A Large Scale Study of Programming Languages and Code Quality in Github

Baishakhi Ray , Daryl Posnett , Vladimir Filkov, Premkumar T Devanbu
{bairay@, dpposnett@, filkov@cs., devanbu@cs.}ucdavis.edu
Department of Computer Science, University of California, Davis, CA, 95616, USA

ABSTRACT

What is the effect of programming languages on software quality? This question has been a topic of much debate for a very long time. In this study, we gather a very large data set from GitHub (729 projects, 80 Million SLOC, 29,000 authors, 1.5 million commits, in 17 languages) in an attempt to shed some empirical light on this question. This reasonably large sample size allows us to use a mixed-methods approach, combining multiple regression modeling with visualization and text analytics, to study the effect of language features such as static *v.s.* dynamic typing, strong *v.s.* weak typing on software quality. By triangulating findings from different methods, and controlling for confounding effects such as team size, project size, and project history, we report that language design does have a significant, but modest effect on software quality. Most notably, it does appear that strong typing is modestly better than weak typing, and among functional languages, static typing is also somewhat better than dynamic typing. We also find that functional languages are somewhat better than procedural languages. It is worth noting that these modest effects arising from language design are overwhelmingly dominated by the process factors such as project size, team size, and commit size. However, we hasten to caution the reader that even these modest effects might quite possibly be due to process factors, *e.g.,* the preference of certain personality types for functional, static and strongly typed languages.

1. INTRODUCTION

A variety of debates ensue during discussions whether a given programming language is "the right tool for the job". While some of these debates can sometimes appear to be tinged with almost religious fervour, most people would agree that a programming language has an effect not only on the coding process but also on the properties of the resulting artifact. The size of the effect can be significant and important overall, and can depend on the language properties.

Advocates of strong static typing argue that type inference will catch software bugs early. Advocates of dynamic typing may argue that rather than spend a lot of time correcting annoying static type

errors arising from sound, conservative static type checking algorithms in compilers, it's better to rely on strong dynamic typing to catch errors as and when they arise. These debates, however, have largely been of the armchair variety; usually the evidence offered in support of one position or the other tends to be anecdotal.

Empirical evidence for the associations, if any, that exist between programming language choice, their properties, usage domains, and code quality, can help developers make more educated choices. Getting such evidence, however, is a challenging task, given the number of other factors that influence software engineering outcomes. Considering software quality, for example, there are a number of well-known influential factors, including size [10], the number of developers [31, 5], and age/maturity [15]. These factors are known to have a pretty strong influence on software quality, and indeed, such process factors can predict defect localities quite effectively [27].

One approach to teasing out just the effect of language properties, even in the face of such daunting confounds, is to do a *controlled experiment*. Recently Hanenberg [13] and colleagues have conducted experiments in controlled settings with tasks of limited scope, with students, using custom-crafted languages with static or dynamic typing (based on experimental treatment setting). While type of controlled study is *"El Camino Real"* to solid empirical evidence, another opportunity has recently arisen, thanks to the large number of open source projects collected in software forges such as GitHub.

GitHub has a very large number of projects, in quite a few different languages, with different combinations of features. Some of these projects have a great many developers, and others not; some are long lived, and others of more recent vintage; some are very large, and others not as much. Each project repository provides extensive historical record, from which extensive data can be gathered, including about contribution history, project size, authorship, and defect repair. We use this extensive and varied data to determine the effects of language features on defect occurrence. To study this complex and multi-dimensional phenomena, we use a multi-pronged approach, combining multiple tools. Our approach is thus best described as mixed-methods, or triangulation [9] approach: we begin with a quantitative (multiple regression) study and further examine the results using mixed methods: text analysis, clustering, and visualization. The observations from the mixed methods largely confirm the findings of the quantitative study. We summarize the main features of our work below.

- We leverage a categorization of some important features of programming languages that prior knowledge suggests are important for software quality (strong *v.s.* weak typing, dynamic *v.s.*. static typing, memory managed *v.s.* unmanaged, and scripting vs. compiled) to study their impact on defect quality.

- We use multiple regression to control for a range of different factors (size, project history, number of contributors, *etc.*) and study the impact of the above features on defect occurrence. The findings are listed under RQ1 and RQ2 in Section 3.

- We use text analysis and clustering methods to group projects into domains of application, and also the defects into categories of defects; we then use heat maps to study relationships of project types and defect types to programming languages. The findings from this study (RQ3 and RQ4 in Section 3) are consistent with statistical results.

While the use of regression analysis to deal with confounding variables is not without controversy, we submit that a couple of factors increase the credibility of our results: a fairly large sample size, and use of mixed methods to qualitatively explore and largely confirm the findings from the regression model.

2. METHODOLOGY

Here, we describe the languages and GitHub projects that we collected, and the analysis methods we used to answer our research questions. First, in Section 2.1, we detail the languages. Then, we describe how we have collected and processed the data, in Section 2.2. Next, in Section 2.3, we talk about our rationale behind categorizing the studied languages into different classes based on certain language primitives. We discuss our methodology of identifying project domains and bug categories in Section 2.4 and Section 2.5, followed by a description of regression model in Section 2.6.

2.1 Study Subjects

To understand whether choice of programming languages have any impact on software quality, first we chose the top 20 programming languages from GitHub. We disregard CSS, Shell script, and Vim script as they are not considered to be general purpose languages. Then, for each of the studied languages we retrieve top 50 projects that are primarily written in that language. Table 1 shows top three projects in each language, based on their popularity[1]. Thus, we analyze 850 projects in total, spanning over 17 languages.

Table 1: Top three projects in each language

Language	Projects
C	linux, git, php-src
C++	node-webkit, phantomjs, mongo
C#	SignalR, SparkleShare, ServiceStack
Objective-C	AFNetworking, GPUImage, RestKit
Go	docker, lime, websocketd
Java	storm, elasticsearch, ActionBarSherlock
CoffeeScript	coffee-script, hubot, brunch
JavaScript	bootstrap, jquery, node
TypeScript	bitcoin, litecoin, qBittorrent
Ruby	rails, gitlabhq, homebrew
Php	laravel, CodeIgniter, symfony
Python	flask, django, reddit
Perl	gitolite, showdown, rails-dev-box
Clojure	LightTable, leiningen, clojurescript
Erlang	ChicagoBoss, cowboy, couchdb
Haskell	pandoc, yesod, git-annex
Scala	Play20, spark, scala

---

[1]Popularity is measured by the number of stars assigned to a project reflecting how many people find the project interesting: https://help.github.com/articles/stars.

2.2 Data Collection

To retrieve the top programming languages and the corresponding projects from GitHub, we used *GitHub Archive* [1] , a database that records all public GitHub activities. The archive logs eighteen different GitHub events including new commits, fork events, pull request, developers' information, and issue tracking of all the open source GitHub projects on an hourly basis. The archive data is uploaded to Google BigQuery [2] service to provide an interface for interactive data analysis.

Identifying top languages. The top languages in GitHub are measured by first finding the number of open source GitHub projects developed in each language, and then choosing the top languages with the maximum number of projects. However, since multiple languages are often used to develop a project, assigning a single language to a project is difficult. GitHub Linguist [11] can measure such a language distribution of a GitHub project repository. Since languages can be identified by the extension of a project's source files, GitHub Linguist counts the number of source files with different extensions. The language with the maximum number of source files is assigned as *primary language* of the project. GitHub Archive stores this information. We query GitHub Archive to aggregate projects based on their primary language. Then we select the top languages having maximum number of projects for further analysis as shown in Table 1.

