	27.38%	1.40
ruby	IMI	2.09%	34.90%	1.36
python	FFI_IMI	9.00%	71.29%	1.35
ruby	IMI_EBD	4.27%	71.14%	1.19
javascript	FFI_IMI_EBD	22.34%	98.67%	1.12
php	IMI_EBD	3.72%	66.07%	1.10
css	FFI_IMI_EBD	2.29%	97.44%	1.10
c#	IMI_EBD	1.13%	65.88%	1.10
typescript	FFI_IMI_EBD	1.95%	97.00%	1.10
shell	FFI_IMI_EBD	1.57%	96.30%	1.09
c	FFI_IMI_EBD	7.87%	96.07%	1.09
html	FFI_IMI_EBD	3.64%	95.26%	1.08
c++	FFI_IMI_EBD	7.59%	94.96%	1.07
c#	FFI_IMI_EBD	1.59%	92.94%	1.05
java	FFI_IMI_EBD	8.11%	92.64%	1.05
c++	FFI_EBD	1.63%	20.40%	1.04
python	FFI_IMI_EBD	11.25%	89.15%	1.01

Table 11. Association between language interfacing mechanisms and language selections in SPC with *all languages* considered

Language Interfacing Type	Language Selection	Support	Confidience	Lift
FFI	`c-c++`	1.34%	20.69%	4.97
IMI	`javascript-shell`	1.59%	6.43%	3.54
EBD	`html-javascript`	2.54%	17.40%	3.08
EBD	`css-javascript`	1.26%	8.60%	3.05
IMI	`makefile-python-shell`	1.12%	4.51%	2.83
EBD	`css-html-javascript`	4.50%	30.78%	2.76
EBD	`javascript-typescript`	1.17%	8.03%	2.66
FFI_IMI	`c-c++-python`	2.51%	5.30%	2.09
FFI_IMI	`c-c++-objective c`	1.26%	2.65%	2.06
FFI_IMI	`c-python`	2.23%	4.71%	2.03
IMI	`java-shell`	1.12%	4.51%	1.92
FFI_IMI	`makefile-python-shell`	1.42%	3.01%	1.89
FFI_EBD	`css-html-javascript`	4.50%	21.07%	1.89
FFI_IMI	`javascript-shell`	1.62%	3.42%	1.88
FFI_IMI	`c++-python`	1.34%	2.83%	1.88
FFI_EBD	`javascript-typescript`	1.17%	5.50%	1.82
IMI	`javascript-typescript`	1.28%	5.19%	1.72
FFI_IMI	`java-javascript`	1.42%	3.01%	1.52
IMI_EBD	`css-html-javascript-ruby`	2.79%	4.32%	1.47
IMI	`css-html-javascript-python`	2.07%	3.20%	1.40

that this interfacing mechanism is commonly applicable between non-programming languages (e.g., `CSS` and `html`)—thus, the relevant data samples were filtered out prior to the association analysis.

> *With either all or only programming languages considered, underlying associations along the chain of {FD→MaL, MaL→LIM, LIM→SL} are strong and consistent with, hence explaining/justifying, the respective overall associations.*

3.2.3 Case Studies. To gain more insights into the association, we randomly chose 10 popular (i.e., having received 1800+ stars) and mature (i.e., having been 6+ years old) projects from all `Music Software` projects in our SPC dataset and manually gained understanding about the functionalities of modules in different languages. We chose `Music Software` as a subcategory in `multimedia`, a random sample of the functionality domains represented in our study datasets, which is also one of the major level-1 domains (Table 2) that covers a non-trivial portion of our sample projects (Table 3). This software functionality category has also seen projects that use a variety of languages and language combinations. Table 15 shows these projects and the top languages used in each project in terms of the size of code written in each language. We found that `javascript` and `python` were used most frequently in these projects and selected together in three cases (boldfaced). Meanwhile, not every music software used them and most of the projects did not select both. This is consistent with our result indicating that the association between `Music Software` and `javascript-python` is relatively weak (hence not listed in Table 8).

Looking further into the three cases, we found that the core functionalities (i.e., accounting for 85%+ of project code) are implemented in `python`, consistently for features such as *resource (e.g., songs and lyrics) search* and

Table 12. Association between (level-1) functionality domains and main languages in SPC with only *programming languages* considered

Functionality Domain	Main Languages	Support	Confidience	Lift
simulation	shell	1.32%	7.80%	2.72
multimedia	javascript	1.03%	47.37%	2.13
end user application	javascript	2.12%	38.95%	1.75
simulation	go	1.49%	8.81%	1.60
email	c	1.09%	9.22%	1.58
communication	python	1.32%	13.22%	1.51
engineering/development	c++	2.80%	10.10%	1.51
spreadsheet	javascript	1.89%	33.33%	1.50
engineering/development	c	2.35%	8.45%	1.45
engineering/development	php	2.75%	9.90%	1.20
word process	java	1.03%	12.41%	1.20
engineering/development	c#	1.26%	4.54%	1.18
communication	java	1.14%	11.49%	1.11
email	php	1.03%	8.74%	1.06
simulation	java	1.83%	10.85%	1.05
simulation	python	1.49%	8.81%	1.01

Table 13. Association between main languages and language interfacing mechanisms in SPC with only *programming languages* considered

Main Language	Language Interfacing Types	Support	Confidience	Lift
c++	FFI	1.59%	19.90%	2.82
shell	IMI	1.13%	69.14%	2.69
c	FFI	1.49%	18.18%	2.58
php	HIT	1.67%	29.64%	2.57
java	FFI	1.37%	15.63%	2.22
ruby	HIT	1.41%	23.49%	2.04
go	IMI	1.93%	48.98%	1.91
c	FFI_IMI	7.65%	93.37%	1.77
c++	FFI_IMI	7.29%	91.18%	1.73
python	IMI	5.23%	41.47%	1.61
shell	FFI_IMI	1.31%	80.25%	1.52
java	FFI_IMI	6.96%	79.54%	1.50
python	FFI	1.31%	10.37%	1.47
go	FFI_IMI	3.00%	76.02%	1.44
ruby	IMI	2.09%	34.90%	1.36
python	FFI_IMI	9.00%	71.29%	1.35

Table 14. Association between language interfacing mechanisms and language selections in SPC with only *programming languages* considered

Language Interfacing Type	Language Selection	Support	Confidience	Lift
HIT	`ruby-swift`	1.81%	21.67%	8.85
FFI	`c-c++`	1.78%	20.65%	7.81
IMI	`c-shell`	1.22%	4.37%	2.49
IMI	`go-shell`	2.28%	8.14%	2.40
IMI	`ruby-shell`	2.50%	8.94%	2.35
IMI	`python-shell`	5.59%	19.96%	2.31
IMI	`php-shell`	1.47%	5.26%	2.28
FFI_IMI	`c-c++-python`	1.50%	2.46%	1.63
FFI_IMI	`c-c++-java-python-shell`	1.53%	2.50%	1.63
FFI_IMI	`c-c++-python-shell`	5.90%	9.64%	1.59
FFI_IMI	`c-c++-shell`	3.62%	5.91%	1.59
FFI_IMI	`c-c++`	2.50%	4.09%	1.55
FFI_IMI	`c-python-shell`	1.70%	2.77%	1.53
IMI	`java-shell`	1.53%	5.46%	1.51
FFI_IMI	`c-c++-javascript-python-shell`	2.17%	3.55%	1.50
HIT	`python-shell`	1.06%	12.67%	1.47
FFI	`python-shell`	1.08%	12.58%	1.46
FFI_IMI	`java-c-shell`	3.17%	5.18%	1.43
IMI	`javascript-shell`	2.67%	9.53%	1.22
FFI_IMI	`c++-ruby-shell`	2.61%	4.27%	1.12

downloads. The remaining functionalities are mostly implemented in `javascript`, consistently for features such as *music play* and *metadata viewing* via web browsing, as a Web UI or a Web plug-in. This finding suggests that the association we observed is justifiable: the language selection appeared to be justified by the features the selected languages can best offer together for the targeted functionalities.

In relation to prior peer studies, although the functionality relevance of language choices was looked at before [4], prior studies focused on how functionality domains were connected to *individual* languages rather than the selection of multiple languages as a whole. We also recall that in practice there are usually many factors (e.g., language features, software functional requirements, and developer expertise/preferences) that may affect the eventual choices of language profiles. In our study, we did not attempt to fully answer the question of why certain language combinations are chosen over others; instead, our goal is to shed light on the justifiable relevance of functionality categories of multilingual software to its language profile.

The association between language selection and functionality domain was justifiable by the collective features of selected languages better facilitating the functionality requirements.

