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

Baishakhi Ray, Daryl Posnett, Vladimir Filkov, Premkumar 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 (728 projects, 63 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 other, intangible 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 may appear to be tinged with an almost religious fervor, most agree that programming language choice can impact both the coding process and the resulting artifact.

Advocates of strong static typing tend to believe that an ounce of prevention is worth a pound of cure and that the static approach catches defects early. Dynamic typing advocates argue, however, that conservative static type checking is wasteful of developer resources, and that it is better to rely on strong dynamic type checking to catch type errors as they arise. These debates, however, have largely been of the armchair variety, often the supporting evidence tends to be anecdotal.

This is perhaps not unreasonable; obtaining empirical evidence to support such claims is a challenging task given the number of other factors that influence software engineering outcomes, such as code quality, language properties, and usage domains. Considering software quality, for example, there are a number of well-known influential factors, including code size [6], team size [2], and age/maturity [9], and indeed, such process factors can effectively predict defect localities [25].

Controlled experiments are certainly one approach to examining the impact of language choice in the face of such daunting confounds, however, owing to cost, such studies typically introduce a confound of their own, *i.e.*, limited scope. The tasks completed in such studies are necessarily limited and seldom compare to a typical *real world* application. There have been several such studies recently that use students, or compare languages with static or dynamic typing through an experimental factor [7, 15, 12].

If we want to study this problem from an observational point of view, we need a large body of software projects to study. GitHub contains many projects in multiple languages that vary substantially across size, age, and number of developers. Each project repository provides a historical record from which we extract contribution history, project size, authorship, and defect repair. We then use a variety of tools to study the effects of language features on defect occurrence. Our approach is best described as mixed-methods, or triangulation [5] approach; we use text analysis, clustering, and visualization to confirm and support the findings of a quantitative regression study.

While the use of regression analysis to deal with confounding variables is not without controversy, we submit that the use of a large sample size and the triangulation of our results from qualitative methods increase the credibility of our quantitative results.

2. METHODOLOGY

Our methods are typical of large scale observational studies in software engineering. We first gather our data from several sources using largely automated methods. We then filter and clean the data in preparation for building a statistical model. We further validate the model using qualitative methods. Filtering choices are driven by a combination of factors including the nature of our research questions, the quality of the data and beliefs about which data is most suitable for statistical study. In particular, GitHub contains many projects written in a large number of programming languages. For this study, we focused our data collection efforts on the most popular projects written in the most popular languages. We choose statistical methods appropriate for evaluating the impact of factors on count data.

2.1 Data Collection

We choose the top 19 programming languages from GitHub. We

disregard CSS, Shell script, and Vim script as they are not considered to be general purpose languages. We further include `TypeScript`, a typed superset of `JavaScript`. Then, for each of the studied languages we retrieve the top 50 projects that are primarily written in that language. Table 1 shows the top three projects in each language, based on their popularity. In total, we analyze 850 projects spanning 17 different languages.

Our language and project data was extracted from the *GitHub Archive*, 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 service to provide an interface for interactive data analysis.

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`	typescript-node-definitions, StateTree, typescript.api
`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

Identifying top languages. We 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. Since multiple languages are often used to develop a project, assigning a single language to a project is difficult. Github Archive stores information gathered fom GitHub Linguist which can measure the language distribution of a GitHub project repository by using the extensions of a project's source files. The language with the maximum number of source files is assigned as the *primary language* of the project.

Retrieving popular projects. For each selected language, we filter the project repositories written primarily in that language by its popularity based on the number of *stars* associated to that project. The number of starts indicates how many people have actively indicated an interest in the project. We assume that this is a reasonable indication of project popularity and select the top 50 projects in each language.

To ensure that these projects have a sufficient development history, we filter out the projects having fewer than 28 commits, where 28 is the first quartile commit number of all projects under consideration. This leaves us with 728 projects. Table 1 shows the top three projects in each language.

Retrieving project evolution history. For each of the 728 projects, we downloaded the non merged commits, commit logs, author date, and author name using the command: `git log -no-merges -numstat`. The numstat flag shows the number of added and deleted lines per file associated with each commit which we use 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. These variables are used as control variables in our regression model, 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 [18].

