A Large-Scale Study of Programming Languages and Code Quality in GitHub

By Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov

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 versus dynamic typing and allowing versus disallowing type confusion 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 disallowing type confusion is modestly better than allowing it, 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 caution the reader that even these modest effects might quite possibly be due to other, intangible process factors, for example, the preference of certain personality types for functional, static languages that disallow type confusion.

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 the static approach catches defects early; for them, an ounce of prevention is worth a pound of cure. 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, supported only by anecdotal evidence.

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, for example, software quality, there are a number of well-known influential factors, such as code size,[6] team size,[2] and age/maturity.[9]

Controlled experiments are 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, that is, limited scope. The tasks completed in such studies are necessarily limited and do not emulate *real world* development. There have been several such studies recently that use students, or compare languages with static or dynamic typing through an experimental factor.[7, 12, 15]

Fortunately, we can now study these questions over a large body of real-world software projects. GitHub contains many projects in multiple languages that substantially vary across size, age, and number of developers. Each project repository provides a detailed record, including 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. This empirical approach helps us to understand the practical impact of programming languages, as they are used colloquially by developers, on software quality.

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. 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 18 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 to provide an interface for interactive data analysis.

Identifying top languages. We aggregate projects based on their primary language. Then we select the languages with the most projects for further analysis, as shown in Table 1. A given project can use many languages; assigning a single language to it is difficult. Github Archive stores information gathered from GitHub Linguist which measures the language distribution of a project repository using the source file extensions. 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 associated number of *stars*. This number indicates how many people have actively expressed interest in the project, and is a reasonable proxy for its popularity. Thus, the top 3 projects in C are *linux, git,* and *php-src*; and for C++ they are *node-webkit, phantomjs,* and *mongo*; and for Java they are *storm, elasticsearch,* and *ActionBarSherlock*. In total, we select the top 50 projects in each language.

To ensure that these projects have a sufficient development history, we drop the projects with fewer than 28 commits (28 is the first quartile commit count of considered projects). This leaves us with 728 projects. Table 1 shows the top 3 projects in each language.

Retrieving project evolution history. For each of 728 projects, we downloaded the non-merged commits, commit logs, author date, and author name using *git*. We compute code churn and the number of files modified per commit from the number of added and deleted lines per file. We retrieve the languages associated with each commit from the extensions of the modified files (a 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 other project-related statistics, including maximum commit age of a project and the total number of developers, used as control variables in our regression model, and discussed in Section 3. We identify 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, similar to a prior study.[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 sufficient activity for each language–project pair.

In summary, we study 728 projects developed in 17 languages with 18 years of history. This includes 29,000 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

Table 1. Top 3 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

Table 2. Study subjects.

| Language | Project details | | Commits | | |
	#Projects	#Devs (K)	#Commits (K)	#Insertion (MLOC)	#BugFixes (K)
C	220	13.8	447.8	75.3	182.6
C++	149	3.8	196.5	46.0	79.3
C#	77	2.3	135.8	27.7	50.7
Objective-C	93	1.6	21.6	2.4	7.1
Go	54	6.6	19.7	1.6	4.4
Java	141	3.3	87.1	19.1	35.1
CoffeeScript	92	1.7	22.5	1.1	6.3
JavaScript	432	6.8	118.3	33.1	39.3
TypeScript	14	2.4	3.3	2.0	0.9
Ruby	188	9.6	122.1	5.8	30.5
Php	109	4.9	118.7	16.2	47.2
Python	286	5.0	114.2	9.0	41.9
Perl	106	0.8	5.5	0.5	1.9
Clojure	60	0.8	28.4	1.5	6.0
Erlang	51	0.8	31.4	5.0	8.1
Haskell	55	0.9	46.1	2.9	10.4
Scala	55	1.3	55.7	5.3	12.9
Summary	728	28	1574	254	564

Table 3. Different types of language classes.