Retrieving popular projects. For each selected language, we query GitHub Archive to retrieve project repositories that are primarily written on that language. We then count the number of *stars* associated with that repository. According to GitHub documentation, the number of stars relate to how many people are interested in that project. Thus, we assume that stars indicates the popularity of a project. We select the top 50 projects in each language having the maximum number of stars. To ensure that these projects have a sufficient development history, we filter out the projects having less than 28 commits, where 28 is the first quartile commit number of all the projects. This leaves us with 729 projects. Table 1 shows the top three projects in each language. This includes projects like Linux, mysql, android-sdk, facebook-sdk, mongodb, python, ruby source code etc.

Retrieving project evolution history. For each of these 729 projects, we downloaded all commits using the command: `git log –no-merges –numstat`. We downloaded the non merged commits, along with the commit logs, author date, and author name. The numstat flag shows the number of added and deleted lines per file associated with each commit. This helps us to compute code churn and the number of files modified per commit. We also retrieve the languages associated with each commit from the extensions of the modified files. Note that, one commit can have multiple language tags. For each commit, we calculate its *commit age* by subtracting its commit date from the first commit of the corresponding project. We also calculate some other project related statistics, including maximum commit age of a project and the total number of developers; we use them as control variables in our regression model as discussed in Section 3. We further identify the bug fix commits made to individual projects by searching for error related keywords: 'error', 'bug', 'fix' , 'issue', 'mistake', 'incorrect', 'fault', 'defect' and 'flaw' in the commit log using a heuristic similar to that in Mockus and Votta [21].

Table 2 summarizes our data set. Since a project may use multiple languages, the first column of the table shows the total number of projects that use a certain language at some capacity. We further exclude some languages from a project that have fewer than 20 commits, in that language, over the project's history, where 20 is the first quartile value of the total number of commits made in a project using a particular language. For example, we find 220 projects that use more than 20 commits in C. This ensures that the

Table 2: Study Subjects

Language	Project Details				Total Commits		BugFix Commits	
	#Projects	#Authors	SLOC (KLOC)	Period	#Commits	#Insertion (KLOC)	#Bug Fixes	#Insertion (KLOC)
C	220	13,769	22,418	1/1996 to 2/2014	447,821	75,308	182,568	20,121
C++	149	3,831	12017	8/2000 to 2/2014	196,534	45,970	79,312	23,995
C#	77	2,275	2,231	6/2001 to 1/2014	135,776	27,704	50,689	8,793
Objective-C	93	1,643	600	7/2007 to 2/2014	21,645	2,400	7,089	723
Go	54	659	591	12/2009 to 1/2014	19,728	1,589	4,423	269
Java	141	3,340	5,154	11/1999 to 2/2014	87,120	19,093	35,128	7,363
CoffeeScript	92	1,691	260	12/2009 to 1/2014	22,500	1,134	6,312	269
JavaScript	432	6,754	5,816	2/2002 to 2/2014	118,318	33,134	39,250	8,676
TypeScript	96	789	18,363	3/2011 to 2/2014	14,987	65,910	2,443	8,970
Ruby	188	9,574	1,656	1/1998 to 1/2014	122,023	5,804	30,478	1,649
Php	109	4,862	3,892	12/1999 to 2/2014	118,664	16,164	47,194	5,139
Python	286	5,042	2,438	8/1999 to 2/2014	114,200	9,033	41,946	2,984
Perl	106	758	86	1/1996 to 2/2014	5,483	471	1,903	190
Clojure	60	843	444	9/2007 to 1/2014	28,353	1,461	6,022	163
Erlang	51	847	2484	05/2001 to 1/2014	31,398	5,001	8,129	1,970
Haskell	55	925	837	01/1996 to 2/2014	46,087	2,922	10,362	508
Scala	55	1,260	1,370	04/2008 to 1/2014	55,696	5,262	12,950	836
Summary	729	28,948	80,657	1/1996 to 2/2014	1,586,333	318,360	566,198	92,618

studied languages have significant activity within the projects. In summary, we study 729 projects developed in 17 languages with 18 years of parallel evolution history. This includes 29 thousand different developers, 1.58 million commits, and 566 thousand bug fix commits.

2.3 Categorizing Languages

We define language classes based on several properties of the language that have been thought to influence language quality [13, 14, 17], as shown in Table 3. The *Programming Paradigm* indicates whether the project is written in a procedural, functional, or scripting language. We do not distinguish object-oriented languages in this work as there is no clear distinction between purely object-oriented languages and procedural languages and the difference mostly depends on programming style. *Compile Class* indicates whether the project is statically or dynamically typed. *Type Class* classifies languages based on static and weak typing. Finally, *Memory Class* indicates whether the language requires developers to manage memory. We treat Objective-C as unmanaged, though Objective-C follows a hybrid model, because we observe many memory errors in Objective-C codebase, as discussed in RQ4 in Section 3. Table 3 summarizes the distribution of the languages within each language class.

2.4 Identifying the Project Domain

We classify the studied projects into different domains based on their features and functionalities using a mix of automated and manual techniques. The projects in GitHub come with project descriptions and Readme files that describe their functionalities. First, we used Latent Dirichlet Allocation(LDA) [6], a well-known topic analysis algorithm, on the text extracted from the project descriptions and Readme files. Given a set of documents, LDA identifies a set of topics where each topic is represented as probability of generating different words. For each document, LDA also estimates the probability of assigning that document to each topic.

We use LDA to detect 30 domains (i.e. topics) and estimate the probability of each project belonging to these domains. For example, LDA assigned the *facebook-android-sdk* project to the following topic with high probability: $(0.042 * facebook + 0.017 * support + 0.010 * swank/slime + 0.009 * directory - based + 0.007 * cassovary + 0.007 * framework + 0.007 * live + 0.007 * environments. + 0.007 * transforming + 0.007 * contextual)$.

Here, the text values are the words and the numbers are their probability of belonging to that domain. For clarity, we only show the top 10 words in the detected domain.

Since such auto-detected domains include several project-specific keywords, such as facebook, swank/slime as shown in the previous example, it is hard to identify the underlying common functionalities. Hence, we manually inspect each of the thirty domains to identify project-name-independent, domain-identifying keywords. Manual inspection helps us in assigning a meaningful name to each domain. For example, for the domain described earlier, we identify the keywords framework, environments, and transforming to call it *development framework*. We manually rename all the thirty auto-detected domains in similar manner and find that the majority of the projects fall under six domains: Application, Database, Code-Analyzer, Middleware, Library, and Framework. We also find that some projects like "online books and tutorials", "scripts to setup environment", "hardware programs" etc. do not fall under any of the above domains, therefore, we assign them to a catchall domain labeled as *Other*. This classification of projects into domains was subsequently checked and confirmed by another member of our re-

Table 3: Different Types of Language Classes

Language Classes	Categories	Languages
Programming Paradigm	Procedural	C, C++, C#, Objective-C, Java, Go
	Scripting	CoffeeScript, JavaScript, TypeScript, Python, Perl, Php, Ruby
	Functional	Clojure, Erlang, Haskell, Scala
Compilation Class	Static	C, C++, C#, Objective-C, Java, Go, Haskell, Scala
	Dynamic	CoffeeScript, JavaScript, TypeScript, Python, Perl, Php, Ruby, Clojure, Erlang
Type Class	Strong	C#, Java, Go, TypeScript, Python, Ruby, Clojure, Erlang, Haskell, Scala
	Weak	C, C++, Objective-C, CoffeeScript, JavaScript, Perl, Php
Memory Class	Managed	Others
	Unmanaged	C, C++, Objective-C

search group. Table 4 summarizes the identified domains resulting from this process. In our study set, the Framework domain has the greatest number of projects (206), while the Database domain has the fewest number of projects (43).