3.3 RQ3: Evolution of Multilingual Systems

In this section, we present the results of our Evolutionary Characterization (EVC) study, reporting the evolutionary characteristics on both overall language use/selection and their functionality relevance as outlined in Table 4. We first look at the evolution of language use/selection (for which an overall characterization was given for RQ1),

Table 15. Case studies on the functionality relevance of language selection: 10 cases for the `Music Software` domain

Project	#Stars:Age(#years)	Top Languages
iScript	4738:6	`python javascript`
KodExplorer	4715:7	`php html javascript css`
Soundnode	4660:6	`javascript html css`
Cmus	3853:8	`c c++ shell python`
Headphones	2937:9	`python html javascript css`
Lmms	4181:6	`c++ objective-c cmake html`
Scdl	1800:6	`python`
Tomahawk	2644:10	`c++ cmake javascript`
Vexflow	2760:10	`javascript html shell`
Beets	9525:10	`python javascript shell`

followed by examining the evolution of the association between language selection and functionality domains (for which an overall characterization was given for RQ2).

3.3.1 Evolution of Language Use/Selection. We found that the diversity of languages had grown continuously, as depicted in Figure 6. In less than a decade, the number of unique languages used across our 10 yearly datasets nearly quadrupled: from 35 in 2010 to 138 in 2019. This result clearly indicated that multilingual software developers had increasing flexibilities and choices in language use and selection for system construction. The monotonic nature of the trend that has sustained for 10 years projects a likely continuing growth of language diversity (at least in the open-source community).

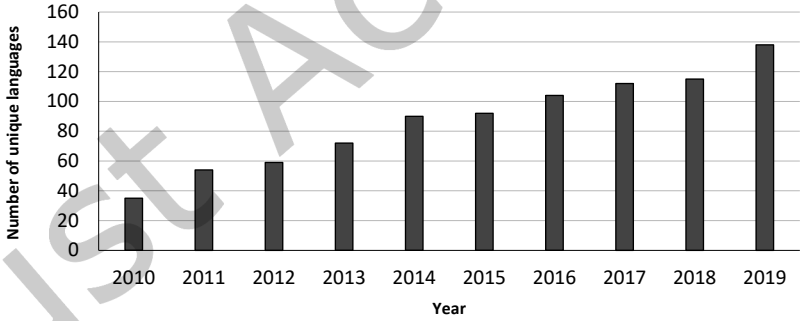

Fig. 6. Evolution of language diversity (#unique languages used year to year).

Plausibly a result of the growing language choices, we also observed a steady uptrend in the prevalence of multilingual software construction. Figure 7 delineates the percentage of projects that were developed in 2 or more languages in each of our 10 yearly EVC datasets. For instance, in 2010, 41% of the 1,000 sampled projects were written in multiple languages, while in 2019 this percentage grew to 74%. This finding resonates with results from a prior study [13] that showed a similar growth of multilingual software prevalence: the percentage of projects using multiple languages increased from 10% to 35% from year 2000 to year 2005 (albeit on a different open-source software repository portal SourceForge). Put together, that earlier study and ours here revealed constant growth of the prevalence of the multilingual software construction practice. By now, multilingual

construction has become a definite norm and clearly dominated over single-language development in modern software practice, so far as our studied projects were concerned with.

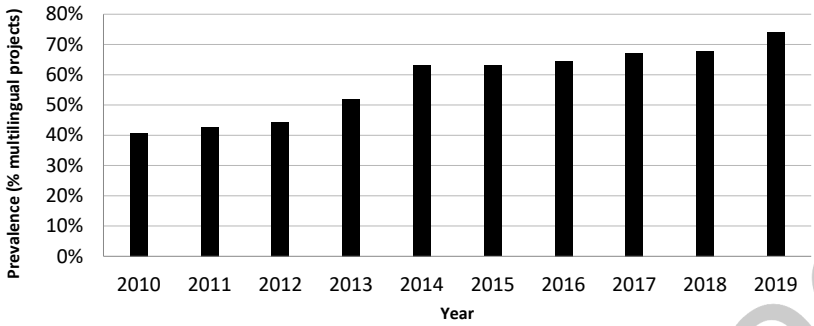

Fig. 7. Evolution of multilingual software prevalence (% multilingual projects year to year).

In the 10-year span examined, both the language diversity and multilingual software prevalence grew steadily, and multilingual construction has become a dominating norm in modern software development as seen in the studied projects.

Complementary to the overall growth of language diversity and multilingual software prevalence, we now examine what has changed in individual projects (language profiles). Figure 8 shows the evolution of language profile size distribution among the studied subjects, where various sizes are encoded with gradual color depth and the height of each single-color bar indicates the percentage of projects having the associated profile size. The results show 3, 4, and 5 as sizes of the fastest growing popularity, consistent with our overall profile size statistics (e.g., mean 4.5); meanwhile, the trend also indicates general decreases in projects using less languages.

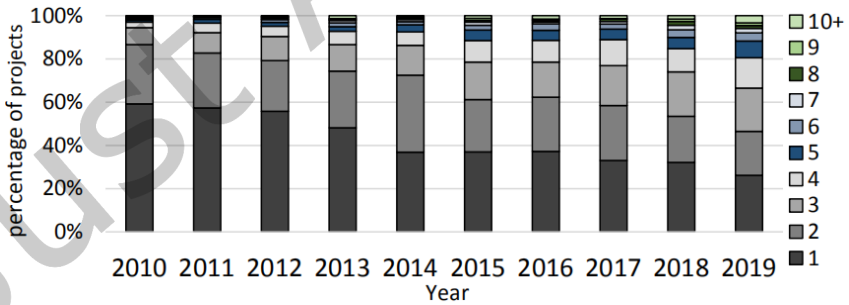

Fig. 8. Evolution of language profile size distribution.

Looking into the profile composition, we found that the top languages used (e.g., `java`, `python`, and `javascript`) remained almost constant over time. Figure 9 shows the percentage of projects (y axis) from each of the 10 years (x axis) in our EVC study that used each of these top languages. Our results show that the use (profile inclusion) frequency of almost all of the contemporary mainstream languages was fairly stable (or slightly up in a few cases such as `javascript` and `java`). One exception was `ruby`, whose popularity dropped considerably (by 15% in

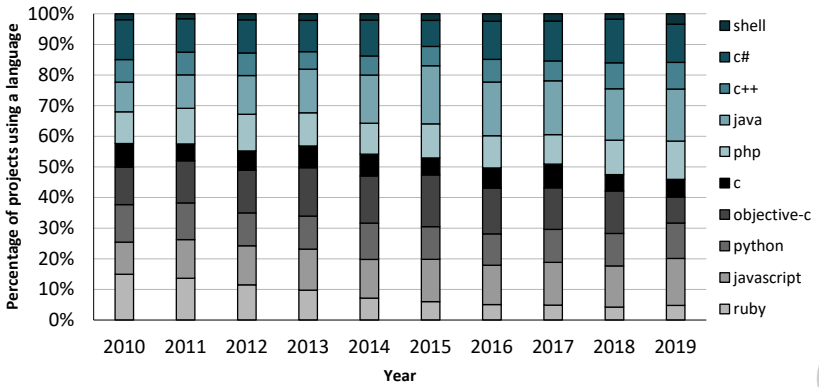

Fig. 9. Evolution of top individual languages used.

terms of the portion of projects using it). Nevertheless, the languages that are mostly known as highly popular sustained a strong and unwavering presence in the language profiles of the studied multilingual systems.

In terms of language combinations in the studied projects (i.e., language profiles), we found the list of top ones was also pretty stable, yet the order has had clear shifts, as Figure 10 shows. The most noticeable were the growth in `python shell` and `python c/c++`, and the reduction in `javascript ruby` and `c objective-c`. These trends can be explained by the evolution of the popularity of constituent languages (e.g., uptrend in `python` and downtrend in `ruby`) and the interoperability between languages (e.g., friendly interface of `shell` with other languages). Again, the top/mainstream individual languages included in these top combinations did not change much over the years and were consistent with the results of Figure 9.

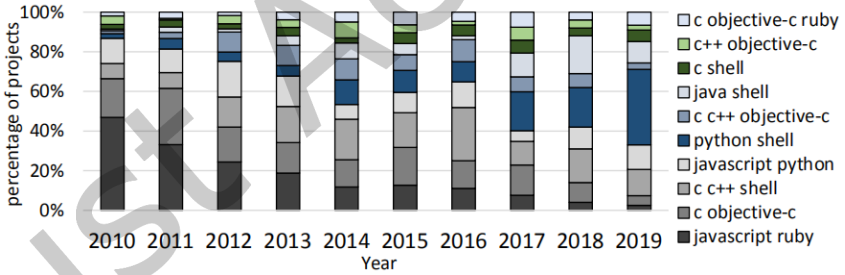

Fig. 10. Evolution of top (10) language combinations.