Language classes	Categories	Languages
Programming paradigm	Imperative procedural	`C, C++, C#, Objective-C, Java, Go`
	Imperative scripting	`CoffeeScript, JavaScript, Python, Perl, Php, Ruby`
	Functional	`Clojure, Erlang, Haskell, Scala`
Type checking	Static	`C, C++, C#, Objective-C, Java, Go, Haskell, Scala`
	Dynamic	`CoffeeScript, JavaScript, Python, Perl, Php, Ruby, Clojure, Erlang`
Implicit type conversion	Disallow	`C#, Java, Go, Python, Ruby, Clojure, Erlang, Haskell, Scala`
	Allow	`C, C++, Objective-C, CoffeeScript, JavaScript, Perl, Php`
Memory class	Managed	`Others`
	Unmanaged	`C, C++, Objective-C`

We omit `TypeScript` from language classification as it allows both explicit and implicit type conversion.

project is written in an imperative procedural, imperative scripting, or functional language. In the rest of the paper, we use the terms procedural and scripting to indicate imperative procedural and imperative scripting respectively.

Type Checking indicates static or dynamic typing. In statically typed languages, type checking occurs at compile time, and variable names are bound to a value and to a type. In addition, expressions (including variables) are classified by types that correspond to the values they might take on at runtime. In dynamically typed languages, type checking occurs at run-time. Hence, in the latter, it is possible to bind a variable name to objects of different types in the same program.

Implicit Type Conversion allows access of an operand of type T1 as a different type T2, without an explicit conversion. Such implicit conversion may introduce type-confusion in some cases, especially when it presents an operand of specific type T1, as an instance of a different type T2. Since not all implicit type conversions are immediately a problem, we operationalize our definition by showing examples of the implicit type confusion that can happen in all the languages we identified as allowing it. For example, in languages like `Perl`, `JavaScript`, and `CoffeeScript` adding a string to a number is permissible (e.g., "5" + 2 yields "52"). The same operation yields 7 in `Php`. Such an operation is not permitted in languages such as `Java` and `Python` as they do not allow implicit conversion. In `C` and `C++` coercion of data types can result in unintended results, for example, `int x; float y; y=3.5; x=y;` is legal C code, and results in different values for x and y, which, depending on intent, may be a problem downstream.[a] In `Objective-C` the data type *id* is a generic object pointer, which can be used with an object of any data type, regardless of the class.[b] The flexibility that such a generic

data type provides can lead to implicit type conversion and also have unintended consequences.[c] Hence, we classify a language based on whether its compiler *allows* or *disallows* the implicit type conversion as above; the latter explicitly detects type confusion and reports it.

Disallowing implicit type conversion could result from static type inference within a compiler (e.g., with `Java`), using a type-inference algorithm such as Hindley[10] and Milner,[17] or at run-time using a dynamic type checker. In contrast, a type-confusion can occur silently because it is either undetected or is unreported. Either way, implicitly allowing type conversion provides flexibility but may eventually cause errors that are difficult to localize. To abbreviate, we refer to languages allowing implicit type conversion as *implicit* and those that disallow it as *explicit*.

Memory Class indicates whether the language requires developers to manage memory. We treat `Objective-C` as unmanaged, in spite of it following a hybrid model, because we observe many memory errors in its codebase, as discussed in RQ4 in Section 3.

Note that we classify and study the languages as they are colloquially used by developers in real-world software. For example, `TypeScript` is intended to be used as a static language, which disallows implicit type conversion. However, in practice, we notice that developers often (for 50% of the variables, and across `TypeScript`-using projects in our dataset) use the `any` type, a catch-all union type, and thus, in practice, `TypeScript` allows dynamic, implicit type conversion. To minimize the confusion, we exclude `TypeScript` from our language classifications and the corresponding model (see Table 3 and 7).

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] 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, that is, topics, and estimate the probability that each project belonging to each domain. Since these auto-detected domains include several project-specific keywords, for example, facebook, it is difficult to identify the underlying common functions. In order to assign a meaningful name to each domain, we manually inspect each of the 30 domains to identify projectname-independent, domain-identifying keywords. We manually rename all of the 30 auto-detected domains 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 do not fall under any of the above

---

[a] Wikipedia's article on type conversion, https://en.wikipedia.org/wiki/Type_conversion, has more examples of unintended behavior in C.