Table 2 summarizes our data set. Since a project may use multiple languages, the second 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, where 20 is the first quartile value of the total number of commits per project per language. For example, we find 220 projects that use more than 20 commits in `C`. This ensures that the studied languages have significant activity within the projects.

In summary, we study 728 projects developed in 17 languages with 18 years of parallel evolution history. This includes 29 thousand different developers, 1.57 million commits, and 564,625 bug fix commits.

2.2 Categorizing Languages

We define language classes based on several properties of the language thought to influence language quality [7, 8, 12], as shown in Table 3. The *Programming Paradigm* indicates whether the project is written in a procedural, functional, or scripting language.

Compilation Class indicates whether the project is statically or dynamically typed.

Type Class classifies languages based on strong and weak typing, based on whether the language admits *type-confusion*. We consider that a program introduces type-confusion when it attempts to interpret a memory region populated by a datum of specific type T1, as an instance of a different type T2 and T1 and T2 are not related by inheritance. We classify a language as *strongly typed* if it explicitly detects type confusion and reports it as such. Strong typing could happen by static type inference within a compiler (*e.g.,* with `Java`), using a type-inference algorithm such as Hendley-Milner [10, 17], or at run-time using a dynamic type checker. In contrast, a language is weakly-typed if type-confusion can occur silently (undetected), and eventually cause errors that are difficult to localize. For example, in a weakly typed language like `JavaScript` adding a string to a number is permissible (*e.g.,* '5' + 2 = '52'), while such an operation is not permitted in strongly typed `Python`. Also, `C` and `C++` are considered weakly typed since, due to type-casting, one can interpret a field of a structure that was an integer as a pointer.

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.

2.3 Identifying Project Domain

We classify the studied projects into different domains based on their features and function using a mix of automated and manual techniques. The projects in GitHub come with `project descriptions` and `Readme` files that describe their features. We used Latent Dirichlet Allocation(LDA) [3], a well-known topic analysis algorithm to analyze this text. 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 detect 30 distinct domains, *i.e.* topics, and estimate the prob-

Table 2: **Study Subjects**

	Project Details				Total Commits		BugFix Commits	
Language	#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	14	240	546	10/2012 to 2/2014	3,272	1,915	870	273
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	728	28,948	62,840	1/1996 to 2/2014	1,574,618	254,365	564,625	83,921

Table 3: **Different Types of Language Classes**

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

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

ability that each project belonging to each domain. For example, LDA assigned the *facebook-android-sdk* project to the following topic with high probability: $(0.042 * facebook + 0.010 * swank/slime + 0.007 * framework + 0.007 * environments. + 0.007 * transforming)$. Here, the text values are the topics and the numbers are their probability of belonging to that domain; for clarity, we only show the top 5 domains.

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. In order to assign a meaningful name to each domain, we manually inspect each of the thirty domains to identify project-name-independent, domain-identifying keywords.

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, CodeAnalyzer, 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 and so 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 research 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).

2.4 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, similar to Tan *et al.* [13, 26].

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.

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

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" [1] was caused due to a missing check (programming error), and the associated impact was a 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) 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 a 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% appear to be incorrect memory handling; 1.99% are concurrency bugs, and 0.11% are algorithmic errors. Analyzing the impact of the bugs, we find 2.01% are related to security vulnerability; 1.55% are performance errors, and 3.77% causes complete failure to the system. Our technique could not classify 1.04% of the bug fix messages in any Cause or Impact category; we classify these with the Unknown type.

[1] `https://lkml.org/lkml/2012/12/18/102`

2.5 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 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 [4].

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 [4]. We verify this by comparing transformed with non transformed data using the AIC and Vuong's test for non-nested models.

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 [4]. 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 [4]. Weighted effects codes allow us to compare each language to the average effect across all languages while compensating for the unevenness of language usage across the projects [24].

To test for the relationship between two factor variables we use a Chi-Square test of independence [14]. After confirming a depen-

dence we use Cramer's V, an $r \times c$ equivalent of the phi coefficient for nominal data, to establish an effect size.