Over the 10 past years, increasingly more projects used 3–5 languages; both the list of top individual languages used and the list of top language combinations selected were stable (even more so with the former list), albeit the ranking of top combinations has shifted considerably.

3.3.2 Evolution of Language Interfacing Mechanisms. Figure 11 depicts the distribution of multilingual software over various kinds of language interfacing mechanisms for each of the yearly datasets in our EVC characterization. Together, these yearly results present a view of how this distribution has evolved from the year of 2010 through 2019. Overall, there has been a clear increase in the diversity of interfacing mechanisms used in multilingual

software construction. The number of different kinds of mechanisms was 4 in 2010, which has gone up to 8 by 2019. And the diversity increased *monotonically* over the years, reflecting the growing complexity of multilingual software construction. For instance, in earlier years (e.g., 2010 and 2011), most (around 70%) of the projects in each year primarily used relatively straightforward interfacing mechanisms such as FFI, IMI, or both. This growth appears to be aligned with the increase in the number of multilingual projects that have increasingly greater language profile sizes shown in Figure 8—with a greater number and variety of languages being selected in constructing a single multi-language software project, there are naturally increasingly more and diverse language interfacing mechanisms adopted in the construction.

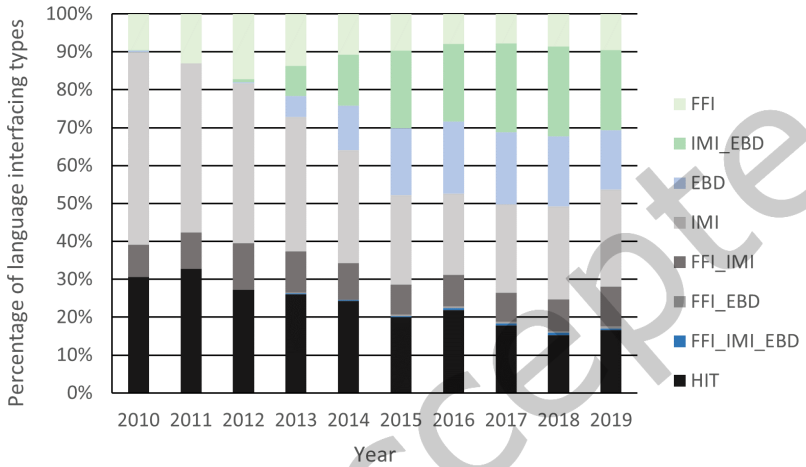

Fig. 11. Evolution of language interfacing type distribution.

On the other hand, we observed some consistency/stability in the use of interfacing mechanisms during this 10-year span. For instance, IMI has always been the most popular interfacing mechanism used in multilingual construction, either exclusively or in conjunction with other mechanisms such as FFI and/or EBD. In fact, in any given year, more than 60% of the projects used IMI. This constant dominance can be justified by the merit of IMI in reducing the coupling between different language units (because IMI features indirect/implicit interaction which implies low coupling), as well as facilitating software extensibility in terms of including additional language units for new functional features (again because of the flexibility IMI offers). Such merits are particularly significant as modern software construction has gone increasingly multilingual and polyglot-ism has become the norm [28]. Considering the evolution of language combinations as shown in Table 10, we observed that language selections such as `python-shell`, `javascript-python`, and `java-shell` have been increasingly dominant. This trend helps justify the growing dominance of IMI in the evolution of language interfacing mechanism distribution as shown in Figure 11 because in these language selections the languages are commonly interfaced via the IMI mechanism.

> *Over the 10 years examined, increasingly diverse language interfacing mechanisms have been used in multilingual construction. On the other hand, IMI has been consistently dominant over interfacing mechanisms.*

3.3.3 Evolution of Functionality Relevance of Language Use/Selection . As a result of the random sampling process underlying our collection of the EVC dataset, the set of functionality domains as categorized with our approach (Section 2.4.2) varied from one year to another. Also, from some of the domains, there may not be any positively associated language selection for any of the years studied. Thus, we focus on functionality domains in common

across the 10 per-year domain sets that have at least one positively associated language selection for at least one of the 10 years and refer to such domains as common domains.

With all languages considered. When all languages were considered, there were five common functionality domains, as shown in (the leftmost column of) Figure 12. To facilitate visual pattern discovery, we visualize the association evolution for these common domains as follows: (1) the legend shows the set of languages most frequently included in the top language selections in the EVC dataset; (2) for each domain and year, these languages are mapped to fixed colors and cell positions[4] to help observe evolution patterns, and each row of cells represents one language selection. For instance, in 2014 the most frequently adopted language combination for `Application library` was `c-c++-objective c-ruby`, while one year later for the same functionality domain the dominating language selection was `css-html-javascript-shell`. As a partial elaboration of the visualization, Table 16 lists one of the language selections associated with each of the five domains (first row) for each year (first column), to illustrate how to observe the evolution pattern. As in the visualization, the order of languages in each language selection is not relevant.

Overall, our evolutionary characterization on the association between language selection and functionality domain indicates that the association has shifted over the 10 past years. For each individual functionality domain, the language selections associated with it changed constantly from year to year. Meanwhile, no selection was always associated with a domain, although some associations were relatively stabler than others. For instance, `c-objective c-ruby` was associated with `Middleware` only in 2011, the association of `css-javascript-ruby` with `End-user Application` stayed the same for three years—2014, 2015, and 2016, while `objective c-ruby` has remained associated with `Engineering/development` for four consecutive years.

On the other hand, looking at subsets of selected languages, we observed that there appeared to be some stable members in the language selections associated with each domain. For instance, `javascript` was selected in `End-user Application` projects in all the 10 years, although the other languages it joined changed (e.g., `python` in 2010 but `ruby` in 2012 and 2013). Another instance is `c` selected in `Engineering/development` projects for six years. We noticed that these stable members are individual languages known to be widely used in the respective functionality domains —for example, `java` and `objective-c` for mobile (Android and iOS, respectively) apps, as well as those that are recognized for their high portability and user-friendliness (i.e., ease to program with) .

Table 16. One example language selection associated with each of the functionality domains (first row) shared among the yearly datasets in EVC for each of the 10 years (first column), with *all languages* considered—serving as a partial elaboration of the visualization of Figure 12

Year	Middleware	Application library	End-user Application	Education	Engineering/development
2010	--	c-c++-python-shell	javascript-python	javascript-ruby	c-c++-java
2011	c-objective c-ruby	c-c++-objective c-shell	javascript-python-ruby	javascript-ruby	c-c++-java
2012	c-c++-objective c	c-c++-objective c-shell	javascript-ruby	coffeescript-javascript-ruby	c-c++-shell
2013	c-c++-python-shell	c-c++-objective c	javascript-ruby	javascript-ruby-shell	assembly-c-c++
2014	css-javascript-ruby-shell	c-c++-objective c-ruby	css-javascript-ruby	css-javascript-ruby	c-c++-python
2015	css-html-javascript	css-html-javascript-shell	css-javascript-ruby	css-html-javascript	objective c-ruby
2016	css-html-javascript-java	--	css-javascript-ruby	css-html-javascript	objective c-ruby
2017	css-html-javascript-php	objective c-ruby-swift	css-javascript-php	css-html-javascript-ruby	objective c-ruby
2018	css-html-javascript-php	--	css-javascript-php	css-html-javascript-ruby	objective c-ruby
2019	--	--	html-javascript-python	css-html-javascript	c-c++-python

[4]This is attempted at best effort but cannot be always enforced due to (1) the considerable variations in language selections associated with different domains across different years and (2) the large number of individual languages that need to be presented—thus, the box enclosing the language selections for each domain at each year would be too wide if we strictly enforce mapping each language to a fixed cell position.

Fig. 12. Evolution of the associations between language selections and common functionality domains over the 10-year span studied, with *all languages* considered.

Over time, language selections were less stable than individual languages in association with functionality domains of the studied multilingual projects, although some selections were more stable for certain domains than others.

With only programming languages considered. To further understand the evolution of the associations between functionality domain and language selection, we examined the potential effects of non-programming languages on the evolutionary dynamics. To that end, we characterized the evolution of those associations with only programming languages considered, as visualized in Figure 13 following the same format as Figure 12.

With respect to what a *common domain* means as defined above, we found six common domains across the 10 yearly sets of sample projects when we only considered programming languages in language selections. In addition to the five observed when all languages were considered (i.e., in Figure 12), another common domain, `Content access`, also has fairly strong associations with some (programming) language selections in most of the (8 out of 10) years.