[b] This Apple developer article describes the usage of "id" http://tinyurl.com/jkl7cby.

[c] Some examples can be found here http://dobegin.com/objc-id-type/ and here http://tinyurl.com/hxv8kvg.

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.

2.4. Categorizing bugs

While fixing software bugs, developers often leave important information in the commit logs about the nature of the bugs; for example, why the bugs arise and how to fix the bugs. We exploit such information to categorize the bugs, similar to Tan *et al.*[13, 24]

First, we categorize the bugs based on their *Cause* and *Impact. Causes* are further classified into disjoint subcategories of errors: Algorithmic, Concurrency, Memory, generic Programming, and Unknown. The bug *Impact* is also classified into four disjoint subcategories: Security, Performance, Failure, and Other unknown categories. Thus, each bug-fix commit also has an induced Cause and an Impact type. 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 them as 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. Such a restrictive set of keywords and phrases helps 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 techniques. Finally, we use Support Vector Machine 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 performance of this process was acceptable with precision ranging from a low of 70% for performance bugs to a high of 100% for concurrency bugs with an average of 84%. Recall ranged from 69% to 91% with an average of 84%.

The result of our bug classification is shown in Table 5. Most of the defect causes are related to generic programming errors. This is not surprising as this category involves a wide variety of programming errors such as type errors, typos, compilation error, etc. Our technique could not classify 1.04% of the bug fix messages in any Cause or Impact category; we classify these as Unknown.

2.5. Statistical methods

We model the number of defective commits against other factors related to software projects using regression. All models use *negative binomial regression* (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.[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.

Table 4. Characteristics of domains.

Domain name	Domain characteristics	Example projects	Total projects
(APP) Application	End user programs	bitcoin, macvim	120
(DB) Database	SQL and NoSQL	mysql, mongodb	43
(CA) CodeAnalyzer	Compiler, parser, etc.	ruby, php-src	88
(MW) Middleware	OS, VMs, etc.	linux, memcached	48
(LIB) Library	APIs, libraries, etc.	androidApis, opencv	175
(FW) Framework	SDKs, plugins	ios sdk, coffeekup	206
(OTH) Other	–	Arduino, autoenv	49

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)	Multithreading/processing issues	Deadlock, race condition, synchronization error	11,111	1.99
	Memory (Mem)	Incorrect memory handling	Memory leak, null pointer, buffer overflow, heap overflow, null pointer, dangling pointer, double free, segmentation fault	30,437	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	495,013	88.53
Impact	**Security** (Sec)	Runs, but can be exploited	Buffer overflow, security, password, oauth, ssl	11,235	2.01
	Performance (Perf)	Runs, but with delayed response	Optimization problem, performance	8651	1.55
	Failure (Fail)	Crash or hang	Reboot, crash, hang, restart	21,079	3.77
	Unknown (Unkn)	Not part of the above categories		5792	1.04

To check that excessive multicollinearity is not an issue, we compute the variance inflation factor of each dependent variable in all of the models with a conservative maximum value of 5.[4] We check for and remove high leverage points through visual examination of the residuals versus 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 projects.[23] To test for the relationship between two factor variables we use a Chi-square test of independence.[14] 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.

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 use a regression model to compare the impact of each language on the number of defects with the average impact of all languages, against defect fixing commits (see 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.

Table 6. Some languages induce fewer defects than other languages.

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)***

Response is the number of defective commits. Languages are coded with weighted effects coding. *AIC=10432, Deviance=1156, Num. obs.=1076.*
***p < 0.001, **p < 0.01, *p < 0.05

The sign and magnitude of the estimated coefficients 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 that they are all positive and significant. The language variables are indicator variables, viz. 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 is those for which the coefficient is statistically insignificant and the modeling procedure could not distinguish the coefficient from zero. These languages may behave similar 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 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.