Table 4: Characteristics of Domains

Domain Name	Domain Characteristics	Example Projects	Total Proj
Application (APP)	end user programs.	bitcoin, macvim	120
Database (DB)	sql and nosql databases	mysql, mongodb	43
CodeAnalyzer (CA)	compiler, parser interpreter etc.	ruby, php-src	88
Middleware (MW)	Operating Systems, Virtual Machine, etc.	linux, memcached	48
Library (LIB)	APIs, libraries etc.	androidApis, opencv	175
Framework (FW)	SDKs, plugins	ios sdk, coffeekup	206
Other (OTH)	-	Arduino, autoenv	49

2.5 Categorizing Bugs

While fixing software bugs, developers often leave important information in the commit logs about the nature of the bugs; *e.g.,* why the bugs arise, how to fix the bugs. We exploit such information to categorize the bugs. The bug categories and classification method are adapted from the bug classification study by Tan *et al.* [18, 28]. First, we categorize the bugs based on their **Cause** and **Impact**. Root Causes are further classified into disjoint sub-categories of errors—Algorithmic, Concurrency, Memory, generic Programming, and Unknown. The bug Impact is also classified into four disjoint sub-categories: Security, Performance, Failure, and other unknown categories. Thus, each bug fix commit has a Cause and a Impact type. For example, a Linux bug corresponding to the bug fix message: "return if prcm_base is NULL.... This solves the following crash" [2] was caused due to a missing check (programming error), and impact was crash (failure). Table 5 shows the description of each bug category. This classification is performed in two phases:

(1) **Keyword search.** We randomly choose 10% of the bug-fix messages and use a keyword based search technique to automatically categorize the messages with potential bug types. We use this annotation, separately, for both Cause and Impact types. We chose a restrictive set of keywords and phrases as shown in Table 5. For example, if a bug fix log contains any of the keywords: deadlock, race condition or synchronization error, we infer it is related to the *Concurrency* error category. Such a restrictive set of keywords and phrases help to reduce false positives.

(2) **Supervised classification.** We use the annotated bug fix logs from the previous step as training data for supervised learning techniques to classify the remainder of the bug fix messages by treating them as test data. We first convert each bug fix message to a bag-of-words. We then remove words that appear only once among all of the bug fix messages. This reduces project specific keywords. We also stem the bag-of-words using standard natural language processing (NLP) techniques. Finally, we use a well-known supervised classifier: Support Vector Machine(SVM) [29] to classify the test data.

To evaluate the accuracy of the bug classifier, we manually annotated 180 randomly chosen bug fixes, equally distributed across

all of the categories. We then compare the result of the automatic classifier with the manually annotated data set. The following table summarizes the result for each bug category.

	precision	recall
Performance	70.00%	87.50%
Security	75.00%	83.33%
Failure	80.00%	84.21%
Memory	86.00%	85.71%
Programming	90.00%	69.23%
Concurrency	100.00%	90.91%
Algorithm	85.00%	89.47%
Average	83.71%	84.34%

The result of our bug classification is shown in Table 5. In the Cause category, we find most of the bugs are related to generic programming errors (88.53%). Such high proportion is not surprising because it involves a wide variety of programming errors including incorrect error handling, type errors, typos, compilation errors, incorrect control-flow, and data initialization errors. The rest 5.44% appears to be incorrect memory handling; 1.99% is concurrency bugs, and 0.11% is algorithmic errors. Analyzing the impact of the bugs, we find 2.01% are related to security vulnerability; 1.55% is performance errors, and 3.77% causes complete failure to the system. There is 1.04% of the bug fix messages which our technique could not classify to any Cause or Impact categories. They are shown as Unknown type.

2.6 Statistical Methods

We use regression modeling to describe the relationship of a set of predictors against a response. In this paper we model the number of defective commits against other factors related to software projects. All regression models use *negative binomial regression*, or *NBR* to model the non-negative counts of project attributes such as the number of commits. NBR is a type of generalized linear model used to model non-negative integer responses. It is appropriate here as NBR is able to handle *over-dispersion*, *e.g.,* cases where the response variance is greater than the mean [7].

In our models we control for several language per-project dependent factors that are likely to influence the outcome. Consequently, each (language, project) pair is a row in our regression and is viewed as a sample from the population of open source projects. We log-transform dependent count variables as it stabilizes the variance and usually improves the model fit [7]. We verify this by comparing transformed with non transformed data using the AIC and Vuong's test for non-nested models [30].

To check that excessive multi-collinearity is not an issue, we compute the variance inflation factor *(VIF)* of each dependent variable in all of the models. Although there is no particular value of VIF that is always considered excessive, we use the commonly used conservative value of 5 [7]. We check for and remove high leverage points through visual examination of the residuals vs leverage plot for each model, looking for both separation and large values of Cook's distance.

We employ *effects*, or *contrast*, coding in our study to facilitate interpretation of the language coefficients [7]. Effects codes differ from the more commonly used dummy, or treatment, codes which compare a base level of the factor with one or more *treatments*. With effects coding, each coefficient indicates the relative effect of the use of a particular language on the response as compared to the weighted mean of the dependent variable across all projects. Since our factors are *unbalanced*, *i.e.*, we have different numbers of projects in each language, we use weighted effects coding which takes into account the scarcity of a language. This method has been used previously in software engineering to study the impact of pattern roles on change proneness [26]. As with treatment coding, it

Table 5: Categories of bugs and their distribution in the whole dataset

	Bug Type	Bug Description	Search keywords/phrases	count	%count
Cause	**Algorithm** (Algo)	algorithmic or logical errors	algorithm	606	0.11
	Concurrancy (Conc)	multi-threading or multi-processing related issues	deadlock, race condition, synchronization error.	11111	1.99
	Memory (Mem)	incorrect memory handling	memory leak, null pointer, buffer overflow, heap overflow, null pointer, dangling pointer, double free, segmentation fault.	30437	5.44
	Programming (Prog)	generic programming errors	exception handling, error handling, type error, typo, compilation error, copy-paste error, refactoring, missing switch case, faulty initialization, default value.	495013	88.53
Impact	**Security** (Sec)	correctly runs but can be exploited by attackers	buffer overflow, security, password, oauth, ssl	11235	2.01
	Performance (Perf)	correctly runs with delayed response	optimization problem, performance	8651	1.55
	Failure (Fail)	crash or hang	reboot, crash, hang, restart	21079	3.77
	Unknown (Unkn)	not part of the above seven categories		5792	1.04

is still necessary to omit a factor to compute the model. Since the use of effects codes compares each level to the grand mean, however, we can easily extract the missing coefficient from the other model coefficients, or more practically, we simply re-compute the coefficients using a different level of the factor as a base [7].

To test for the relationship between two factor variables we use a Chi-Square test of independence [19]. After confirming a dependence we use Cramer's V, an $r \times c$ equivalent of the phi coefficient for nominal data, to establish an effect size [8].

