FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Analyzing Large Code Repositories

by

Petr Máj

A dissertation thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Dissertation degree study programme: Informatics
Department of Theoretical Computer Science

Prague, January 2023

Supervisor:
doc. Ing. Jan Janoušek Ph.D.
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Co-Supervisor:
prof. Jan Vitek Ph.D.
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Abstract and contributions

Software engineering benefits from insights gleaned from large-scale software repositories as they offer an unmatched window into the development process. Their sheer size holds the promise of broadly applicable results. Yet, that very size presents scalability challenges. One answer to such challenges is to limit studies to representative samples and generalize observations to the entire population. But finding such a representative sample is often impossible and researchers must compromise by using smaller datasets with imprecise sampling, or sacrifice reproducibility of their results. This thesis analyzes the challenges in project selection for mining large software repositories and provides a tool that supports precise, scalable and reproducible sampling of software projects based on their attributes. Its contributions are detailed in four papers:

1. *A Map of Code Duplicates on GitHub* [A.3] analyses source code clones present in GitHub projects. It verifies the existence of one of the most common biases and shows its scale. Our findings signify the necessity for dedicated project selection and filtering steps in big code analyses.

2. *On the Impact of Programming Languages on Code Quality* [A.2] is a reproduction study focusing on the data filtering, reproducibility, and statistical interpretation of large corpora analyses. The paper shows the problems pointed out by this thesis are present in contemporary research and that they affect our results.

3. *Reproducible Queries over Large-Scale Software Repositories* [A.1] introduces the infrastructure that forms the statement of this thesis: a scalable, precise, deterministic, up-to-date and reproducible project selection pipeline.

4. *How to Design Reproducible Large-scale Code Analysis Experiments* [A.4] then devises and argues for an explicit and rigorous project filtering step and demonstrates how it can be done with the tool presented in the previous paper.

Keywords: Repository mining, big code, code duplication, selection bias, sampling.

Abstrakt

Softwarové inženýrství těží z poznatků získaných z velkých softwarových repozitářů, které nabízejí bezkonkurenční vhled do vývojového procesu. Jejich rozsah sám o sobě je příslibem široké použitelnosti výsledků. Tento rozsah však také představuje výzvy pro škálovatelnost. Jednou z odpovědí na tyto výzvy je omezení analýzy na reprezentativní vzorky a zobecnění pozorovaných jevů na celou populaci. Najít tento reprezentativní vzorek je však často nemožné. Výzkumníci musí dělat kompromisy jako je použití menších souborů dat s nepřesným vzorkováním nebo obětování reprodukovatelnosti jejich výsledků. Tato práce analyzuje výzvy pro výběr projektů v rámci vytěžování velkých softwarových úložišť a poskytuje nástroj, který podporuje přesné, škálovatelné a reprodukovatelné vzorkování softwarových projektů podle jejich atributů. Přínosy této práce jsou podrobně popsány v těchto čtyřech článcích:

1. *A Map of Code Duplicates on GitHub* [A.3] analyzuje klony ve zdrojových kódech GitHub projeků. Naše práce ověřila existenci jednoho z nejběžnějších zkreslení v rámci datových souborů a ukazuje jeho rozsah. Naše zjištění potvrzují nutnost specializovaného výběru projektů a jejich pečlivému filtrování v analýzých velkých repozitářů.

2. *On the Impact of Programming Languages on Code Quality* [A.2] je reprodukční studie se zaměřením na filtrování dat, reproducibilitu a statistickou interpretaci výsledků z velkých datasetů. Článek dokazuje, že problémy, na které tato práce upozorňuje existují v současném výzkumu a že ovlivňují jeho výsledky.

3. *Reproducible Queries over Large-Scale Software Repositories* [A.1] představuje infrastrukturu, která tvoří jádro této práce: škálovatelnou, přesný a reprodukovatelný výběr projektů.

4. *How to Design Reproducible Large-scale Code Analysis Experiments* [A.4] pak prezentuje design pečlivého a explicitního výběru projektů pro analýzy velkých repozitářů kódu a ukazuje jak je možné takový výběr realizovat za pomoci nástroje prezentovaného v předchozím článku.

Klíčová slova: vytěžování repozitářů, big code, duplikace kódu, vzorkování, selection bias.

Acknowledgements

First of all, I would like to express my gratitude to my dissertation thesis co-supervisor, Professor Jan Vitek. His vast knowledge, guidance and patience had taught me a lot more than just the skills necessary for finishing this thesis. I would also like to thank my supervisor, associate professor Jan Janousek who has welcomed me at FIT and guided me throughout my studies.