One should take care not to overestimate the impact of language on defects. While the observed relationships are statistically significant, the effects are quite small. Analysis of deviance reveals that language accounts for less than 1% of the total explained deviance.

	Df	**Deviance**	**Resid. *Df***	**Resid. dev**	**Pr (>Chi)**
NULL			1075	25,176.25	
Log commits	1	4256.89	1071	1286.74	0
Log age	1	8011.52	1074	17,164.73	0
Log size	1	10,082.78	1073	7081.95	0
Log devs	1	1538.32	1072	5543.63	0
Language	16	130.78	1055	1155.96	0

We can read the model 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; that is, for a coefficient β_i, a one unit change in β_i yields an expected change in the response of $e^{\beta i}$. For the factor variables, this expected change is compared to 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 defective 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 depends on all other factors remaining the same, a challenging proposition for all but the most trivial of projects. All observational studies face similar limitations; 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. (2) *Generic bug*: more generic in nature and has less to do with project function, for example, typeerrors, 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, for example, `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 on the programming language featured. However, since a domain-specific bugs 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, that is, 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[d]; 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 results.

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 to defects. Broadly, each of these properties divides languages along lines that are often discussed in the context of errors, drives user debate, or has been the subject of prior work. Since the individual properties are highly correlated, we create six model factors that combine all of the individual factors across all of the languages in our study. We then 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*.

As with language (earlier in Table 6), we are comparing language *classes* with the average behavior across all language classes. The model is presented in Table 7. It is clear that `Script-Dynamic-Explicit-Managed` class has the smallest magnitude coefficient. The coefficient is insignificant, that is, the z-test for the coefficient cannot distinguish the coefficient from zero. Given the magnitude of the standard error, however, we can assume that the behavior of languages in this class is very close to the average across all languages. We confirm this by recoding the coefficient using `Proc-Static-Implicit-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-Explicit-Managed` is significantly

[d] Chi-squared value of 243.6 with 96 *df*. and $p = 8.394\mathrm{e}{-15}$

Table 7. Functional languages have a smaller relationship to defects than other language classes whereas procedural languages are greater than or similar to the average.

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-Explicit-Managed	−0.25 (0.04)***
Functional-Dynamic-Explicit-Managed	−0.17 (0.04)***
Proc-Static-Explicit-Managed	−0.06 (0.03)*
Script-Dynamic-Explicit-Managed	0.001 (0.03)
Script-Dynamic-Implicit-Managed	0.04 (0.02)*
Proc-Static-Implicit-Unmanaged	0.14 (0.02)***

Language classes coded with weighted effects codes (AIC = 10,419, Deviance = 1132, Num. obs. = 1067).
***$p < 0.001$, **$p < 0.01$, *$p < 0.05$.

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-Implicit-Unmanaged` languages are more likely to induce defects. This implies that either implicit type conversion or memory management issues contribute to greater defect proneness as compared with other procedural languages.

Among scripting languages we observe a similar relationship between languages that allow versus those that do not allow implicit type conversion, providing some evidence that implicit type conversion (vs. explicit) is responsible for this difference as opposed to memory management. We cannot state this conclusively given the correlation between factors. However when compared to the average, as a group, languages that do not allow implicit type conversion are less error-prone while those that do are more error-prone. 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. Statically typed languages have a 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-Explicit-Managed` language class is significantly less defect-prone than the `Functional-Dynamic-Explicit-Managed` language class with $p = 0.034$.

	Df	**Deviance**	**Resid. *Df***	**Resid. Dev**	**Pr (>Chi)**
NULL			1066	32,995.23	
Log commits	1	31,634.32	1065	1360.91	0
Log age	1	51.04	1064	1309.87	0
Log size	1	50.82	1063	1259.05	0
Log devs	1	31.11	1062	1227.94	0
Lang. class	5	95.54	1057	1132.40	0

As with language and defects, the relationship between language class and defects is based on a small effect. The deviance explained is similar, albeit smaller, with language class explaining much less than 1% of the deviance.