Similar to the patterns shown in Figure 12, given a specific functionality domain, although language selections also constantly changed over the years, some individual/constituent languages are stably present in the language selections throughout the evolution. For example, in `Application library` projects, `c++` and `shell` were always selected in all the nine years in which any strong association was successfully found. As another example, in the `Engineering/development` domain, the sampled multi-language projects constantly selected `c` and `c++` for every single year during the studied 10-year span. Also, more holistically, there was also at least one stably associated language selection (as opposed to individual languages) for every one of the six common domains. In particular, with both `Middleware` and `Application library`, `shell-python-c++-c` was strongly associated for 6 years (2010, 2011, 2013, 2015, 2016, and 2019); with `End-user application`, `javascript-ruby-shell` was strongly associated for 5 years (2010, 2012, 2013, 2015, and 2018); with `Education`, `javascript-ruby-python` was strongly associated for 3 years (2010, 2011, and 2012); with `Content access`, `shell-python-javascript-c++`

was strongly associated for 6 years (2012, 2013, 2014, 2015, 2017, and 2019); and with `Engineering/development`, `shell-c++-python` was strong associated for 5 years (2015, 2016, 2017, 2018, and 2019).

These statistics revealed that developers did seem to have preferred particular languages and language selections for constructing multi-language software projects in a particular functionality domain.

On the other hand, for each of the (five) domains shared between Figure 13 and Figure 12, the associated language selections changed in most cases; in fact, the language selections in the former (Figure 13) were not often a subset of the respective ones in the latter (Figure 12). The reason is that the language selections changed for each project after the elimination of non-programming languages, causing variations in the language selection distributions hence the positive/strong associations. The overall patterns and evolutionary characteristics, however, are not quite different between these two figures.

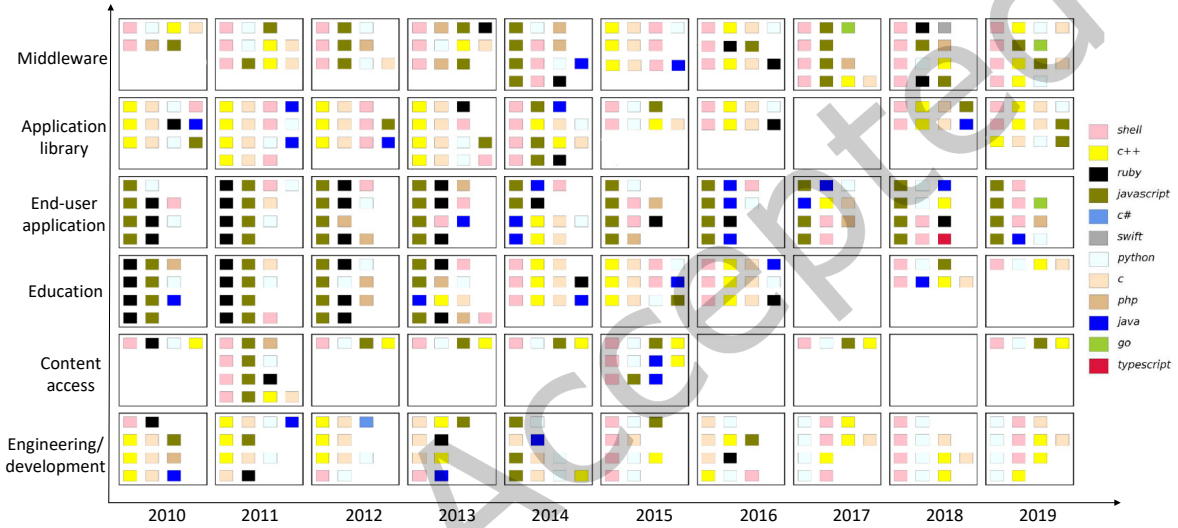

Fig. 13. Evolution of the associations between language selections and common functionality domains over the 10-year span studied, with only *programming languages* considered.

> *When only considering programming languages, the associations between functionality domains and language selections evolved generally similarly (i.e., in terms of main evolutionary characteristics regarding what changed constantly and what were more stable) to those with all languages considered.*

3.3.4 Evolution of Functionality Relevance of Language Interfacing. Now that we have looked at the evolution of language interfacing mechanisms (Figure 11) and the evolution of how functionality domains are associated with language selections (Figure 12), it is naturally helpful to see next how the associations between functionality domains and language interfacing mechanisms have evolved. The rationale is twofold. First, these mechanisms are an essential, unique/defining (relative to single-language software) aspect of multilingual software construction. Second, the interfacing mechanism of a language selection is clearly a key underlying property of that language selection.

Again we focus on the results for *common* domains: i.e., the functionality domains that are in common among the 10 yearly sets of sample projects and that each has at least one positively/strongly associated language

interfacing mechanism (including mixed/hybrid ones such as FFI_EBD) for at least one of the 10 years. As shown in Figure 14, we found eight such common domains.

Overall, similar to the associations between functionality domains and language selections, for any given year, there were multiple specific domains that were strongly associated with one or more language interfacing mechanisms; in fact, for almost every year, the majority of these common domains had at least one strongly associated interfacing mechanism. This observation revealed that developers did generally choose different preferred interfacing mechanisms for constructing multilingual software of different functionality domains regardless of the change of time, although the preferences also changed over time—for any of these common domains.

In particular, it is worth noting that for most of these domains, combining two or three individual language interfacing mechanisms (LIMs) was a dominant practice in multilingual software construction, especially since the year of 2013. The sheer number of strongly associated LIM choices also grew over the years. For instance, in the domain of `End-user application`, the LIM choices consist in `IMI_EBD`, `IMI`, and `HIT` during 2010 through 2012. In 2013 and later years, the number of associated LIM choices rose up to 5. Note that these results are pretty consistent with those observed in Figure 12. For instance, prior to 2013, the primary language choices for `End-user application` multi-language projects were `javascript` and `ruby`, which is consistent with the observation that IMI was the dominant interfacing mechanism (standalone or mixed with one or two other mechanisms) before 2013 because `javascript` and `ruby` are mostly commonly interfaced via IMI—`javascript` is used for constructing front-end code while `ruby` for back-end construction. Starting in 2014, the language choices became notably diversified (e.g., with `java`, `objective-c`, and `php` becoming popular choices), leading to the growing diversity of the interfacing mechanisms adopted.

Overall, during the 10-year span studied, multilingual software construction has been featured with growing diversity of LIM choices, and with fewer and fewer multi-language projects only adopting one single LIM—most of the associated LIM choices are mixed LIMs. And the most popular mixture scheme was to combine two or three single LIMs—in fact, we have not found any project combining more than three single LIMs. This observation is consistent with the rising adoption of mixed LIMs along with the growing diversity of such LIMs as observed in the general evolution of LIMs shown in Figure 11. This trend, seen in any of the eight common functionality domains, indicates the growing complexity of multilingual software construction, with more language choices available and diversifying ways in which multiple languages interact with each other, generally in any software (functionality) domain.

Multilingual software construction, irrespective of the targeted functionality domain, has been constantly featured with having certain strongly-associated language interfacing mechanisms, which have been increasingly hybrid/mixed and diverse, indicating growing complexity of multilingual software construction over time.

4 DISCUSSION

In this section, we systematize our study results across the three research questions and distill further insights into the construction of modern multilingual software systems from our empirical results. Based on these insights and results, we provide actionable suggestions on multilingual software development and research. We also discuss threats to the validity of our results and other limitations of our study.

4.1 Systematization and Implications of Results

Our study results revealed some of the notable practice in modern multilingual software construction concerning frequent/popular individual language choices and language combinations, the ways in which languages interact with each other, and the functionality considerations in relation to language use and selection. These results

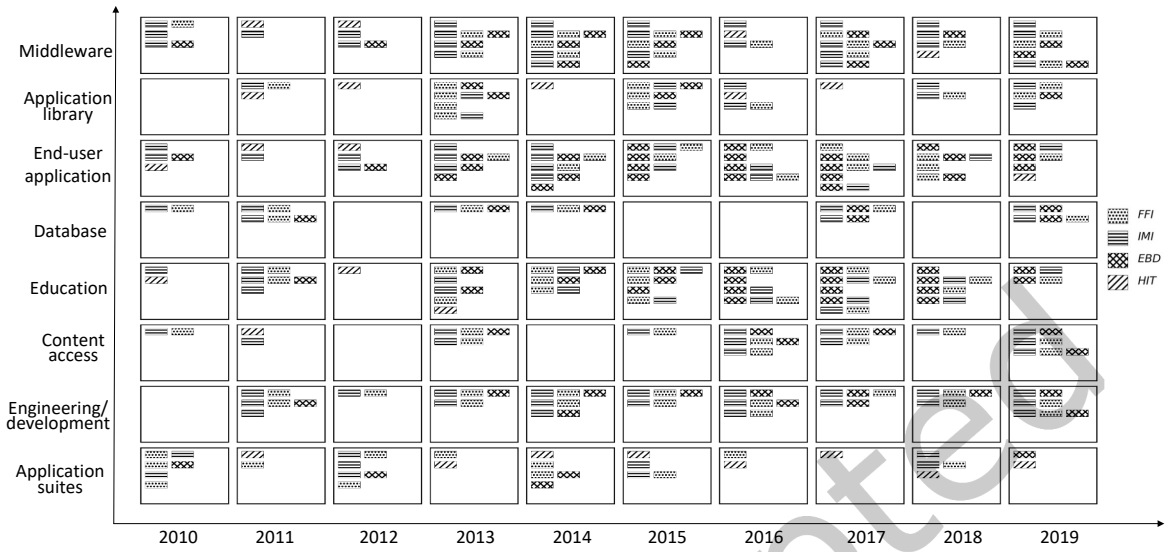

Fig. 14. Evolution of the association between language interfacing types and common functionality domains.

have implications to various stakeholders of multilingual systems, including relevant researchers and software developers.