Special thanks go to the staff of the Department of Theoretical Computer Science, who maintained a pleasant and flexible environment for my research and my colleagues for the inspiring discussions throughout the years.

Finally, my greatest thanks go to my family. To Kristina, my wife, for her infinite support and care and to Ada and Frantisek, our kids, for being who they are and thus making sure that research never proved too difficult:)

My research has been supported by the Czech Ministry of Education, Youth and Sports from the Czech Operational Programme Research, Development, and Education, under grant agreement No. CZ.02.1.01/0.0/0.0/15_003/0000421.

Dedication

In memory of doc. Ing. Karel Müller, CSc.

for his knowledge, wisdom, inspiration, and kindness

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"The temptation to form theories upon insufficient data is the bane of our profession."
- Sherlock Holmes

The world as we know it relies upon billions and billions of computers. From the tiny ones embedded in their hundreds in our cars, television sets and washing machines to the larger ones in smartphones, to the laptops and desktop computers, all the way to the supercomputers and datacenters. Those computers are integral part of modern society. Over the years their size has decreased almost as fast as their power increased: the smart watches we wear on our wrists are as powerful as big box computers from the turn of the century, about 100,000 times faster than the hardware that landed Apollo on the Moon.

The true ingenuity of a computer comes from the fact that a single physical computer can do different tasks depending only on its code, instructions that break the complex tasks to series of very simple operations the computer knows how to perform. To make a machine perform a new task simply means to load a new software for it. But as the number of computers exploded, supplying them with new software quickly and reliably became a problem. As Edsger Dijkstra famously said in the Humble Programmer [3]:

> *The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*

As the software tasks shifted away from number crunching to more complex tasks, the values manipulated by computers became proxies of much more abstract objects with more complicated relations. But while the runtime complexity of programs increased tremendously, the low-level instructions that computers perform remained largely intact. Instead of expressing the complex interactions that describe the intended functionality of the software, programmers spent most of their time in routine decomposition. To remedy this, high level programming languages were designed: those languages abstract from the low-level capabilities of the machine and focus on expressivity at levels better suited to the relations and

algorithms of the more complex tasks. As an example, consider manipulating an array of integers in three languages: C, Java and Rust:

```
int * arr = malloc (sizeof(int) * n);
fill_array(arr);
int sum = 0;
for (size_t i = 0; i != n; ++i)
     sum += *(arr + i * sizeof(int));
printf("Sum: %zu", sum);
free(arr);
```

Figure 1.1: Manipulating integer array in C

The oldest language, C (figure 1.1), requires the programmer to pay attention to low-level details: the array must be manually allocated with correctly calculated size and deallocated when no-longer needed. Accessing array elements is done by offset calculation and iteration must be explicitly bounded. This level of detail allows programmers to squeeze out every bit of performance from the machine but has to be repeated for every use. Mistakes lead to costly errors.

```
ArrayList<int> arr = new ArrayList<int>();
fill_array(arr);
int sum = 0;
Iterator<int> = arr.iterator();
while (iter.hasNext()) {
    sum = sum + iter.next();
}
System.out.println(sum);
```

Figure 1.2: Manipulating integer array in Java

Unlike C, Java (figure 1.2) uses automatic memory management so memory does not have to be explicitly deallocated. Memory offsets and sizes do not have to be computed explicitly. Iteration is done using an iterator which ensures no location outside of the array can be accessed. The same mechanism can be used to iterate over various structures, such as lists and trees for better modularity. The language provides more correctness guarantees to the programmer, increasing the ease of writing code.

```
let mut array = [i32; n];
fill_array(& mut array);
let sum = array.iter().reduce(|a,b| a + b);
println!("Sum : {sum}");
```

Figure 1.3: Manipulating integer array in Rust

But automatic memory management is not the only productivity feature. Even manual memory management can be safe. Rust (figure 1.3) is a newer language with manual memory management and strong safety guarantees. In Rust, the array does not have to be freed manually, deallocation is inserted by the compiler when the variable goes out of scope. It-

eration looks different too: inspired by functional languages, one can write code with fewer opportunities for errors. One simple task, three very different approaches.