We now revisit the question of application domain. Does domain have an interaction with language class? ==Does the choice of, for example, a functional language, have an advantage for a particular domain?== As above, a Chi-square test for the relationship between these factors and the project domain yields a value of 99.05 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.

Result 2: *There is a small but significant relationship between language class and defects. Functional languages are associated with fewer defects than either procedural or scripting languages.*
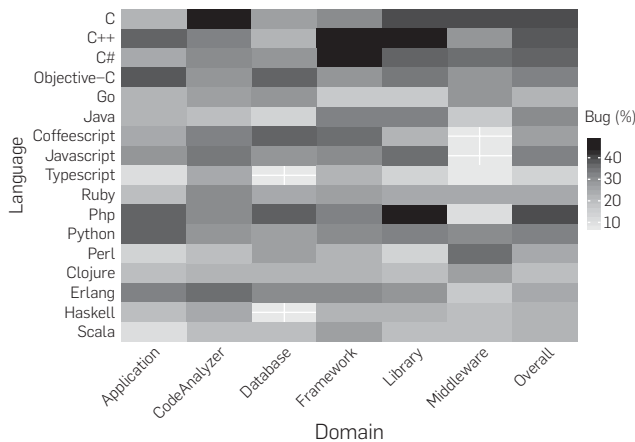
It is somewhat unsatisfying that we do not observe a strong association between language, or language class, and domain within a project. An alternative way to view this same data is to disregard projects and aggregate defects over all languages and domains. Since this does not yield independent samples, 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?

In order to answer this question we first filtered out projects that would have been viewed as outliers, filtered as high leverage points, in our regression models. This was necessary

here as, even though this is a nonstatistical method, some relationships could impact visualization. For example, we found that a single project, Google's v8, a `JavaScript` project, was responsible for all of the errors in Middleware. This was surprising to us since `JavaScript` is typically not used for Middleware. This pattern repeats in other domains, consequently, we filter out the projects that have defect density below 10 and above 90 percentile. The result is in Figure 1.

We see only a subdued variation in this heat map which is a result of the inherent defect proneness of the languages as 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 except Database, the correlation is positive, and p-values are significant (<0.01). Thus, w.r.t. defect proneness, the 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 application 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 does not 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 an approach similar to RQ3 to understand the relation between languages and bug categories. First, we study the relation between bug categories and language class. A heat map (Figure 2) shows aggregated defects over language classes and bug types. To understand the interaction between

Figure 1. Interaction of language's defect proneness with 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, that is, no commits were made corresponding to that cell.

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.
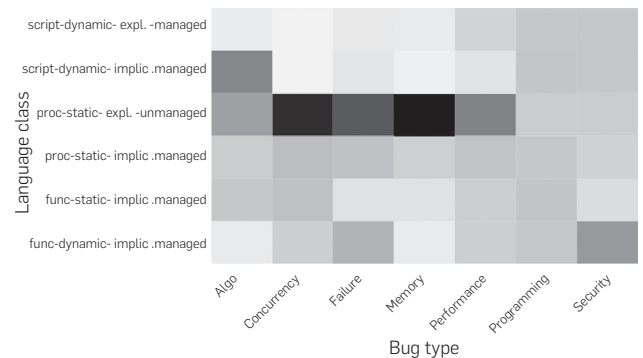
bug categories and languages, we use an NBR regression model for each category. For each model we use the same control factors as RQ1 as well as languages encoded with weighted effects to predict defect fixing commits.

The results along with the anova value for language are shown in Table 8. 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. Consequently, the regression analysis draws a similar conclusion as of RQ1 (see Table 6). All languages incur programming errors such as faulty error-handling, faulty definitions, typos, etc.

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-Implicit-Unmanaged` class and memory errors. This is expected as languages with unmanaged memory are known for memory bugs. Table 8 confirms that such languages, for example, `C`, `C++`, and `Objective-C` introduce more memory errors. Among the

managed languages, `Java` induces more 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 our data, 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-Implicit-Unmanaged` dominates this error type. `C` and `C++` introduce 19.15% and 7.89% of the errors, and they are distributed across the projects.