3. RESULTS

Prior to analyzing language properties in more detail, we begin with a straightforward question that directly addresses the core of what some fervently believe must be true, namely:

RQ1. Are some languages more defect prone than others?

We evaluate this question using a negative binomial regression model with languages encoded with weighted effects codes as predictors for the number of defect fixing commits. The model details are shown in Table 6.

In order to model project language against the number of defect fixing commits we must control for factors that will clearly influence the number of defect fixes. We include the age of the project as older projects will generally have a greater number of defect fixes. Trivially, the number of commits to a project will also impact the number of defect fixing commits. Additionally, the number of developers who touch a project and the raw size of the project are both expected to grow with the defect fixing commits.

The sign and magnitude of the Estimate in the above model relates the predictors to the outcome. As discussed, the first four variables are control variables and we are not interested in their impact on the outcome other than to say, in this case, that they are all positive, as expected, and significant.

The language variables are indicator, or factor, variables for each project. The coefficient compares each language to the grand weighted mean of all languages in all projects. The language coefficients can be broadly grouped into three general categories. The first category are those for which the coefficient is statistically insignificant and the modeling procedure could not distinguish the coefficient from zero. These languages may behave similarly to the average or they may have wide variance. The remaining coefficients are significant and either positive or negative. For those with positive coefficients we can expect that the language is associated with, *ceteris paribus*, a greater number of defect fixes. These languages include C, C++, JavaScript, Objective-C, Php, and Python. The languages Clojure, Haskell, Ruby, Scala, and TypeScript, all have negative coefficients implying that these languages are less likely than the average to result in defect fixing commits.

One should take care not to overestimate the impact of language on defects. While these relationships are statistically significant, the effects are quite small. In the analysis of deviance table below we can see that activity in a project accounts for the majority of explained deviance. Note that all variables are significant, that is, all of the factors above account for some of the variance in the number of defective commits. The next closest predictor, which accounts for less than one percent of the total deviance, is language. All other controls taken together do not account for as much deviance as language.

	Df	Deviance	Resid. Df	Resid. Dev	Pr(>Chi)
NULL			1113	38526.51	
log commits	1	36986.03	1112	1540.48	0.0000
log age	1	42.70	1111	1497.78	0.0000
log size	1	12.25	1110	1485.53	0.0005
log devs	1	48.22	1109	1437.30	0.0000
language	16	242.89	1093	1194.41	0.0000

Although we expressed the impact of language in this model as a percentage of deviance, we hasten to remind the reader that although interpretation is similar, in a rough sense, to a percentage of the total variance explained in an ordinary least squares regression, it is not accurate to say that the measures are synonymous [7]. About the best we can do with this information is to observe that it is a small affect. We can read the coefficients as the expected change in the log of the response for a one unit change in the predictor with all other predictors held constant; *i.e.*, for a coefficient β_i, a one unit change in β_i yields an expected change in the response of e^{β_i}. For the factor variables, this expected change is normally interpreted as a comparison to a base factor. In our case, however, the use of weighted effects coding allows us to compare this expected change to the grand mean, *i.e.*, the average across all languages. Thus, if, for some number of commits, a particular project developed in an *average* language had four defective commits, then the choice to use C++ would mean that we should expect one additional buggy commit since $e^{0.23} \times 4 = 5.03$. For the same project, choosing Haskell would mean that we should expect about one fewer defective commit as $e^{-0.23} \times 4 = 3.18$. The accuracy of this prediction is dependent on all other factors remaining the same, a challenging proposition for all but the most trivial of projects. All observational studies face similar limitations and we address this concern in more detail in section 5.

> **Result 1:** *Some languages have a greater association with defects than other languages, although the effect is small.*

In the remainder of this paper we expand on this basic result by considering how different categories of application, defect, and language, lead to further insight into the relationship between languages and defect proneness.

Software bugs usually fall under two broad categories: (1) *Domain Specific bug*: these are specific to project functionalities and do not depend on the underlying programming language. For example, we find a bug fix in Linux with log message: "Fix headset mic support for Asus X101CH". The bug was due to a missing functionality [25] in Asus headset[3] and less to do with language feature. Previous researches term these errors as *Software Component* bug [18, 28]. (2) *Generic bug*: these are more generic in nature and have less to do with project functionalities. For example, type-error, concurrency error etc.

Consequently, it is reasonable to think that the interaction of application domain and language might directly impact the number of defects within a project. It is often believed that some languages excel at some tasks more so than others, *e.g.,* `C` for low level work, or `Java` and `C#` for user applications. A less than ideal language for an application might lead to a greater number of defects within the project. Thus, to compare error proneness of different languages we ideally should ignore the domain specific bugs and focus only on the generic ones since they are more likely to depend on the programming language feature. However, since a domain specific bug may also arise due to a generic programming error, it is difficult to separate the two. A possible workaround is to study the defect proneness of languages while controlling the domain. To this end, we would like to confirm specific interactions between language and domain with respect to defects within projects. Statistically, this goal poses challenges for regression. Interaction terms in a regression model might yield some insight, however, with 17 languages across 7 domains this will yield an excessive number of terms and a challenging, and most likely spurious, interpretation.

Given this, we first consider testing for the dependence between domain and language usage within a project, using a Chi-Square test of independence. Unfortunately, out of 119 cells in our data set, 46, *i.e.* 39%, are below the value of 5. This exceeds the recommendation that no more than 20% of the counts should be below 5 [19]. We include the value here for completeness[4], however, the low strength of association of 0.191 as measured by Cramer's V, suggests that although there is likely some relationship between domain and language in our data set, including domain in regression models is unlikely to lead to meaningful models.

One option to address this concern would be to remove languages or combine domains, however, our data here presents no clear choices. Alternatively, we could combine languages; this choice leads to a related but slightly different question. Rather than considering languages individually, we aggregate them by language class, as described in Section 2.3, and analyze the relationship between defects and language class.

RQ2. Which language properties relate to defects?

We have seen that language choice has some impact on defects and in this section we want to discuss in detail how language properties, as described in Section 2.3, relate to defects. Broadly, each of these properties divides languages along some line that is often discussed in the context of errors, drives user debate, or have been the subject of prior work. To arrive at the six factors in the model we combined all of these factors across all of the languages in our study.

Ideally, we would want to include each of the separate properties in the regression model so that we can assert with some assurance that a particular property, while holding others constant, is more or less responsible for particular defects. Unfortunately, however, the properties are highly correlated and models with all properties are not stable. To avoid this issue in this study we model the impact of the six different language classes, which result from combining the language properties, on the number of defects while controlling for the same basic covariates that we used in the model in *RQ1*.

As with language, we are comparing language classes with the average behavior across all languages. The model is presented in Table 7. As with the previous model, the first four variables are control variables and we are not interested in their impact on the outcome other than to say, in this case, that, with the exepction of size, they are all positive and significant.