4.1.1 Relevant to Software Developers. Despite the rising popularity of the multilingual software construction practice, we do not claim or suggest that developers should all move to multilingual development. In fact, there is still a significant portion (e.g., 18% as shown in Figure 2) of software projects in our study dataset developed in one single language. Also, note that the overall growth in the popularity of multilingual software construction has been *gradual*, especially in the recent past years (as shown in Figure 7). Thus, it is reasonably expected that the conventional practice of using a single language (e.g., `c`, `java`, `python`, and `javascript`) for software development will continue to stay for years to come.

Nevertheless, the continuously growing and increasingly dominating prevalence of multilingual software among software projects (at least in the open-source world) as seen in our study (Section 3.3.1) implied the well-recognized merits of combining the benefits of multiple languages in modern software development. For this reason, we believe that the adoption of multilingual construction can be considered a viable option to developers thinking about language selection/use for their future projects. Adding a language to the development does not necessarily mean a need for writing a significant amount of code in that language, according to our results on language distribution (Figure 4) and significance by language code size (Figure 5). Thus, the choice of multilingual construction may not imply an increase in workload and software development costs. Our results show that some language choices, especially scripting languages such as `shell` and declarative languages such as `make`/`cmake`, are *used quite frequently* in multilingual construction, yet they only *contribute minimally* in terms of code size—from an experiential point of view, these languages could be considered when multilingual developers make decisions on (expanding) language selection. Moreover, the evolution of associations between software domains and language selections/interfacing mechanisms also suggests that combining multiple languages through certain interfacing mechanisms is becoming popular (Figures 12–14). For instance, `objective c-ruby` has been a

steadily common language selection for Engineering/development software development, with which FFI has been chosen as a primary interfacing mechanism.

By the same token, developers who are proficient with one language only may potentially consider to add the ability to program with more languages to their skill sets, so that they can leverage the benefits of multilingual software construction and/or more effectively contribute to a *collaborative* multilingual project. Now that the increasing majority of open-source software projects adopted the practice of multilingual construction (Figure 2), those developers should equip themselves with according skills if they aim to become a significant contributors to the open-source community. In particular, Figure 3, together with Figure 9 and Table 5, suggests that these additional languages that developers may want to choose mainly include shell, css, html, ruby, swift, objective c, typescript, and make/cmake.

More specifically, when **making decisions on language choices**, we suggest developers may (1) start with the high-level functionality requirements of their target software project and accordingly, choose the main language that often served for the functionality domain as evidenced historically (e.g., using Table 12 as a reference). For instance, php and c are common choices here for email applications, while go and python have been often selected to start with for developing simulation software. Then (2) based on the main language selected, developers can choose language interfacing mechanisms to decide how to construct the software regarding the connection between different language units. For example, FFI usually brings high performance but also high couplings due to the intra-process invocations with it; in contrast, IMI helps decouple different language units through inter-process communication, leading to lower coupling but also relatively lower performance. Finally, (3) with the main language and specific language interfacing mechanism chosen, developers can select the rest of the eventual language selection for code implementation. As demonstrated in our results, the strong associations between main languages and language interfacing mechanisms and those between language interfacing mechanisms and language selections justify/explain associations between respective functionality domains and the associated language selections. For instance, after choosing javascript as the main language for a spreadsheet development project (according to Table 9), the developer would choose IMI as the interfacing mechanism according to how javascript typically interoperates with other languages, hence choosing the rest of language selection, css and html, based on the common association between these languages and IMI given that javascript is already chosen. Those domains have been shown to be well supported by the associated language combinations in the past and are likely to (albeit not necessarily) remain so in the future. Intuitively, for a particular domain, one reason that some language selections may be more preferable to others is because that each of the selected languages is known to best suit part of the common functionalities of the domain.

We also found that for a given domain the associated language combinations were not unique. This gives developers leeway in making the choices of languages, allowing for preferences in other regards (e.g., familiarity with certain languages). Importantly, our evolutionary characterization revealed that some subsets of languages stood the test of time, showing a strong and persistent presence in language combinations associated with certain functionality domains (Figure 12). We thus particularly suggest developers to begin with those stable subsets of languages associated with the target functionality domains (e.g., css-html-javascript for the development of Education software) and then choose other secondary, cooperating languages.

Similarly, in terms of choosing individual languages for multilingual construction, for some specific functionality domains (e.g., mobile application) there was not a particular language that was *always* considered a must (i.e., always associated with the domain). Yet there appeared to be a *primary* language commonly associated with implementing certain kinds of functionalities (e.g., javascript for End-user application and c for Engineering/development), despite the variations in the companion languages. Thus, it would be a reasonable/viable option for developers to consider including these primary languages in their language selection if they target the respective, associated functionality domains given the historical successes in using the languages for those kinds of functionalities.

Our results revealed some highly significant (in terms of code size attribution) individual languages although they may not be used as frequently as others. These languages include `c`, `c++`, `java`, `c#`, `javascript`, and `python`, which are all well-known mainstream programming languages—a significant portion of multilingual software code is written in these languages due to their rich language features and the strong support (e.g., third-party libraries and developer tools) available in the respective ecosystems. On the other hand, some individual languages were highly frequently used in a variety of domains. These languages include `shell`, `makefile`, `hmtl`, and `css`, due to their great usability and flexibility—a few dozens of lines of code in these languages often suffice for the assistive purposes they serve (e.g., system maintenance/DevOps and data transfer across heterogeneous system components). Again, these historically popular choices can be a good reference for developers to make language use/selection decisions.

When it comes to **deciding on the holistic set of languages** for a project, our results revealed some quite strong co-occurrences between certain domains and the associated language selections. Thus, multilingual developers can immediately refer to such frequent associations to make decisisons on language selection once they have nailed down the target software domain of the project. In particular, `css-javascript-php` has been a solid option for developing application libraries, likely due to the fact (at least in our studied projects) that these libraries often serve for Web applications, for which Web languages (`css, javascripts, php`, etc.) are widely known to be used together often. For `middleware` development, `c-c++-python` has been most frequently selected plausibly due to the well-known merits of `c` and `c++` for low-level system implementation and the complementary merits of `python` in dealing with user interactions and system configurations.

With `End-user applications`, *various* language combinations have been shown to have statistically strong associations, including `php-shell`, `css-html-ruby`, and `java-kotlin`, among others—the diversity of choices here are partly due to the diversity of end-user applications: indeed, our level-1 domain categorization is focused on breaking down end-user applications (into 14 categories). Thus, to start with, developers may want to first nail down the specific category of end-user application they are targeting, and then refer to our findings (Table 8) on which language selections were strongly associated with each (level-1) category. For instance, the `go-shell` combination could be a good starting point to consider for `simulation` software given the super strong association between them; another frequent choice in this domain was `makefile-python-shell`. It turns out that most of the `multimedia` applications (in our dataset) are Web applications; thus, all the past frequently associated language selections included `css-html-javascript`.

As we discussed earlier, an intermediate step and key factor during the language selection decision making is to decide on the language interfacing mechanisms after the main languages are nailed down (e.g., which interfacing mechanisms are suitable for or compatible with the selected main language). Our results revealed some strong historical preferences of language interfacing mechanisms for specific main languages. For instance, FFI had a solid bond with `c++`, `c`, `java`, and `python`, which is unsurprising because these most popular mainstream programming languages enjoyed the availability of dedicated support for interfacing with other languages through FFI (e.g., JNI for `c` to interact with `Java` and ctypes for `c` to interact with `python`). As another example, languages such as `shell` and `go` most frequently interfaced with other languages via IMI, which is also a well-known practice (e.g., the `shell` code invokes other language units through pipes or other inter-process communication (IPC) channels, which all fall in the category of IMI).