	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

Both of the `Static-Strong-Managed` language classes are in the darker zone in the heat map confirming, in general static languages produce more concurrency errors than others. Among the dynamic languages, only `Erlang` is more prone to concurrency errors, perhaps relating to the greater use of this language for concurrent applications. Likewise, the negative coefficients in Table 8 shows that projects written

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.

	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
Deviance	895.02	665.17	896.58	1043.02
Num. obs.	1081	1081	1073	1077
Residual deviance (NULL)	5065.30	2124.93	2170.23	3769.70
Language deviance	522.86	139.67	42.72	240.51

For all models the deviance explained by language type has $p < 0.0003076$.
***$p < 0.001$, **$p < 0.01$, *$p < 0.05$.

in dynamic languages like `Ruby` and `Php` have fewer concurrency errors. Note that, certain languages like `JavaScript`, `CoffeeScript`, and `TypeScript` do not support concurrency, in its traditional form, while `Php` has a limited support depending on its implementations. These languages introduce artificial zeros in the data, and thus the concurrency model coefficients in Table 8 for those languages cannot be interpreted like the other coefficients. Due to these artificial zeros, the average over all languages in this model is smaller, which may affect the sizes of the coefficients, since they are given w.r.t. the average, but it will not affect their relative relationships, which is what we are after.

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. Across all language, race conditions are the most frequent cause of such errors, for example, 92% in `Go`. The enrichment of race condition errors in `Go` is probably due to an accompanying race-detection tool that may help developers locate races. The synchronization errors are primarily related to message passing interface (MPI) or shared memory operation (SHM). `Erlang` and `Go` use MPI[e] for inter-thread communication, which explains why these two languages do not have any SHM related errors such as 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` associate with more security errors than average (Table 8). `Clojure` projects associate with fewer security errors (Figure 2). From the heat map we also see that `Static` languages are in general more prone to failure and performance errors, these are followed by `Functional-Dynamic-Explicit-Managed` languages such as `Erlang`. The analysis of deviance results confirm that language is strongly associated with failure impacts. While security errors are the weakest among the categories, the deviance explained by language is still quite strong when compared with the residual deviance.

Result 4: *Defect types are strongly associated with languages; some defect type like memory errors and 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[7] compared static versus dynamic typing by monitoring 48 programmers for 27 h while developing a parser program. He found no significant difference in code quality between the two; however, dynamic type-based languages were found to 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 disallowing implicit type conversion is better than allowing it, 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 and Rabkin[16] survey developers' views of programming languages, to study why some languages are more popular than others. They report strong influence from non-linguistic factors: prior language skills, availability of open source tools, and existing legacy systems. Our study also confirms that the availability of external tools also impacts software quality; for example, concurrency bugs in `Go` (see RQ4 in Section 3).

(3) ***Repository mining***. Bhattacharya and Neamtiu[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. 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 rely on the keywords that developers often use to indicate a bug fix. Our choice was deliberate. We wanted to capture the issues that developers continuously face in an ongoing development process, rather than reported bugs. However, this choice possesses threats of over estimation. Our categorization of domains is subject to interpreter bias, although another member of our group verified the categories. Also, our effort to categorize bug fix commits could potentially be tainted by the initial choice of keywords. The descriptiveness of commit logs vary across projects. To mitigate these threats, we evaluate our classification against manual annotation as discussed in Section 2.4.

We determine the language of a file based on its extension. 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 against a randomly sampled file set.

To interpret language class 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 example, we classify `Objective-C` as unmanaged memory type rather

---

[e] MPI does not require locking of shared resources.

than hybrid. Similarly, we annotate `Scala` as functional and `C#` as procedural, although both support either design choice.[19, 21] We do not distinguish object-oriented languages (OOP) from procedural languages in this work as there is no clear distinction, the difference largely depends on programming style. We categorize `C++` as allowing implicit type conversion because a memory region of a certain type can be treated differently using pointer manipulation.[22] We note that most `C++` compilers can detect type errors at compile time.