Despite the Herculean efforts, best intentions and bold claims of the language designers, the goal of improving programmers' productivity has suffered from a persistent lack of evaluation. There is very little we know about the effect of language design choices on the productivity of programmers and quality of the code they write. This invites a sea of opinions. While we all see the differences between the above solution, we will likely not agree that one language is, strictly speaking, the best. Fruitless arguments continuously erupt over all aspects of software development from benefits and scope of particular language features to superiority of programming styles, or even whether to use spaces or tabs in code.

1.1 On comparing languages

How can one determine which programming languages make programmers more effective? Such comparisons should be answered scientifically with controlled experiments: experiments are run with all variables controlled for, except one, in our case the choice of language. Any difference in outcome can then be attributed to that variable.

The first problem we encounter is how to determine what to measure. The naïve approach would be to simply measure the time it takes to complete a given task in a language. The faster the development, the better. However, if a solution contains more errors, the time required to find and fix them will likely offset the initial productivity gains. Instead of measuring the raw development time, we could focus on errors (or bugs). The fewer bugs in programs written in some language, the better the language.

Let us make this discussion specific. Imagine that we wanted to render judgement on two very distinct programming languages: C++ and Haskell. The former is a general-purpose language created as an object-oriented extension to C and designed to allow writing low-level code found in, for example, operating systems. It is statically typed and uses manual memory management. It first appeared in 1985 and has been continuously improved by the C++ Language Committee. The second language, Haskell first appeared in 1990 and, while it too is a general purpose statically typed language, it emphasizes fundamentally different programming model: it is a purely functional language where state cannot be mutated. It has a powerful type inference system so that variable types can be computed by the compiler and lazy evaluation of arguments leading to much denser code. The differences between the languages are profound and more subtle than we can outline. While the details are outside of our scope, a simple example such as finding prime numbers can illustrate the two approaches.

The C++ program (figure 1.4) is verbose. A growable array of integers is created, then the code tries successively each number from 2 to 100. For each, we assume it is prime and then check that no smaller number is a divisor. If a divisor is found, the prime flag is cleared and the inner loop terminated. After the inner loop, the number is added to the results if the flag is true.

The Haskell program (figure 1.5) filters numbers from 2 to 100 such that there is no i that divides n completely for i's between 2 and n-1. While the algorithm is the same, the two programs are quite different. Haskell is shorter, contains no type annotations and leverages high level operators. It is conceivable that the terse description with much more implicit information is, in the long run, hard to understand and can hide subtle bugs that

```
std::vector<int> result;
for (int n = 2; n <= 100; ++n) {
    bool prime = true;
    for (int i = 2; i < n; ++i) {
        if ((n % i) == 0) {
            prime = false;
            break;
        }
    }
    if (prime)
        result.push_back(n);
}
```

Figure 1.4: Primes in C++

```
[n | n <- [2..100], not . any( \i -> n `mod` i==0) $ [2..(n-1)]]
```

Figure 1.5: Primes in Haskell

would be made obvious by a verbose algorithm. Or maybe the verbose algorithm forces the programmers to focus on technicalities and the added complexity will lead to more errors.

But computing primes is likely not the right task to compare languages. For a more complex program, we could, for example, write a web server. It is a reasonably well specified task that can be automatically tested for errors. One could set the duration of the experiment to one month, a decent expectation for such a problem. After a month, programs in both languages can be subjected to extensive testing and compared to determine which language is better. To make sure the experiment is controlled, we would have to ensure that all variables are identical. In particular that the programmers implementing both versions have identical skill levels and experience in their respective languages. One could select them from a pool of candidates with an advertisement that might look like this:

> Looking for programmers with exactly 5 years of industrial experience and experience in server programming. Applicants are required to submit their CV with emphasis on their programming language and application development skills.

Although a technical discipline, programming is dependent on human factors that are hard to control for. Even after carefully matching their resumes, the closest pair of programmers will differ in a myriad of ways. Those differences, ranging from upbringings and personal traits all the way to the quality of their teachers and the first programming language they became comfortable with, may have substantial impact on their performance. So it is not unlikely that a month later, instead of settling the language war, we would have only poured petrol on the fire.

Software engineering is not the only discipline where controlling for humans is needed. Consider clinical drug trials: faced with the similar challenges, doctors sort patients into a control group that is not given a medication and the test group that is. The results for each group are averaged under the assumption that variables not controlled for will be similarly distributed in both groups. In simple terms, if the latter group recovers faster, the drug is

effective. Larger samples effectively allow to reclaim the uncertainty due to the uncontrollable variables.