Finally, the choices of the main languages and language interfacing mechanisms intuitively affect the final set of languages to use—the remaining languages have to be interoperable with the main language via the chosen interfacing mechanisms. Or the remaining languages may be determined after the main languages are chosen—then the choices of interfacing mechanisms would be limited to those that have existing support available with respect to the entire set of languages selected. In this regard, our results revealed that, if the main language is one of `c`, `c++`, `python`, `javascript`, and `java`, the choices for the other languages have typically been among the same set if FFI is chosen as the interfacing mechanism, `shell` if IMI is chosen, or one of `css`, `html`, and

typescript if EBD is chosen. The reason is that those main languages have well defined interfaces among them, shell has a broad interoperability with those main languages through IPC (i.e., IMI), and javascript, css, html, and typescript are widely known to integrate via embedding one lanugage unit within another (i.e., EBD).

These strong associations did provide a good reference in practice when developers look for a possible set of languages to use in tandem. Of course, the fact that a main language choice, a choice of language interfacing mechanism, or the whole language selection was used frequently for a domain in the past does not necessarily mean it is the best choice for future projects of the same domain. Nevertheless, such associations still provide a pointer for decision making regarding those choices for language selection—e.g., developers may look further into why some sample projects in the past used the associated main languages, language interfacing mechanisms, and language selections, and then make their best decisions based on such deeper understandings.

In summary, our results support a general practical strategy for language use and selection during multilingual software construction, including how to combine the various languages selected: *follow the chain of FD→MaL→LIM→LS*. That is, developers may first decide on the functionality domain (FD) as per the requirements of the software project under development. This target then guides the choice of the main language (MaL), which further informs the selection of the language interfacing mechanism (LIM). Finally, given the decisions on MaL and LIM, the holistic language selection (LS) can be derived. In each of these steps, the specific decisions can be made by referring to the historically strong associations between respective variables (e.g., Table 9 for choosing MaL according to the target FD and Table 10 for selecting LIM according to the chosen MaL). In particular, the decision making as regards to LIM choices immediately addresses the question of how to combine the selected languages since the LIM informs how these chosen languages should interoperate with each other.

4.1.2 Relevant to Researchers. Software construction using multiple languages has been a norm for long (Section 3.1.1), yet our software engineering research community has not paid sufficient attention to particularly support multilingual software development. For instance, tool support for multilingual systems (e.g., testing, maintenance, evolution, and security defense) remains largely lacking, despite a few relevant works addressing a particular case of such systems (e.g., for java-c programs [20, 26]). We hope that our study results could serve as an advocate for more researchers to invest in studying developers needs in multilingual software construction and proposing techniques to assist them with common software engineering tasks in developing multilingual systems.

In particular, in light of our results showing the diversity of language selections, we suggest researchers to keep this diversity in mind when developing techniques and tools to support multilingual software quality assurance. For instance, we found that, despite the almost exclusive focus on java-c (e.g., JNI[5]) programs by existing relevant tool support, java-c was not even among the top 20 language combinations in our studied projects (Section 3.1.4). For example, python was a more frequent collaborating language with c (Table 5), and highly impactful machine learning frameworks such as TensorFlow [19] and PyTorch [39] are developed mainly in python and c. Thus, our results clearly call for research on multilingual software beyond java-c programs (as one particular, non-dominating language selection) and JNI (as one specific mechanism for language interoperability).

On the other hand, despite the fast growing diversity of language choices (from 35 to 138 in 10 years as shown in Figure 8), the number of languages used in one project did not grow as fast. In fact, using 3–5 languages has increasingly become a dominating multilingual system construction decision in terms of the language profile size. This implies that with more individual languages available, developers did not keep adding more languages to a project; rather, they tended to pick a stable number of languages, albeit differently. Moreover, as we showed in Section 3.3, the well-known mainstream languages (e.g., javascript, c/c++, java, c#, shell,

[5]Without loss of generality, java-c programs mainly use the Java native interface (JNI) to realize language interoperations (between Java and C). Yet other interoperability options do exist between Java and C, such as interprocess communication (IPC) [14] and implicit mechanisms such as data transfer through file systems or databases.

and python) had a constant leading presence in the language selections over time. This essentially allows for concentrated efforts that yield meaningful and enduring results. We recommend researchers concentrate on devising techniques that address the interoperability of these few mainstream/primary languages with each other and other secondary/supplementary languages. This approach eliminates the concern that such techniques might swiftly become outdated.

While also addressing interfacing mechanisms among different languages (Figure 14), prior works [26, 31, 32] focused primarily on analyzing multilingual software in which the languages interact via the FFI mechanism. However, our results revealed that FFI, when used alone, was not a popular choice for language interfacing during multilingual software construction. Thus, there is a clear disconnect between research and practice here. Also, our results indicate that IMI is a dominant interfacing mechanism among the studied projects regardless of their functionality domains—we observed the dominance in all the mined domains. Yet currently there has been little existing work on analyzing multilingual code with IMI interfacing. Thus, future techniques enabling (e.g., multi-process [5]) analyses of multilingual software that handle IMI interfacing are critically and urgently needed—note that pursuing a multilingual analysis that is fully agnostic of the interfacing mechanism (i.e., working with any interfacing mechanisms) may not be a fruitful future research direction [48].

In addition, our results on the evolution of language interfacing mechanisms and that of the association between these mechanisms and functionality domains show that, regardless of the target domains, one growing trend is multilingual software construction is the increasing use of hybrid/mixed interfacing mechanisms and the diversity of such mechanisms. However, we are not aware of any existing multilingual code analysis that supports more than one interfacing mechanism at the same time—a gap to be filled in future multilingual software analysis.

4.2 Threats to Validity and Study Limitations

We discuss various kinds of threats to the validity of our results, including threats to internal, external, and construct validity, during we also discuss limitations of our study.

Internal validity. As a common threat to *internal validity*, possible errors may happen during the development procedure of our study toolkit, which might have negatively affected our results. In particular, the functionality domains referred to in our study were identified through a coding process. During this process, both the codebook derivation and coding steps are subject to human biases and errors. To mitigate this threat, we addressed disagreement through meetings/discussions and followed a negotiated agreement, a common approach to dealing with the human biases and errors in inductive/axial coding.

In addition, the correctness of the functionality domain categorization was limited by by the quality of the data sources (e.g., descriptiveness of project topics/descriptions). To reduce this threat, we ignored projects whose descriptions/topics are empty or insignificant in length. A similar threat is that we used the GitHub linguist [18] tool to identify the language profile for each project, making our results subject to the imperfect accuracy of this tool.

Another limitation of our study lies in the inaccuracy of PolyFax [29], the tool we used for identifying the interfacing mechanisms used in a given multi-language project. This tool was evaluated manually in its original paper, which reports precision of 78% (for IMI) up to 96% (for EBD), and the recall ranged from 82% to 90%. The imprecision implies that our results are subject to mistakenly identified interfacing mechanisms, and the limitation in recall means that not all of the interfacing mechanisms were recognized for some of the studied projects.

External validity. The primary threat to the external validity of our study results concerns the sample projects we have collected from GitHub and used. To make the samples more representative of the projects on GitHub, we purposely chose to randomly select a sizable dataset that included projects each meeting several criteria

regarding popularity, liveness, and recency (Section 2.3), for both the SPC and EVC studies. For example, we enforced that any sampled project had at least 1,000 stars, which has been used in prior works [38, 43] as an indicator of popularity. We also have shown that our sample projects covered a variety of software domains (e.g., from OS to musical apps). Yet relative to the entire project set on GitHub, our sample sizes were still considered small. Moreover, our datasets may not well represent all real-world multilingual systems with respect to the multilingual software construction practice, the focus of our holistic study. For this reason, we cannot broadly claim that our findings would surely generalize to any multilingual software. Instead, our results should be best interpreted for the projects that we actually studied. Yet on a side note, we would like to point out that although the total number of projects on GitHub seemed to be huge [47], we found that most of the projects are inactive and many are not software development projects at all [24] hence by nature cannot be considered in our study anyway. This potentially dwarfs the threats to our study results' external validity concerning the sample size. On the other hand, since we only considered software projects on GitHub as the single data source, our findings and conclusions should be best interpreted with respect to open-source software on this particular platform, not necessarily representing any software project in the wild. We chose GitHub as we believe, as many prior peer studies have assumed also, that GitHub is a reasonably credible source of software projects to support studies like ours.

Per our study goals, ideally we would want to use industrial software systems as subjects for our study. However, we currently do not have access to a substantial set of software projects in the industry. Thus, we chose to sample open-sources projects on GitHub because they are readily accessible to us and GitHub is a widely used source of software projects to enable a range of software engineering studies as done in the current literature. Nevertheless, not all of these open-source projects on GitHub can fully represent real-world software systems when it comes to multilingual construction particularly concerning language use/selection. Therefore, our results should be best interpreted with respect to the open-source projects we actually sampled.