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 variance along multiple dimensions. Our sample size allows a mixed-methods study of the effects of language, and of the interactions of language, domain, and defect type while controlling for a number of confounds. The data indicates that functional languages are better than procedural languages; it suggests that disallowing implicit type conversion is better than allowing it; 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. Additionally, 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. Consequently, with an increasing number of dependent variables it is difficult to answer questions about a specific variable's effect, especially where variable interactions 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.

References

1. Bhattacharya, P., Neamtiu, I. 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* (New York, NY, USA, 2011). ACM, 171–180.
2. Bird, C., Nagappan, N., Murphy, B., Gall, H., Devanbu, P. 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* (2011). ACM, 4–14.
3. Blei, D.M. Probabilistic topic models. *Commun. ACM 55,* 4 (2012), 77–84.
4. Cohen, J. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences.* Lawrence Erlbaum, 2003.
5. Easterbrook, S., Singer, J., Storey, M.-A., Damian, D. Selecting empirical methods for software engineering research. In *Guide to Advanced Empirical Software Engineering* (2008). Springer, 285–311.
6. El Emam, K., Benlarbi, S., Goel, N., Rai, S.N. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Softw. Eng. 27,* 7 (2001), 630–650.
7. Hanenberg, S. 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* (New York, NY, USA, 2010). ACM, 22–35.
8. Harrison, R., Smaraweera, L., Dobie, M., Lewis, P. Comparing programming paradigms: An evaluation of functional and object-oriented programs. *Softw. Eng. J. 11,* 4 (1996), 247–254.
9. Harter, D.E., Krishnan, M.S., Slaughter, S.A. Effects of process maturity on quality, cycle time, and effort in software product development. *Manage. Sci. 46* 4 (2000), 451–466.
10. Hindley, R. The principal type-scheme of an object in combinatory logic. *Trans. Am. Math. Soc.* (1969), 29–60.
11. Jump, M., McKinley, K.S. Cork: Dynamic memory leak detection for garbage-collected languages. In *ACM SIGPLAN Notices,* Volume 42 (2007). ACM, 31–38.
12. Kleinschmager, S., Hanenberg, S., Robbes, R., Tanter, É., Stefik, A. Do static type systems improve the maintainability of software systems? An empirical study. In *2012 IEEE 20th International Conference on Program Comprehension (ICPC)* (2012). IEEE, 153–162.
13. Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C. 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. Marques De Sá, J.P. *Applied Statistics Using SPSS, Statistica and Matlab,* 2003.
15. Mayer, C., Hanenberg, S., Robbes, R., Tanter, É., Stefik, A. An empirical study of the influence of static type systems on the usability of undocumented software. In *ACM SIGPLAN Notices,* Volume 47 (2012). ACM, 683–702.
16. Meyerovich, L.A., Rabkin, A.S. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (2013). ACM, 1–18.
17. Milner, R. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17,* 3 (1978), 348–375.
18. Mockus, A., Votta, L.G. Identifying reasons for software changes using historic databases. In *ICSM'00. Proceedings of the International Conference on Software Maintenance* (2000). IEEE Computer Society, 120.
19. Odersky, M., Spoon, L., Venners, B. *Programming in Scala.* Artima Inc, 2008.
20. Pankratius, V., Schmidt, F., Garretón, G. 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* (2012). IEEE Press, 123–133.
21. Petricek, T., Skeet, J. *Real World Functional Programming: With Examples in F# and C#.* Manning Publications Co., 2009.
22. Pierce, B.C. *Types and Programming Languages.* MIT Press, 2002.
23. Posnett, D., Bird, C., Dévanbu, P. An empirical study on the influence of pattern roles on change-proneness. *Emp. Softw. Eng. 16,* 3 (2011), 396–423.
24. Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C. Bug characteristics in open source software. *Emp. Softw. Eng.* (2013).

Baishakhi Ray (rayb@virginia.edu), Department of Computer Science, University of Virginia, Charlottesville, VA.

Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov ([dpposnett@, devanbu@cs., filkov@cs.]ucdavis.edu), Department of Computer Science, University of California, Davis, CA.