Looking at the coefficients for language class we can see that the coefficient with smallest magnitude belongs to `Proc Static-Strong-Managed`. The coefficient is insignificant which implies that the z-test for the coefficient cannot distinguish the coefficient from zero. Given the magnitude of the standard error, however, we can reasonably assume that the lack of significance is not related to wide variance or insufficient data, rather, it is because the behavior of languages in this class is very close to the average behavior across all languages. We confirm this by recoding the coefficient using `Proc Static-Weak-Unmanaged` as the base level and employing treatment, or dummy coding that compares each language class with the base level. In this case, `Proc Static-Strong-Managed` is significantly different with $p < 2e - 16$. We note here that while choosing different coding methods affects the coefficients and z-scores, the models are identical in all other respects. When we change the coding we are rescaling the coefficients to reflect the comparison that we wish to make [7]. Comparing the other language classes to the grand mean, `Proc Static-Weak-Unmanaged` languages are more likely to induce defects.

| Defective Commits Model	Coef.	Std. Err.
(Intercept)	-1.93	$(0.10)^{***}$
log commits	2.26	$(0.03)^{***}$
log age	0.11	$(0.03)^{**}$
log size	0.05	$(0.02)^{*}$
log devs	0.16	$(0.03)^{***}$
C	0.15	$(0.04)^{***}$
C++	0.23	$(0.04)^{***}$
C#	0.03	(0.05)
Objective-C	0.18	$(0.05)^{***}$
Go	-0.08	(0.06)
Java	-0.01	(0.04)
CoffeeScript	0.07	(0.05)
JavaScript	0.06	$(0.02)^{**}$
TypeScript	-0.43	$(0.06)^{***}$
Ruby	-0.15	$(0.04)^{*}$
Php	0.15	$(0.05)^{***}$
Python	0.10	$(0.03)^{**}$
Perl	-0.15	(0.08)
Clojure	-0.29	$(0.05)^{***}$
Erlang	-0.00	(0.05)
Haskell	-0.23	$(0.06)^{***}$
Scala	-0.28	$(0.05)^{***}$

$^{***}p < 0.001, ^{**}p < 0.01, ^{*}p < 0.05$

Table 6: Some languages induce fewer defects than other languages. Response is the number of defective commits. Languages are coded with weighted effects coding so each language is compared to the grand mean. *AIC=10673, BIC=10783, Log Likelihood = -5315, Deviance=1194, Num. obs.=1114*

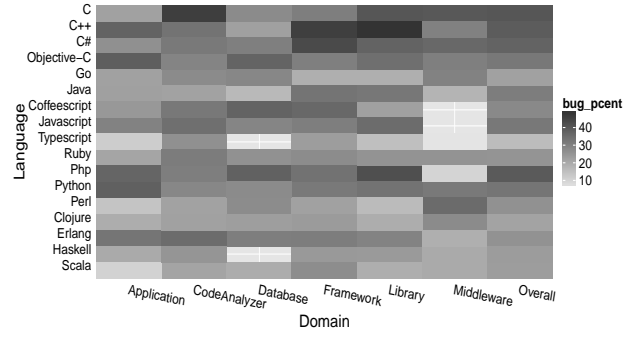[3] https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1169138
[4] Chi-Squared value of 243.6 with 96$d.f.$ and $p = 8.394e - 15$

Figure 1: Interaction between language's defect proneness and domain
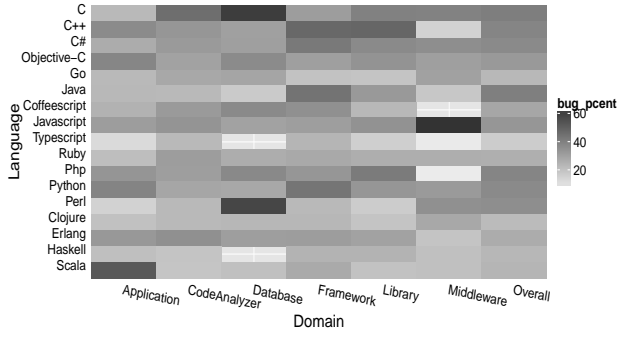
(a) Variation of defect proneness across languages for a given domain

(b) Variation of defect proneness across languages for a given domain after removing the outliers

Each cell in the heat map represents defect proneness of a language (row header) for a given domain (column header). The 'Overall' column represents defect proneness of a language over all the domains. The cells with white cross mark indicate null value, *i.e.* no commits were made corresponding to that cell.

This implies that either weak typing or memory management issues contribute to greater defect proneness as compared with other procedural languages.

Among scripting languages we observe a similar relationship between weak and strong typing. Script languages with strong typing are less related to defects whereas script languages with weak typing show a stronger relationship. This is some evidence that weak vs strong typing is more likely responsible for this difference as opposed to memory management, we cannot, of course, state this conclusively given the correlation between factors. It is possible that among script languages strong versus weak typing is driving the relationship and among procedural languages, memory management is driving the difference. However, as a group, strongly typed languages are less error prone than average while the weakly typed languages are more error prone than the average.

Another interesting observation that we can take from this model is that the coefficients for scripting languages are more negative than those for procedural languages. This could be driven by language class, or, it could be driven by the fact that the scripting languages are dynamically typed vs the static typing of procedural languages. The contrast between static and dynamic typing is also visible in functional languages.

The functional languages as a group show a strong difference

Defective Commits		
(Intercept)	−1.93	$(0.10)^{***}$
log commits	2.29	$(0.03)^{***}$
log age	0.13	$(0.03)^{***}$
log size	0.01	(0.01)
log devs	0.19	$(0.03)^{***}$
Functional-Dynamic-Strong-Managed	−0.16	$(0.04)^{***}$
Functional-Static-Strong-Managed	−0.27	$(0.04)^{***}$
Proc-Static-Strong-Managed	−0.02	(0.03)
Proc-Static-Weak-Unmanaged	0.18	$(0.02)^{***}$
Script-Dynamic-Strong-Managed	−0.09	$(0.02)^{***}$
Script-Dynamic-Weak-Managed	0.06	$(0.02)^{***}$
$^{***}p < 0.001, ^{**}p < 0.01, ^{*}p < 0.05$		

Table 7: Functional languages have a smaller relationship to defects than other language classes where as prodecural languages are either greater than average or similar to the average. Language classes are coded with weighted effects coding so each language is compared to the grand mean. *AIC=10739, BIC=10794, Log Likelihood = -5358, Deviance=1187, Num. obs.=1114*

from the average. Compared to all other language types, both `Functional-Dynamic-Strong-Managed` and `Functional--Static-Strong-Managed` languages show a smaller relationship with defects. Statically typed languages have substantially smaller coefficient yet both functional language classes have the same standard error. This is strong evidence that functional static languages are less error prone than functional dynamic languages, however, the z-tests only test whether the coefficients are different from zero. In order to strengthen this assertion we recode the model as above using treatment coding and observe that the `Functional-Static-Strong-Managed` language class is significantly less defect prone than the `Functional-Dynamic--Strong-Managed` language class with $p = 0.045$.

As with the relationship between language and defects, the relationship between language class and defects is based on a small effect. The deviance explained is shown in the anova table below and is similar, although smaller, to the deviance related to language and, consequently, has a similar interpretation.

	Df	Deviance	Resid. Df	Resid. Dev	Pr(>Chi)
NULL			1113	34910.02	
log commits	1	33491.18	1112	1418.84	0.0000
log age	1	38.96	1111	1379.88	0.0000
log size	1	10.94	1110	1368.94	0.0009
log devs	1	44.80	1109	1324.14	0.0000
Lang. Class	5	136.87	1104	1187.28	0.0000

Having discussed the relationship between language class and defects, we revisit the question of application domain.As before we ask whether domain has an interaction with language class. Does the choice of, *e.g.,* a functional language, have an advantage for a particular domain? For this pair of factors, the contingency table confirms to assumptions. As above, a Chi-Square test for the relationship between these factors and the project domain yields a value of 99.0494 and $df = 30$ with $p = 2.622e-09$ allowing us to reject the null that the factors are independent. Cramer's-V yields a value of 0.133, a weak level of association. Consequently, although there is some relation between domain and language, there is only a weak relationship between domain and language class.