3. RESULTS

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 compare the impact of each language on the number of defects with the average impact of all languages against defect fixing commits using a regression model. The model details are shown in Table 6.

We include some variables as controls for factors that will clearly influence the response. Project age is included as older projects will generally have a greater number of defect fixes. Trivially, the number of commits to a project will also impact the response. Additionally, the number of developers who touch a project and the raw size of the project are both expected to grow with project activity.

The sign and magnitude of the Estimate in the above model relates the predictors to the outcome. 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++`, `Objective-C`, `Php`, and `Python`. The languages `Clojure`, `Haskell`, `Ruby`, and `Scala`, all have negative coefficients implying that these languages are less likely than average to result in defect fixing commits.

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=10432, BIC=10542, Log Likelihood = -5194, Deviance=1156, Num. obs.=1076***

Defective Commits Model	Coef.	Std. Err.
(Intercept)	−2.04	$(0.11)^{***}$
log age	0.06	$(0.02)^{***}$
log size	0.04	$(0.01)^{***}$
log devs	0.06	$(0.01)^{***}$
log commits	0.96	$(0.01)^{***}$
`C`	0.11	$(0.04)^{**}$
`C++`	0.18	$(0.04)^{***}$
`C#`	−0.02	(0.05)
`Objective-C`	0.15	$(0.05)^{**}$
`Go`	−0.11	(0.06)
`Java`	−0.06	(0.04)
`CoffeeScript`	0.06	(0.05)
`JavaScript`	0.03	(0.03)
`TypeScript`	0.15	(0.10)
`Ruby`	−0.13	$(0.05)^{**}$
`Php`	0.10	$(0.05)^{*}$
`Python`	0.08	$(0.04)^{*}$
`Perl`	−0.12	(0.08)
`Clojure`	−0.30	$(0.05)^{***}$
`Erlang`	−0.03	(0.05)
`Haskell`	−0.26	$(0.06)^{***}$
`Scala`	−0.24	$(0.05)^{***}$

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

	Df	Deviance	Resid. Df	Resid. Dev	Pr(>Chi)
NULL			1075	25176.25	
log commits	1	4256.89	1071	1286.74	0.0000
log age	1	8011.52	1074	17164.73	0.0000
log size	1	10082.78	1073	7081.95	0.0000
log devs	1	1538.32	1072	5543.63	0.0000
language	16	130.78	1055	1155.96	0.0000

One should take care not to overestimate the impact of language on defects. While these relationships are statistically significant, the effects are quite small. As shown above in the analysis of deviance table, language accounts for less than one percent of the total deviance of the model. Note that all variables are significant, that is, all of the factors account for some of the variance in the number of defective commits. While the interpretation of percentage of deviance is roughly similar 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; About the best we can do is to observe that it is a small affect [4].

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 compared 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 $e^{0.18} \times 4 = 4.79$. For the same project, choosing `Haskell` would mean that we should expect about one fewer defective commit as $e^{-0.26} \times 4 = 3.08$. 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*: specific to project function 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 [23] in Asus headset[2] and less to do with language feature. Prior research term these errors as *Software Component* bugs [13, 26]. (2) *Generic bug*: more generic in nature that has less to do with project function. For example, type-errors, concurrency errors, etc.

Consequently, it is reasonable to think that the interaction of application domain and language might impact the number of defects within a project. Since some languages are believed to excel at some tasks more so than others, *e.g.,* `C` for low level work, or `Java` for user applications, making an inappropriate choice might lead to a greater number of defects. To study this we should ideally ignore the domain specific bugs as generic bugs are more likely to depend

[2] `https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1169138`

on the programming language featured. 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 languages while controlling the domain. Statistically, however, with 17 languages across 7 domains, the large number of terms would be challenging to interpret given the sample size.

Given this, we first consider testing for the dependence between domain and language usage within a project, using a Chi-Square test of independence. Of 119 cells, 46, *i.e.* 39%, are below the value of 5 which is too high. No more than 20% of the counts should be below 5 [14]. We include the value here for completeness[3], however, the low strength of association of 0.191 as measured by Cramer's V, suggests that any relationship between domain and language is small and that inclusion of domain in regression models would not produce 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.