On a related note, our study results are pertinent to multilingual software construction in the open-source world, and may not fully reflect modern software development technologies and practices in general (e.g., as applied in software and information industries). For instance, widely used machine learning frameworks such as Tensorflow and PyTorch are multilingual systems, in which the language use/selection decisions are potentially also based on various domain-specific design concerns (e.g., neural network model optimizations towards greater efficiency) in addition to what we have explored in our study. The interfacing mechanisms to be chosen may also be more diverse than the ones we discussed. For example, industrial multilingual software systems may use dedicated interfacing frameworks (e.g., D-bus [40] and gRPC [21]) to enable interoperability support. Another example is the Common Object Request Broker Architecture (CORBA) [11], which, through its Interface Definition Language (IDL), provides language independence in that CORBA objects written in one language can send requests to objects implemented in a different language.

Construct validity. The main threat to construct validity lies in the metrics and measurement procedures adopted in our studies. Concerning the characterization metrics used, we cannot ensure that they were absolutely comprehensive for characterizing multilingual software construction in terms of language use and selection. To mitigate this threat, we chose a diverse set of statistics and dimensions in quantifying the characteristics of multilingual systems, including those used in peer prior works (e.g., the number of unique languages in total used across all the studied projects and that number used for each of the projects). For example, to characterize the overall language use and selection, we have considered metrics for both language prevalence (Section 3.1.2) and language significance (Section 3.1.3) which seemingly overlap with but actually complement to each other.

Regarding measurements, in our EVC study, we used 1,000 sample projects in total for each year, but the actual number of *multilingual* samples varied across the years—as shown in Figure 7, the proportion of these 1,000 that were multilingual projects ranged from 41% to 74%. As a result, the basis of the yearly results for RQ3 (e.g., results

on the evolution of functionality relevance and the evolution of language profile size) was not always consistent. However, we chose to do so for two reasons. First, ensuring the size balance of the yearly datasets avoided overall sampling biases. Second, although the eventual numbers of multilingual projects varied from year to year, the variation exactly represents the actual ratio of multilingual software over software projects of all kinds on GitHub and likely reflects the real-world distribution of multilingual versus single-language software systems.

Another threat to construct validity lies in the consistency of language profiles of the studied projects under the longitudinal lens. As software evolves, which is a norm for any successful software project, the language profile of a software project may evolve as well—some languages can be added while others may be removed during the evolution. We currently cannot guarantee that the language profiles of the projects in our studies are constant during their evolution and maintenance. Instead, we only considered the latest language profiles for all projects in the study dataset. Thus, our empirical results and findings should be interpreted with respect to the language profiles at the time when we obtained them, not necessarily always reflecting the language selection/use of the respective projects throughout their entire lifetime.

Yet another construct validity threat is that we dismissed the possibly varying importance of different languages in characterizing language prevalence and significance in Section 3.1. For instance, we treated a byte of `shell` code equally to a byte of `c` code in computing the language distribution by code size and language significance in terms of code size attribution to different languages within a language profile. We also note that in this paper we aimed to study the multilingual construction of modern software in a holistic manner, thus we did not exclude non-programming languages (e.g., `css` and `html`). In doing so, we also dismissed the differential importance of programming languages versus other assisting (e.g., data modeling) languages, which may have caused biases in the explanations of our results.

5 RELATED WORK

Prior peer works that are related to ours fall in two major categories: characterizing language use and analyzing the effects of language selection.

5.1 Characterizing Language Use

As discussed earlier (Section 1), most of the previous studies concerning language use/selection focused on the use of *individual* languages, as opposed to our focus on the holistic language profiles of multilingual systems (i.e., how multiple languages are *used together* in a single project). We have also discussed how their results relate to our empirical findings in Section 3 when presenting our results.

Like ours, an earlier study [36] also examined the connection between language selection and functionality domains. Yet again this study addressed different languages individually rather than language combinations. In fact, the study did not particularly characterize *multilingual* software but the general language use in any software project. In contrast, part of our study is dedicated to discovering statistical relationships between functional domains and holistic language selections. A high-level summary of our study results was recently presented as an abstract [30], which highlighted those statistical relationships.

Studies explicitly targeting multilingual systems do exist, but they dealt with aspects different from our work and/or approached the characterization in very different ways. For instance, Bissyande et al. reported the popularity of languages in various dimensions (number of projects, code size, age, etc.) and the interoperability mechanisms between languages [4]. Similar kinds of results were also obtained through an empirical assessment of what is called *polyglot-ism* by sampling GitHub projects, which found that there were strong relations between different languages such that various languages tended to be used together in practice [44]. However, unlike our study, the justification of the connections among languages was not examined in depth. Recently, Yang et al. examined developers' discussions on Stack Overflow (SO) regarding the issues and challenges with multilingual

software development, and the current solutions developers have to those challenges [49]. In contrast, our study is based on the actual multilingual code, not natural-language discussions by developers.

Further research on multilingual software demonstrated the associations between different *language groups* as used in different application domains. For example, Mayer and Bauer [33] showed the relationships between one general-purpose language (GPL) and another GPL, between a GPL and a domain-specific language (DSL), and between one DSL and another DSL. Similarly, Delorey et al. [13] studied the links among various individual languages and found some companionship patterns of languages. These studies, while different from ours, potentially complement to our characterization of multilingual software construction. Overall language use statistics and choices have also been investigated directly from developers' opinions [1, 34], which are also complementary to our study based on the work products of multilingual software developers.

In all, despite the existence of a few earlier studies examining language use and selection, existing related works were either limited to single-language software and/or failed to look into the underlying rationale (e.g., as we did from the perspective of functionality relevance) and mechanism (e.g., as we did from the perspective of cross-language interaction) that justify/explain language use and selection. Our study also offers an evolutionary viewpoint on the use/selection of multiple languages *together* and their underlying justification, which is missing in existing peer studies.

5.2 Analyzing the Effects of Language Selection

Beyond the general statistics on language use, researchers have also looked at the consequences of language choices and into how the use of languages affects the various properties of the resulting software products. Ray et al. [43] studied the impact of language features (strong versus weak typing, dynamic versus static typing, etc.) on the defect occurrence in software written in the respective languages. They also went further to explore the relationships of language choices with defect types, both in general and in separate application domains. The study revealed that these relationships/impact were significant but small. Based on these results, a further research refined the approach and obtained more interesting findings that revealed correlations between bug resolution characteristics and language features (e.g., strong/weak typing) and project features (e.g., age, size and domain) [50]. On the other hand, Berger et al. [3] attempted to reproduce the study conducted by Ray et al. [43] and found that the relationships between languages and quality were even much weaker than the originally reported.

In addition, Mayer, Kirsch, and Le [34] provided empirical evidence that most developers had encountered at least one bug related to cross-language linking, and that the use of multiple languages increased the difficulty of bug fixing. Later, Abidi et al. [1] reported that understandability was the most impacted quality attribute in a multi-language system (by the use of multiple languages), based on the perception of 93 developers. More recently, the same authors examined the impact of design smells on fault-proneness of a particular case of multilingual systems—JNI software [2], and revealed positive associations between the two. Likewise, in a study that was also focused on JNI software [20], the researchers found that having more dependencies between code units in different languages increased the risk of functional bugs and security vulnerabilities. More recently, we examined and quantified how language selection affects the proneness of the multi-language projects that select the languages to various kinds of vulnerabilities [28].

Compared to these prior studies, we focused on the characteristics concerning the multilingual construction of open-source software projects by looking at the size and composition of language profiles, offering an updated, multifaceted overview of language use and selection in contemporary multilingual systems. Also, instead of assessing the quality impact of language use and selection, we addressed the functionality relevance of language selection in terms of the quantitative association between the selection and functionality domains. We also dove into this overall association, dissecting/justifying it through studying the effects of the choices of, as

well as intermediate associations with, language interfacing mechanisms and main programming languages on the ultimate language selection holistically. Moreover, we offered a longitudinal view of such associations in multilingual systems, which has not been explored before.

6 CONCLUSIONS

We presented a large-scale characterization study on language use and selection in multilingual software with projects randomly sampled from GitHub in order to understand the multilingual construction of modern software systems. Using carefully chosen and specially developed tools along with relevant statistical analyses, we provided a recent, multi-faceted, and evolutionary view of language use and selection in the multi-language world, and looked into the functionality rationales behind the language selection decisions as a way to justify the decisions.

Our study revealed dominating and continuously rising prevalence of multilingual construction in modern open-source software projects. We also discovered the growing trend of using 3 to 5 languages in multilingual software and top language selections, along with the increasing diversity of language choices. We further found that language selection was generally quite strongly associated with some functionality domains. Over time, the top language selections for those domains changed considerably, whereas the primary languages appeared to be relatively stable. The strong association patterns regarding how functionality domains targeted by a project, languages selected in the project, the main language included in the selection, and interfacing mechanism used for those selected languages to communicate with each other provide immediate references or even guidance for language use/selection during multilingual software construction. We reported major findings and drew insights from empirical results, which together led to practical, actionable suggestions for both researchers and developers of multilingual systems.