Result 2: *There is a small but significant relationship between language class and defects. Functional languages have a smaller relationship to defects than either procedural or scripting languages.*

It is somewhat unsatisfying that we do not observe a strong association between language, or language class, and domain within a project. An alternative way to view this same data is to aggregate defects over all languages and domains, disregarding the relationship to projects. Since we cannot view this data as independent samples, we do not attempt to analyze it statistically, rather we take a descriptive, visualization based approach.

We define *Defect Proneness* as the ratio of bug fix commits over total commits made in a given language for a given domain. Figure 1 illustrates the interaction between domain and language using a heat map, where the defect proneness increases from lighter to darker zone. Each cell in the heat map represents defect proneness of a language (row header) for a given domain (column header). We are interested in determining which language factors influence defect fixing commits across a collection of projects written across a variety of different languages. This leads to the following research question:

RQ3. Does language defect proneness depend on domain?

A first glance at Figure 2(a) reveals that defect proneness of the languages indeed depends on the domain. For example, in the Middleware domain `JavaScript` is most defect prone, with a defect proneness value of 31.06%. This was little surprising to us since `JavaScript` is typically not used for Middleware domain. On a closer look, we find that `JavaScript` has only one project, v8 (Google's `JavaScript` virtual machine), in Middleware domain that is responsible for all the errors.

Also, `Scala` is most defect prone in Application domain with a defect density, defined as number of bug fix commit over total commit in a project, of 52.67%. However, there are only two projects written in `Scala` in the Application domain: *metrics* and *zipkin*. Out of that *zipkin* has a defect density of 80%. Similarly, `Perl` and `C` are the two top most buggy languages in Database domain with defect proneness 58.63% and 61.47% respectively. However, we find that *mysql* project is solely responsible for such high defect count, where overall defect density of mysql is 70.25%.

Thus, it turns out that variation of defect density of the domains with languages may be an attribute of individual projects. To verify this, we re-evaluate domain-language interaction after ignoring the outliers. We filter out the projects that have defect density below 10 percentile and above 90 percentile. Figure 2(b) shows the result. Note that, the outliers' effects are also controlled in all our regression models as we filter them out as high leverage points, as discussed in Section 2.6.

The variation of defect proneness and languages per domain is much subdued in the new heat map. The remaining variation comes from the inherent defect proneness of the languages, as we have seen in RQ1. To ensure this, we measure the pairwise rank correlation (Spearman Correlation [32]) between the language defect proneness for each domain with the Overall. For all the domains, the correlation is positive, and p-values are significant (< 0.01) except for Database domain. This shows that *w.r.t.* defect proneness, language ordering in each domain (except Database) is correlated with the overall language ordering. Hence, domain has less significance when it comes to defect proneness of the languages.

	APP	CA	DB	FW	LIB	MW
Spearman Corr.	0.71	0.56	0.30	0.76	0.90	0.46
p-value	0.00	0.02	0.28	0.00	0.00	0.09

> **Result 3:** *Our findings indicate that there is no general relationship between domain and language defect proneness.*

We have shown that different languages induce a larger number of defects and that this relationship is not only related to particular languages but holds for general classes of languages, however, we find that the type of project doesn't mediate this relationship to a large degree. We now turn our attention to categorization of the response. We want to understand how language relates to specific kinds of defects and how this relationship compares to the more general relationship that we observe. We divide the defects into categories as described in Table 5 and ask the following question:

RQ4. What is the relation between language and bug category?

We use a mixed-method analysis to understand the relation between languages and bug categories. First, using a descriptive, visualization based approach similar to RQ3, we study the relation between bug categories and language class. We generate a heat map (see Figure 2) that aggregates defects over language classes and bug types and illustrate an overall relationship between language class and the bug categories.

Next, to understand the interaction between bug categories with individual languages, we rerun the negative binomial regression model as discussed in RQ1, separately for each bug category. For a given category, the predictor of the model is languages encoded with weighted effects codes and the independent variable is the number of defect fixing commits belong to that category; commit age, total commits, raw project size, and total developers are the control factors. For each language, regression coefficient compares its number of bug fix commits in a bug category to the grand mean of weighted mean of all languages in all projects. The results are shown in Table 8. For these models we present only a subset of the anova coefficients as the distribution of deviance explained is very similar to the language model presented in Table 6 and its associated anova table. The number of commits still commands the largest proportion of the deviance, and the deviance explained by the remaining control variables is substantially smaller than that explained by the language. The overall deviance for each model is substantially smaller, however, and the proportion explained by language for a specific defect type is similar in magnitude for most of the categories. We interpret this relationship to mean that language as a greater impact on specific categories of bugs, than it does on bugs overall.
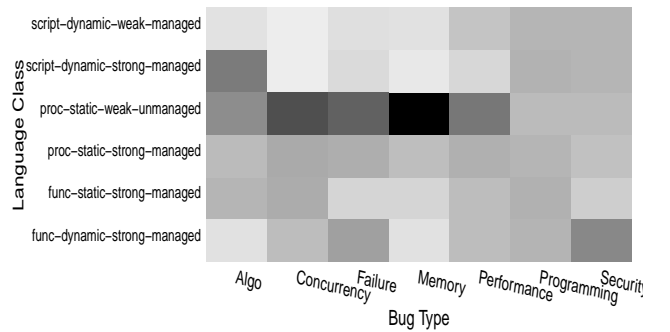

Figure 2: Relation between bug categories and language class
Each value represents percentage of bug fix commit out of all bug fix commits in a given language class (row header) belong to a bug category (column header). The values are then normalized column wise.

In the following section, we elaborate the result for bug categories with significant bug count as reported in Table 5. However, our conclusion generalizes for all categories.