RQ2. Which language properties relate to defects?

Rather than considering languages individually, we aggregate them by language class, as described in Section 2.2, and analyze the relationship between defects and language class. 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 is responsible for particular defects. The properties are highly correlated, however, and models with all properties are not stable. Hence, we model the impact of the six different factors on the number of defects while controlling for the same basic covariates that we used in the model in *RQ1*.

Table 7: **Functional languages have a smaller relationship to defects than other language classes where as procedural 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=10419, Deviance=1132, Num. obs.=1067***

Defective Commits		
(Intercept)	-2.13	$(0.10)^{***}$
log commits	0.96	$(0.01)^{***}$
log age	0.07	$(0.01)^{***}$
log size	0.05	$(0.01)^{***}$
log devs	0.07	$(0.01)^{***}$
`Functional-Static-Strong-Managed`	-0.25	$(0.04)^{***}$
`Functional-Dynamic-Strong-Managed`	-0.17	$(0.04)^{***}$
`Proc-Static-Strong-Managed`	-0.06	$(0.03)^{*}$
`Script-Dynamic-Strong-Managed`	0.001	(0.03)
`Script-Dynamic-Weak-Managed`	0.04	$(0.02)^{*}$
`Proc-Static-Weak-Unmanaged`	0.14	$(0.02)^{***}$

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

As with language, we are comparing language classes with the average behavior across all languages. The model is presented in Table 7. It's clear that `Script-Dynamic-Strong-Managed` class has the smallest magnitude coefficient. The coefficient is insignificant, *i.e.*, the z-test for the coefficient cannot distinguish the coefficient from zero. Given the magnitude of the standard error, however, we can can assume that 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, `Script-Dynamic-Strong-Managed` is significantly different with $p = 0.00044$. 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 [4]. Comparing the other language classes to the grand mean, `Proc-Static-Weak-Unmanaged` languages are more likely to induce defects. 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. This is some evidence that weak vs strong typing is more likely responsible for this difference as opposed to memory management, we cannot state this conclusively given the correlation between factors. 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. The contrast between static and dynamic typing is also visible in functional languages.

The functional languages as a group show a strong difference 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 $p = 0.034$.

	Df	Deviance	Resid. Df	Resid. Dev	Pr(>Chi)
NULL			1066	32995.23	
log commits	1	31634.32	1065	1360.91	0.0000
log age	1	51.04	1064	1309.87	0.0000
log size	1	50.82	1063	1259.05	0.0000
log devs	1	31.11	1062	1227.94	0.0000
Lang. Class	5	95.54	1057	1132.40	0.0000

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 above and is similar, although smaller, to the deviance related to language and, consequently, has a similar interpretation.

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 hypothesis 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.

[3]Chi-Squared value of 243.6 with $96 d.f.$ and $p = 8.394e-15$

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. However, 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 per language per domain. Figure 1 illustrates the interaction between domain and language using a heat map, where the defect proneness increases from lighter to darker zone. We investigate which language factors influence defect fixing commits across a collection of projects written across a variety of languages. This leads to the following research question:

RQ3. Does language defect proneness depend on domain?

A first glance at Figure 1(a) reveals that defect proneness of the languages indeed depends on the domain. For example, in the Middleware domain `JavaScript` is most defect prone (31.06% defect proneness). This was little surprising to us since `JavaScript` is typically not used for Middleware. On a closer look, we find that `JavaScript` has only one project, v8 (Google's `JavaScript` virtual machine), in the Middleware domain that is responsible for all of the errors. This pattern repeats for several other domains which suggests 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 observed outliers. We filter out the projects that have defect density below 10 percentile and above 90 percentile. Figure 1(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 2.5.

The previously observed variation is subdued in the new heat map and what remains is a result of the inherent defect proneness of the languages, as we have seen in RQ1. To validate this, we measure the pairwise rank correlation between the language defect proneness for each domain with the Overall. For all of the domains, the correlation is positive, and p-values are significant (< 0.01) except for the Database domain. Thus, *w.r.t.* defect proneness, with the exception of the Database domain, language ordering in each domain is strongly correlated with the overall language ordering.