The same can be done for our language comparison problem. Instead of one programmer, we can choose many. We no longer have to pay so much attention to their skills, as long as their distributions are similar, which is an easier task. If one group of programmers is, on average, able to produce less buggy code, then their language is the winner.

Alas, many professionals would object to our methodology on the grounds that C++ has not been designed for writing web servers. Other languages, are better suited to the task. To silence them, we could rerun the experiment with a different programming assignment only to hear similar objections from the Haskell folks if C++ comes up on top this time.

But let us not despair, for we can use statistics again. Perhaps the actual programming task itself should not be controlled – what if we gave multiple different programming assignments and again averaged the results. Clearly, this will settle it.

Maybe, but likely not. The devil lies in the details. According to Statista [7], Haskell and C++ salaries averaged to a $10K per month. For our experiment, we require 200 man-months for a total cost of $2M for a single programming assignment. Since this amount surpasses many research grants, it should be obvious that our proposal does not survive contact with reality, and the win of C++ or Haskell remains as elusive as ever. And indeed, for much of the short history of the computer science, studies about the qualities of programming language design similar to our proposal were scattered far and few between, and of limited impact. Yet with more and more software being written and penetrating still larger parts of our lives, the need to make programmers measurably more productive has only grown in importance.

There are adjustments we can make to lower the costs. Shortening the duration, simplifying the task, reducing the number of developers, or even using cheaper (less experienced) developers (students). But those compromises come with their own drawbacks. Shorter duration means potentially less feature complete programs (i.e., instead of bugs we will have missing features), simpler tasks mean less pressure on the language features to prevent bugs, fewer developers impairs the cancellation of their differences and less experience shifts the results from measuring the language efficiency to that of the speed of its acquisition. No matter what we do, we will be trading practicality (cost) for signal (results).

1.2 Analyzing Software Development

The biggest hurdle in the experiment is the practical impossibility to develop software in a controlled fashion. This is why we had to create an experiment of our own instead of turning to already developed programs (surely, there has already been a web server developed in C++ and Haskell). We overcame our inability to control for the human condition by analyzing and averaging more humans. The same could be done for software: Instead of one carefully controlled experiment, we can grab thousands of uncontrolled ones and average the results. This will require even more statistics - the software analyzed will not have identical, not even similar specification, it will be implemented by teams of various sizes and from various backgrounds. The development process will differ too. But if we can observe the development process in enough detail, it might just be possible to refine the results enough for a statistically significant answer.

To summarize, for our new approach, we require:

- access to very large amount of software,

- knowledge of their development process (teams, developers, changes, sizes, etc.),
- ability to infer the number of bugs to analyze the efficiency of the development,
- a statistical framework to correctly interpret the results,
- and a fully automated analysis pipeline due to large data volumes

15 years ago, we would have already hit the wall with the first item, as most code was developed privately by companies or individuals. However, recent evolution and widespread adoption of version control systems with advanced features combined with the shift towards open source and visible development process, have dramatically improved our ability to obtain large bodies of code and analyze their development process in detail:

Version control systems (VCS), first introduced in [13] allow programmers to record and track changes to the source code by storing smaller changes to the program in batches called commits. Each commit usually contains a text message explaining the changes to the source code themselves as shown in figure 1.6. Version control systems impose order on commits made by different users, help dealing with conflicts (i.e. when two developers alter the same part of the program at the same time) and allow reverting the code to any previous state. They quickly matured to *distributed version control systems* offering the same functionality for entire teams of developers collaborating on a single project. Their use skyrocketed when `git`, a decentralized distributed VCS conceived by Linus Torvalds was published.

With the increased availability of internet bandwidth and the transition towards cloud based services, several organizations started providing internet hosting for version control