Programming Errors. Generic programming errors account for

	Memory		Concurrency		Security		Failure	
(Intercept)	−7.45	$(0.45)^{***}$	−8.19	$(0.75)^{***}$	−7.65	$(0.56)^{***}$	−6.41	$(0.40)^{***}$
log commits	2.32	$(0.12)^{***}$	2.60	$(0.21)^{***}$	2.04	$(0.15)^{***}$	2.03	$(0.11)^{***}$
log age	0.31	$(0.15)^{*}$	0.37	(0.24)	0.74	$(0.18)^{***}$	0.14	(0.13)
log size	0.00	(0.09)	−0.23	(0.16)	−0.07	(0.11)	0.30	$(0.08)^{***}$
log devs	0.18	(0.10)	0.25	(0.17)	0.46	$(0.14)^{***}$	−0.03	(0.10)
C	1.81	$(0.11)^{***}$	0.58	$(0.22)^{**}$	0.51	$(0.17)^{**}$	0.67	$(0.12)^{***}$
C++	1.14	$(0.10)^{***}$	1.18	$(0.18)^{***}$	0.49	$(0.15)^{**}$	1.15	$(0.11)^{***}$
C#	−0.05	(0.17)	0.93	$(0.24)^{***}$	−0.30	(0.22)	−0.02	(0.16)
Objective-C	1.47	$(0.15)^{***}$	0.52	(0.29)	−0.04	(0.23)	1.16	$(0.15)^{***}$
Go	0.00	(0.25)	1.73	$(0.30)^{***}$	0.55	$(0.27)^{*}$	−0.38	(0.24)
Java	0.59	$(0.14)^{***}$	0.97	$(0.22)^{***}$	0.14	(0.18)	0.32	$(0.13)^{*}$
CoffeeScript	−0.36	(0.23)	−1.64	$(0.55)^{**}$	−0.26	(0.26)	0.00	(0.19)
JavaScript	−0.11	(0.10)	−0.12	(0.17)	−0.02	(0.12)	−0.16	(0.09)
TypeScript	−1.32	$(0.40)^{**}$	−2.15	$(0.98)^{*}$	−1.34	$(0.41)^{**}$	−0.34	$(0.07)^{***}$
Ruby	−1.12	$(0.20)^{***}$	−0.82	$(0.30)^{**}$	−0.24	(0.20)	−0.34	$(0.16)^{*}$
Php	−0.63	$(0.17)^{***}$	−1.64	$(0.35)^{***}$	0.27	(0.19)	−0.61	$(0.17)^{***}$
Python	−0.44	$(0.14)^{**}$	−0.21	(0.23)	0.25	(0.16)	−0.07	(0.13)
Perl	0.41	(0.36)	−0.86	(0.86)	−0.16	(0.43)	−0.40	(0.40)
Scala	−0.41	$(0.18)^{*}$	0.73	$(0.25)^{**}$	−0.16	(0.22)	−0.91	$(0.19)^{***}$
Clojure	−1.16	$(0.27)^{***}$	0.10	(0.30)	−0.69	$(0.26)^{**}$	−0.53	$(0.19)^{**}$
Erlang	−0.53	$(0.23)^{*}$	0.76	$(0.29)^{**}$	0.73	$(0.22)^{***}$	0.65	$(0.17)^{***}$
Haskell	−0.22	(0.20)	−0.17	(0.32)	−0.31	(0.26)	−0.38	(0.19)
AIC	3033.55		2234.29		3411.57		4188.71	
BIC	3143.90		2344.63		3521.92		4299.06	
Log Likelihood	-1494.77		-1095.14		-1683.79		-2072.36	
Deviance	905.38		666.41		921.50		1070.60	
Num. obs.	1114		1114		1114		1114	
Residual Deviance (NULL)	8508.6		3426.5		3372.5		6071.5	
Language Type Deviance	587.2		157.03		61.40		303.8	

$^{***}p < 0.001, ^{**}p < 0.01, ^{*}p < 0.05$

Table 8: While the impact of language on defects varies across defect category, language has a greater impact on specific categories than it does on defects in general. For all models above the deviance explained by language type has $p < 3.031e − 07$.

around 88.53% of all bug fix commits and occur in all the languages classes. The regression analysis draws a similar conclusion as of RQ1 (see Table 6) since programming errors represent the majority of the studied fixes. All languages incur programming errors such as faulty error-handling, faulty object and variable definitions, incorrect data initialization, typos, *etc.*. Sill, some programming errors are more language-specific. For example, we find 122 runtime errors in JavaScript that are not present in TypeScript[5]. In contrast, TypeScript has more type related errors, since the TypeScript compiler flags them during development. Thus, although generic programming errors occur in all languages, some kinds relate to specific language features found only in some languages.

Memory Errors. Memory errors count for 5.44% of all the bug fix commits. The heat map in Figure 2 shows a strong relationship between Proc-Static-Weak-Unmanaged class and memory errors. This is obvious as languages with unmanaged memory type are known for memory bugs. Regression analysis in Table 8 also confirms that languages with unmanaged memory type, *e.g.,*C, C++, and Objective-C introduce more memory errors than the average with statistical significance. Among the managed languages, Java has significantly more memory errors than the average, though its regression coefficient is less than the unmanaged languages. Though Java has its own garbage collector, having memory leak is not surprising since unused object references often prevents the garbage collector from reclaiming the memory [16]. In fact, we notice 28.89% of all the memory errors in Java comes from a memory leak. In terms of effect size, language has a larger impact on memory defects than all other *cause* categories.

Concurrency Errors. 1.99% of total bug fix commits are related to Concurrency Errors. The heat map indicates that Proc--Static-Weak-Unmanaged class dominates concurrency er-

rors. In this class, C and C++ introduce 19.15% and 7.89% of the Concurrency errors, and they are distributed across the projects. The regression result of Table 8 also confirms that with significant p-value. Language classes Proc-Static-Strong-Managed and Functional-Static-Strong-Managed are also in the darker zone in the heat map. The major languages that contribute concurrency errors from these classes are Go, C#, and Scala and their regression coefficient is also statistically significant. These results confirm, in general static languages produce more concurrency errors than others. Among the dynamic languages, only Erlang is more prone to concurrency errors. The regression analysis also shows that projects written in dynamic languages like CoffeeScript, TypeScript, Ruby, and Php have fewer concurrency errors (note statistically significant negative coefficients, in Table 8). A textual analysis based on word-frequency of the bug fix messages suggests that most of the concurrency errors occur due to race condition, deadlock, or incorrect synchronization, as shown in the table below.

	C	C++	C#	Java	Scala	Go	Erlang
race	63.11	41.46	77.7	65.35	74.07	92.08	78.26
deadlock	26.55	43.36	14.39	17.08	18.52	10.89	15.94
SHM	28.78	18.24	9.36	9.16	8.02	0	0
MPI	0	2.21	2.16	3.71	4.94	1.98	10.14

In all the languages, race condition is most frequent, ranging from 41% in C++ to 92% is Go. The enrichment of race condition errors in Go is likely because the Go is distributed with a race-detection tool [6], which may advantage Go developers in detecting races. Deadlocks are also noteworthy, ranging from 43.36% in C++ to 10.80% in Go. The synchronization errors are mostly related to message passing (MPI) or shared memory operation (SHM). Erlang and Go use MPI [12, 3] (which do not require locking of shared resources) for inter-thread communication, which explains why these two languages do not have any SHM related errors like

[5]TypeScript is a JavaScript like language with support of static typing

[6] http://talks.golang.org/2013/advconc.slide#20

locking, mutex etc. In contrast, projects in the other languages use SHM primitives for communication and can thus have locking-related errors.

Security and Other Impact Errors. Around 7.33% of all the bug fix commits is related to Impact errors. Among them `Erlang`, `C`, `C++`, and `Go` produce more security errors than average, as confirmed by the regression model (Table 8). The regression also suggests that projects written in `TypeScript` and `Clojure` are less likely to introduce a security error than average. (Figure 2). From the heat map we also see that `Static` languages are in general more prone to failure and performance errors, followed by `Functional-Dynamic-Strong-Managed` languages. In the later category, `Erlang` is more prone to induce a failure (positive regression coefficient with statistical significance). The analysis of deviance results confirm that language is strongly associated with failure impacts. While security errors are the weakest among the categories, with respect to the residual deviance of the model, the deviance explained by language is still quite strong.