	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: *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 & 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. A heat map (see Figure 2) shows aggregated defects over language classes and bug types and illustrate an overall relationship between language class and the bug categories.

To understand the interaction between bug categories with individual languages, we use a separate NBR model, as discussed in RQ1, for each bug category. For each of these models we use the same control factors as RQ1 as well as languages encoded with weighted effects as predictors, with the number of defect fixing commits belong to that category as a response.

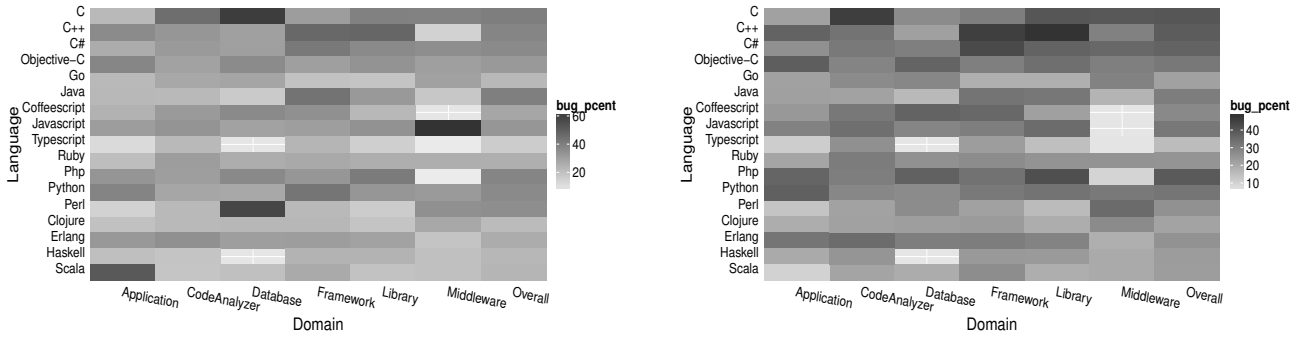
The results are shown in Table 8. For these models we present only a subset of the anova coefficients as the distribution of deviance explained is very similar to the language model presented in Table 6 and its associated anova table. The overall deviance for each model is substantially smaller and the 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 has a greater impact on specific categories of bugs, than it does on bugs overall. In the next section we expand on these results for the bug categories with significant bug counts as reported in Table 5. However, our conclusion generalizes for all categories.

Programming Errors. Generic programming errors account for around 88.53% of all bug fix commits and occur in all the language 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.*. Some errors are still more language-specific. For example, we find 122 runtime errors in `JavaScript` that do not appear in `TypeScript`, and similarly, `TypeScript` has more type related errors.

Memory Errors. Memory errors account 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 expected 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 a statistically greater number of memory errors. Among the managed languages, `Java` has a greater than average number of memory errors, although fewer than the unmanaged languages. Although `Java` has its own garbage collector, memory leaks are not surprising since unused object references often prevent the garbage collector from reclaiming memory [11]. In fact, we notice 28.89% of all the memory errors in `Java` are the result of 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 the total bug fix commits are related to Concurrency Errors. The heat map shows that `Proc-Static-Weak-Unmanaged` dominates this error type. `C` and `C++` introduce 19.15% and 7.89% of the errors, and they are distributed across the projects. Table 8 shows that this relationship is significant.

Language classes `Proc-Static-Strong-Managed` and `Functional-Static-Strong-Managed` are also in the darker zone in the



(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**

Figure 1: **Interaction between language's defect proneness and domain**

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.