For future work, we plan to leverage our insights gained from this study to develop practical tools to support the quality assurance of multilingual software. An additional next step is to expand our current study by further analyzing the interfacing between languages and assessing the implications of different interfacing mechanisms to the correctness and security of multilingual code.

ACKNOWLEDGMENT

We thank our associate editor and reviewers for insightful and constructive comments. This work was supported in part by the U.S. National Science Foundation (NSF) under Grant CCF-2146233 and in part by the U.S. Office of Naval Research (ONR) under Grant N000142212111.

REFERENCES

[1] Mouna Abidi, Manel Grichi, and Foutse Khomh. 2019. Behind the scenes: developers' perception of multi-language practices. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*. 72–81.

[2] Mouna Abidi, Md Saidur Rahman, Moses Openja, and Foutse Khomh. 2021. Are multi-language design smells fault-prone? An empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–56.

[3] Emery D Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. 2019. On the impact of programming languages on code quality: a reproduction study. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 4 (2019), 1–24.

[4] Tegawendé F Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillère. 2013. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *2013 IEEE 37th annual computer software and applications conference*. 303–312.

[5] Haipeng Cai and Xiaoqin Fu. 2022. D^2ABS: A Framework for Dynamic Dependence Analysis of Distributed Programs. *IEEE Transactions on Software Engineering (TSE)* 48, 12 (2022), 4733–4761.

[6] Haipeng Cai and Barbara Ryder. 2017. DroidFax: A Toolkit for Systematic Characterization of Android Applications. In *International Conference on Software Maintenance and Evolution (ICSME)*. 643–647.

[7] Haipeng Cai and Barbara Ryder. 2021. A Longitudinal Study of Application Structure and Behaviors in Android. *IEEE Transactions on Software Engineering (TSE)* 47, 12 (2021), 2934–2955.

[8] Haipeng Cai and Douglas Thain. 2016. DistIA: A Cost-Effective Dynamic Impact Analysis for Distributed Programs. In *IEEE/ACM Conference on Automated Software Engineering (ASE)*. 344–355.
[9] John L Campbell, Charles Quincy, Jordan Osserman, and Ove K Pedersen. 2013. Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. *Sociological methods & research* 42, 3 (2013), 294–320.
[10] Yaofei Chen, Rose Dios, Ali Mili, Lan Wu, and Kefei Wang. 2005. An empirical study of programming language trends. *IEEE software* 22, 3 (2005), 72–79.
[11] CORBA. 1991. Common Object Request Broker Architecture (CORBA). https://www.omg.org/spec/CORBA/.
[12] Juliet Corbin and Anselm Strauss. 2014. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications.
[13] Daniel P Delorey, Charles D Knutson, and Christophe Giraud-Carrier. 2007. Programming language trends in open source development: An evaluation using data from all production phase sourceforge projects. In *Second International Workshop on Public Data about Software Development (WoPDaSD'07)*. 1–5.
[14] Xiaoqin Fu, Haipeng Cai, Wen Li, and Li Li. 2020. Seads: Scalable and Cost-Effective Dynamic Dependence Analysis of Distributed Systems via Reinforcement Learning. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2020), 1–45.
[15] Xiaoqin Fu, Boxiang Lin, and Haipeng Cai. 2022. DistFax: A Toolkit for Measuring Interprocess Communications and Quality of Distributed Systems. In *IEEE/ACM International Conference on Software Engineering (ICSE), Companion Proceedings*. 51–55.
[16] GitHub. 2023. GitHut 2.0 - GitHub Language Stats. https://madnight.github.io/githut/.
[17] GitHub, Inc. 2020. GitHub: a US-based global company, provides hosting for software development version control using Git. https://github.com/.
[18] GitHub, Inc. 2020. GitHub Developer: provides APIs to retrive or query repositories in GitHub. https://developer.github.com/v3.
[19] Google Brain Team. 2021. The TensorFlow project. https://github.com/tensorflow/tensorflow.
[20] Manel Grichi, Mouna Abidi, Fehmi Jaafar, Ellis E Eghan, and Bram Adams. 2020. On the impact of interlanguage dependencies in multilanguage systems empirical case study on java native interface applications (JNI). *IEEE Transactions on Reliability* 70, 1 (2020), 428–440.
[21] gRPC. 2020. gRPC Tutorial. https://grpc.io/docs/.
[22] Fauna Herawati, Muhamad Satria Mandala Pua Upa, Rika Yulia, and Retnosari Andrajati. 2019. Antibiotic Consumption at a Pediatric Ward at a Public Hospital in Indonesia. *Asian Journal of Pharmaceutical and Clinical Research* 12, 8 (2019), 64–67.
[23] Capers Jones. 2009. *Software engineering best practices*. McGraw-Hill, Inc.
[24] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071.
[25] Siim Karus and Harald Gall. 2011. A study of language usage evolution in open source software. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 13–22.
[26] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. 2020. Broadening Horizons of Multilingual Static Analysis: Semantic Summary Extraction from C Code for JNI Program Analysis. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 127–137.
[27] Justin Lestal. 2023. How many programming and coding languages are there? https://devskiller.com/how-many-programming-languages/.
[28] Wen Li, Li Li, and Haipeng Cai. 2022. On the Vulnerability Proneness of Multilingual Code. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 847–859.
[29] Wen Li, Li Li, and Haipeng Cai. 2022. PolyFax: a toolkit for characterizing multi-language software. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1662–1666.
[30] Wen Li, Na Meng, Li Li, and Haipeng Cai. 2021. Understanding Language Selection in Multi-Language Software Projects on GitHub. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 256–257.
[31] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. 2022. PolyCruise: A Cross-Language Dynamic Information Flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. 2513–2530.
[32] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. 2023. PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1379–1396.
[33] Philip Mayer and Alexander Bauer. 2015. An empirical analysis of the utilization of multiple programming languages in open source projects. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. 1–10.
[34] Philip Mayer, Michael Kirsch, and Minh Anh Le. 2017. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *Journal of Software Engineering Research and Development* 5, 1 (2017), 1.
[35] Slashdot Media. 2020. SourceForge: The Complete Open-Source and Business Software Platform. https://sourceforge.net/.
[36] Leo A Meyerovich and Ariel S Rabkin. 2013. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 1–18.
[37] Matthew B Miles, A Michael Huberman, and Johnny Saldaña. 2018. *Qualitative data analysis: A methods sourcebook*. Sage publications.

[38] Michail Papamichail, Themistoklis Diamantopoulos, and Andreas Symeonidis. 2016. User-perceived source code quality estimation based on static analysis metrics. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 100–107.
[39] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2021. The PyTorch project. https://github.com/pytorch/pytorch.
[40] Havoc Pennington. 2020. D-Bus Tutorial. https://dbus.freedesktop.org/doc/dbus-tutorial.html.
[41] Raffaele Perego, Salvatore Orlando, and P Palmerini. 2001. Enhancing the apriori algorithm for frequent set counting. In *International Conference on Data Warehousing and Knowledge Discovery*. 71–82.
[42] Sebastian Raschka. 2020. Mlxtend: (machine learning extensions), a Python library of useful tools for the day-to-day data science tasks. http://rasbt.github.io/mlxtend.
[43] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in GitHub. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 155–165.
[44] Federico Tomassetti and Marco Torchiano. 2014. An empirical assessment of polyglot-ism in GitHub. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. 1–4.
[45] Sergi Valverde and Ricard V Solé. 2015. Punctuated equilibrium in the large-scale evolution of programming languages. *Journal of The Royal Society Interface* 12, 107 (2015), 20150249.
[46] Bogdan Vasilescu, Alexander Serebrenik, and Mark GJ van den Brand. 2013. The Babel of software development: Linguistic diversity in Open Source. In *International Conference on Social Informatics*. 391–404.
[47] Jason Warner. 2018. Thank you for 100 million repositories. https://github.blog/2018-11-08-100M-repos/.
[48] Haoran Yang, Wen Li, and Haipeng Cai. 2022. Language-Agnostic Dynamic Analysis of Multilingual Code: Promises, Pitfalls, and Prospects. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Ideas, Visions and Reflections*. 1621–1626.
[49] Haoran Yang, Weile Lian, Shaowei Wang, and Haipeng Cai. 2023. Demystifying Issues, Challenges, and Solutions for Multilingual Software Development. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 1840–1852.
[50] Jie M Zhang, Feng Li, Dan Hao, Meng Wang, Hao Tang, Lu Zhang, and Mark Harman. 2019. A study of bug resolution characteristics in popular programming languages. *IEEE Transactions on Software Engineering* 47, 12 (2019), 2684–2697.