> **Result 4:** *Defect types are strongly associated with languages; Some defect type like memory error, concurrency errors also depend on language primitives. Language matters more for specific categories than it does for defects overall.*

4. RELATED WORK

Prior work on programming language comparison falls in 3 categories: (1) *Controlled Experiment*: For a given task, developers are monitored while programming in different languages. Researchers then compare outcomes such as development effort and program quality. Hanenberg *et al.* [13] compared static vs. dynamic typing by monitoring 48 programmers for 27 hours while developing a parser program. They found no significant difference in code quality between the two, but that their dynamic type-based language have shorter development time. Their study was conducted with undergraduate students in a lab setting with custom-designed language and IDE. Our study, by contrast is a field study of popular software applications. While we can only indirectly (and *post facto*) control for confounding factors using regression, we benefit from much larger sample sizes, and more realistic, widely-used software. We find that statically typed languages in general are less defect prone than the dynamic types, and that strong typing is better than weak typing in the same regard. The effect sizes are modest; it could be reasonably argued that they are visible here precisely because of the large sample sizes.

Harrison *et al.* [14] compared `C++`, a procedural language, with `SML`, a functional language, and noticed no significant difference in total number of errors, although `SML` has higher defect density than `C++`. `SML` was not represented in our data, which does, however, suggest that functional languages are in general less defect prone than procedural languages. Another line of work primarily focus on comparing development effort across different languages [17, 23]. However, they do not analyze language defect proneness.
(2) *Surveys*: Meyerovich *et al.* survey developers' perception of programming languages, to study why some languages are more popular than others[20]. They report strong influence from non-linguistic factors: prior language skills, availability of open source tools, and existing legacy systems. Such factors also arise in our findings: we confirm that availability of external tools also impact software quality; `Go` has lot more concurrency bugs related to race condition due to its race condition detection tool (see RQ4 in Section 3).
(3) *Repository Mining*: Bhattacharya *et al.* [4] study four projects developed in both `C` and `C++` and find that the software compo-

nents developed in `C++` are in general more reliable than `C`. We find that both `C` and `C++` are more defect prone than average defect proneness of all the studied languages, although `C++` has a higher regression coefficient (0.23) than `C` (0.15) (see Table 6). However, for certain bug types like concurrency errors, `C` is more defect prone than `C++` (see RQ4 in Section 3).

5. THREATS TO VALIDITY

We recognize few threats to our reported results. First, to identify bug fix commits we did not check the bug database; instead we rely on the keywords that developers often use to indicate a bug fix commit. Our choice was deliberate. We wanted to capture the issues that developers continuously face in an ongoing development process, not just the reported bugs. However, such choice possesses threats of over estimation. Our categorization of the domains is subjected to interpreter's bias, although another member of our group verified the categories. Also, our effort to categorize a large number of bug fix commits could potentially raise some questions. Especially, the categorization can be tainted by the initial choice of keywords. To mitigate the threat, we evaluate our classification against manual annotation as discussed in Section 2.5. Our interpretation of language classes, as discussed in Section 2.3 can also raise questions. For instance, we classify `Objective-C` with unmanaged memory type, although it follows a hybrid memory model. Similarly, we annotate `Scala` as functional and `C#` as procedural, although they give users choice of both procedural and functional design [22, 24]. Finally, we associate defect fixing commits to language properties, although they could reflect reporting style or other developer properties. Availability of external tools or libraries may also impact the extent of bugs associated with a language.

6. CONCLUSION

We have presented a "big data" study of language type and use, as it relates to software quality. More than just the quantity, though, the Github data we used is characterized by its complexity, especially the variance along multiple dimensions of language, language type, usage domain, amount of code, sizes of commits, and the various characteristics of the many issue types.

The large sample sizes afforded by the Github "big data" allows us to perform a mixed-methods study of the effects of language, while controlling for a number of confounds. Through a combination of regression modeling, text analytics, and visualization, we have examined the interactions of language, domain, and defect type. The data indicates functional languages are better than procedural languages; it suggests that strong typing is better than weak typing; that static typing is better than dynamic; and managed memory usage is better than unmanaged. Further, defect proneness of languages in general are not associated with software domains. Also, languages are more related to individual bug categories than bugs overall.

On the other hand, even big data becomes small and insufficient when it is sliced and diced many ways simultaneously, *i.e.* when the underlying connectivity between variables is rich. The implications are that the more dependent variables there are, the more difficult it becomes (vis-a-vis the amount of data available) to answer questions about a specific variable's effect on any outcome where interactions with other variables exist. Hence, our inability to quantify the specific effects of language type on usage. Additional methods such as surveys could be helpful here. Addressing these challenges remains for future work.

Acknowledgements

7. REFERENCES

[1] Github archive, https://githubarchive.org/.

[2] Google big query, https://developers.google.com/bigquery/.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. Concurrent programming in erlang. 1993.

[4] P. Bhattacharya and I. Neamtiu. Assessing programming language impact on development and maintenance: A study on c and c++. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 171–180, New York, NY, USA, 2011. ACM.

[5] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.

[6] D. M. Blei. Probabilistic topic models. *Communications of the ACM*, 55(4):77–84, 2012.

[7] J. Cohen. *Applied multiple regression/correlation analysis for the behavioral sciences*. Lawrence Erlbaum, 2003.

[8] H. CRAMÉR et al. *Mathematical methods of statistics.* Princeton University Press, 1946.

[9] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008.

[10] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *Software Engineering, IEEE Transactions on*, 27(7):630–650, 2001.

[11] GitHub. Linguist: https://github.com/github/linguist.

[12] Google. http://golang.org/doc/effective_go.html#concurrency.

[13] S. Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 22–35, New York, NY, USA, 2010. ACM.

[14] R. Harrison, L. Smaraweera, M. Dobie, and P. Lewis. Comparing programming paradigms: an evaluation of functional and object-oriented programs. *Software Engineering Journal*, 11(4):247–254, 1996.

[15] D. E. Harter, M. S. Krishnan, and S. A. Slaughter. Effects of process maturity on quality, cycle time, and effort in software product development. *Management Science*, 46(4):451–466, 2000.

[16] M. Jump and K. S. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. In *ACM SIGPLAN Notices*, volume 42, pages 31–38. ACM, 2007.

[17] S. Kleinschmager, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik. Do static type systems improve the maintainability of software systems? an empirical study. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 153–162. IEEE, 2012.

[18] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, October 2006.

[19] J. P. Marques De Sá. Applied statistics using spss, statistica and matlab. 2003.

[20] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 1–18. ACM, 2013.

[21] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*, page 120. IEEE Computer Society, 2000.

[22] M. Odersky, L. Spoon, and B. Venners. *Programming in scala*. Artima Inc, 2008.

[23] V. Pankratius, F. Schmidt, and G. Garretón. Combining functional and imperative programming for multicore software: an empirical study evaluating scala and java. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 123–133. IEEE Press, 2012.

[24] T. Petricek and J. Skeet. *Real World Functional Programming: With Examples in F# and C#*. Manning Publications Co., 2009.

[25] A. A. Porter and L. G. Votta. An experiment to assess different defect detection methods for software requirements inspections. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 103–112, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[26] D. Posnett, C. Bird, and P. Dévanbu. An empirical study on the influence of pattern roles on change-proneness. *Empirical Software Engineering*, 16(3):396–423, 2011.

[27] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[28] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 2013.

[29] V. Vapnik. *The nature of statistical learning theory*. springer, 2000.

[30] Q. Vuong. Likelihood ratio tests for model selection and non-nested hypotheses. *Econometrica: Journal of the Econometric Society*, pages 307–333, 1989.

[31] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008.

[32] J. H. Zar. Significance Testing of the Spearman Rank Correlation Coefficient. *Journal of the American Statistical Association*, 67(339):578–580, 1972.