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 $p < 0.0003076$.**

	Memory		Concurrency		Security		Failure	
(Intercept)	−7.49	$(0.46)^{***}$	−8.13	$(0.74)^{***}$	−7.29	$(0.58)^{***}$	−6.21	$(0.41)^{***}$
log commits	0.99	$(0.05)^{***}$	1.09	$(0.09)^{***}$	0.89	$(0.07)^{***}$	0.88	$(0.05)^{***}$
log age	0.15	$(0.06)^{*}$	0.19	(0.10)	0.30	$(0.08)^{***}$	0.07	(0.06)
log size	0.01	(0.04)	−0.08	(0.07)	−0.01	(0.05)	0.14	$(0.04)^{***}$
log devs	0.07	(0.04)	0.09	(0.07)	0.07	(0.06)	−0.11	$(0.04)^{*}$
C	1.71	$(0.12)^{***}$	0.39	(0.22)	0.28	(0.18)	0.43	$(0.13)^{**}$
C#	−0.12	(0.17)	0.81	$(0.24)^{***}$	−0.42	(0.23)	−0.07	(0.16)
C++	1.08	$(0.10)^{***}$	1.07	$(0.18)^{***}$	0.40	$(0.16)^{*}$	1.05	$(0.11)^{***}$
Objective-C	1.40	$(0.15)^{***}$	0.41	(0.28)	−0.14	(0.24)	1.10	$(0.15)^{***}$
Go	−0.05	(0.25)	1.62	$(0.30)^{***}$	0.35	(0.28)	−0.49	$(0.24)^{*}$
Java	0.53	$(0.14)^{***}$	0.80	$(0.22)^{***}$	−0.07	(0.19)	0.15	(0.14)
CoffeeScript	−0.41	(0.23)	−1.73	$(0.54)^{**}$	−0.36	(0.27)	−0.05	(0.19)
JavaScript	−0.16	(0.10)	−0.21	(0.16)	0.02	(0.12)	−0.15	(0.09)
TypeScript	−0.58	(0.62)	−0.63	(1.02)	0.37	(0.51)	−0.42	(0.41)
Ruby	−1.16	$(0.19)^{***}$	−0.89	$(0.29)^{**}$	−0.18	(0.21)	−0.32	$(0.16)^{*}$
Php	−0.69	$(0.17)^{***}$	−1.70	$(0.34)^{***}$	0.11	(0.21)	−0.62	$(0.17)^{***}$
Python	−0.48	$(0.14)^{***}$	−0.25	(0.22)	0.36	$(0.16)^{*}$	0.04	(0.12)
Perl	0.15	(0.35)	−1.23	(0.83)	−0.62	(0.45)	−0.64	(0.38)
Scala	−0.47	$(0.18)^{**}$	0.63	$(0.24)^{**}$	−0.22	(0.22)	−0.93	$(0.18)^{***}$
Clojure	−1.21	$(0.27)^{***}$	−0.01	(0.30)	−0.82	$(0.27)^{**}$	−0.62	$(0.19)^{**}$
Erlang	−0.60	$(0.23)^{**}$	0.63	$(0.28)^{*}$	0.62	$(0.22)^{**}$	0.59	$(0.17)^{***}$
Haskell	−0.28	(0.20)	−0.27	(0.32)	−0.45	(0.26)	−0.49	$(0.20)^{*}$
AIC		2991.47		2210.01		3328.39		4086.42
BIC		3101.15		2319.70		3437.91		4196.03
Log Likelihood		-1473.73		-1083.01		-1642.19		-2021.21
Deviance		895.02		665.17		896.58		1043.02
Num. obs.		1081		1081		1073		1077
Residual Deviance (NULL)		5065.3		2124.93		2170.23		3769.7
Language Type Deviance		522.86		139.67		42.72		240.51

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

heat map. The major languages that contribute concurrency errors from these classes are Go, C++, Java, C#, and Scala. These results confirm, in general static languages statistically 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, Ruby, and Php have fewer concurrency errors (note statistically significant negative coefficients, in Table 8).

	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

A textual analysis based on word-frequency of the bug fix messages suggests that most of the concurrency errors occur due to a race condition, deadlock, or incorrect synchronization, as shown in the table above. In all the languages, race condition is most frequent cause, ranging from 41% in C++ to 92% in Go. The enrichment of race condition errors in Go is likely because the Go is distributed with a race-detection tool that 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 (which does not require locking of shared resources) for inter-thread communication, which explains why these two languages do not have any SHM related errors like locking, mutex etc. In contrast, projects in the other languages use SHM primitives for communication and can thus may have locking-related errors.

Security and Other Impact Errors. Around 7.33% of all the bug fix commits are related to Impact errors. Among them Erlang, C++, and Python produce more security errors than av-

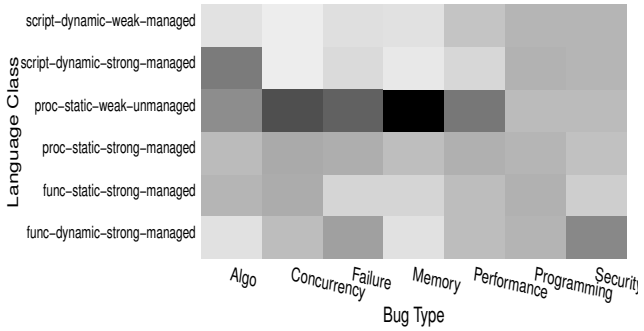


Figure 2: **Relation between bug categories and language class**
Each cell represents percentage of bug fix commit out of all bug fix commits per language class (row header) per bug category (column header). The values are normalized column wise.

erage (Table 8). The regression also suggests that projects written in `Clojure` are less likely to introduce a security error (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. 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 three 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.* [7] 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. However, 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.* [8] compared `C++`, a procedural language, with `SML`, a functional language, finding no significant difference in total number of errors, although `SML` has higher defect density than `C++`. `SML` was not represented in our data, which however, suggest that functional languages are generally less defect prone than procedural languages. Another line of work primarily focuses on comparing development effort across different languages [12, 20]. However, they do not analyze language defect proneness.

(2) *Surveys*: Meyerovich *et al.* survey developers' views of programming languages, to study why some languages are more popular than others[16]. 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.* [1] study four projects developed in both `C` and `C++` and find that the software components 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 (0.18) than `C` (0.11) (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. Also, the descriptiveness of commit logs vary across the projects. To mitigate the threat, we evaluate our classification against manual annotation as discussed in Section 2.4.

We determine the language of a file based on its extension, using GitHub Linguist.This can be error prone if a file written in a different language takes a common language extension that we have studied. To reduce such error, we manually verified language categorization to a randomly sampled file set.

To interpret the language classes in Section 2.2, we make certain assumptions based on how a language property is most commonly used, as reflected in our data set. For instance, we classify `Objective-C` as unmanaged memory type, although it may follow a hybrid memory model. Similarly, we annotate `Scala` as functional and `C#` as procedural language, although they support both procedural and functional design [19, 21].We do not distinguish object-oriented languages (OOP) in this work as there is no clear distinction between pure OOP languages and procedural languages.The difference largely depends on programming style. We categorize `C++` as weakly typed because a memory region of a certain type can be treated differently using pointer manipulation [22]. However, depending on the compiler some `C++` type errors can be detected in compile time. We further exclude `TypeScript` from our language classification model (see Table 3 and Table 7); `TypeScript` is intended to be used as a static, strongly typed language. However, in practice, we notice that developers often (for 50% of the variables, and in all the `TypeScript`-using projects in our dataset) use *any* type, a catch-all union type, and thus makes `TypeScript` dynamic and weak.

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 large scale study of language type and use, as it relates to software quality. The Github data we used is characterized by its complexity and 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.

Our sample-size allows 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 that managed memory usage is better than unmanaged. Further, that the defect proneness of languages in general is not associated with software domains. Also, languages are more related to individual bug categories than bugs overall.

On the other hand, even large datasets become small and insufficient when they are 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, we are unable 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.

7. ACKNOWLEDGEMENTS

We thank Sameer Khatri for cross checking domain categorization. We acknowledge support from the National Science Foundation under Grants No. CCF-1247280 and CCF-1446683 and from AFOSR award FA955-11-1-0246.

8. REFERENCES

[1] 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.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, pages 29–60, 1969.

[11] 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.

[12] 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.

[13] 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.

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

[15] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *ACM SIGPLAN Notices*, volume 47, pages 683–702. ACM, 2012.

[16] 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.

[17] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[18] 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.

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

[20] 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.

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

[22] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[23] 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.

[24] 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.

[25] 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.

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