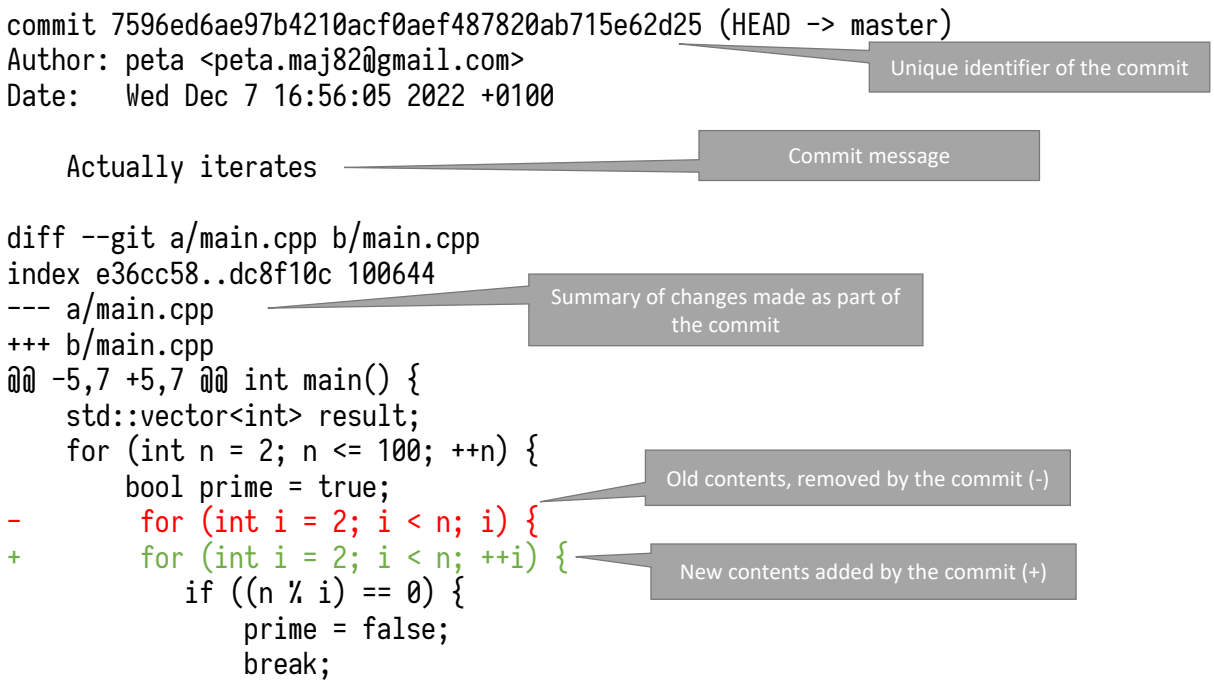


Figure 1.6: Example of a `git` commit with timestamp, author, commit message and changes to the source code visible.

systems around the turn of the century. The versioned source code thus moved from a large number of private company servers and personal machines to a few large online *software repositories*.

Although the software projects were now physically located in a few very large software repositories, it was still private, accessible only to its authors. In a bid to increase their adoption, many repositories offered their services for free to open source projects. Those projects are developed in the open and while only their authors can make changes, anyone can download and analyze their code. Initially picked up by hobbyists, large companies took notice and started developing some of their software in the open en masse. The State of the Octoverse for the year 2022, an annual report published by GitHub, the largest software hosting provider, dedicates an entire section to the open source projects owned by large companies, stating that *"In 2022, some of the largest open source projects on GitHub were owned, led, or maintained by companies. How those projects are growing reveals broader changes in how developers—and organizations—are building software"* [9]. One of the best examples of the magnitude of this shift is Microsoft: A company well known for fiercely protecting its codebase started using public GitHub for some of its projects as early as 2012, later becoming one of the top open-source contributors, including very large projects such as Visual Studio Code, the JavaScript engine for the original Edge browser and the Windows Terminal.[1]

This brings us to the last important feature of modern software development process that makes it amenable to data mining: *metadata*. The collaborative development goes far beyond having the source code and its changes in the open. Code hosting providers often provide additional features such as discussions about the committed changes, *issues* raised by the developers or users coupled with the changes that fix them, *pull requests* (in fig. 1.7), which consist of a change to the code proposed by developers and their evaluation, *continuous integration* that executes various checks upon each commit or pull request to ensure that new changes do not introduce new bugs, *release management* that allows labelling certain commits as versioned releases of the software and many more. These features are widely used by external developers as well: While responsible for only a small percentage of source code development, they engage in comments to new features, issues and pull request review [9]. The metadata is often provided in computer readable formats so that they can be visualized by external software development tools such as task planners or code coverage tools. This further increases the ease of its mining (an abridged version of the same pull request in JSON format is shown in figure 1.8).

So here we are: the information about the entire software development process for millions of projects is at our fingertips. We have access to source code of millions of public projects. Better still, those projects vary from single person projects of passion to large open source applications and to applications developed in the industry. Thanks to the version control systems we can reconstruct the historic record of how their source code changed. And thanks to the metadata, we can correlate those changes to new features, bugs, their fixes and much more.

[1] Microsoft seems to be focused on increasing its open development even further, as in 2018 the company purchased GitHub

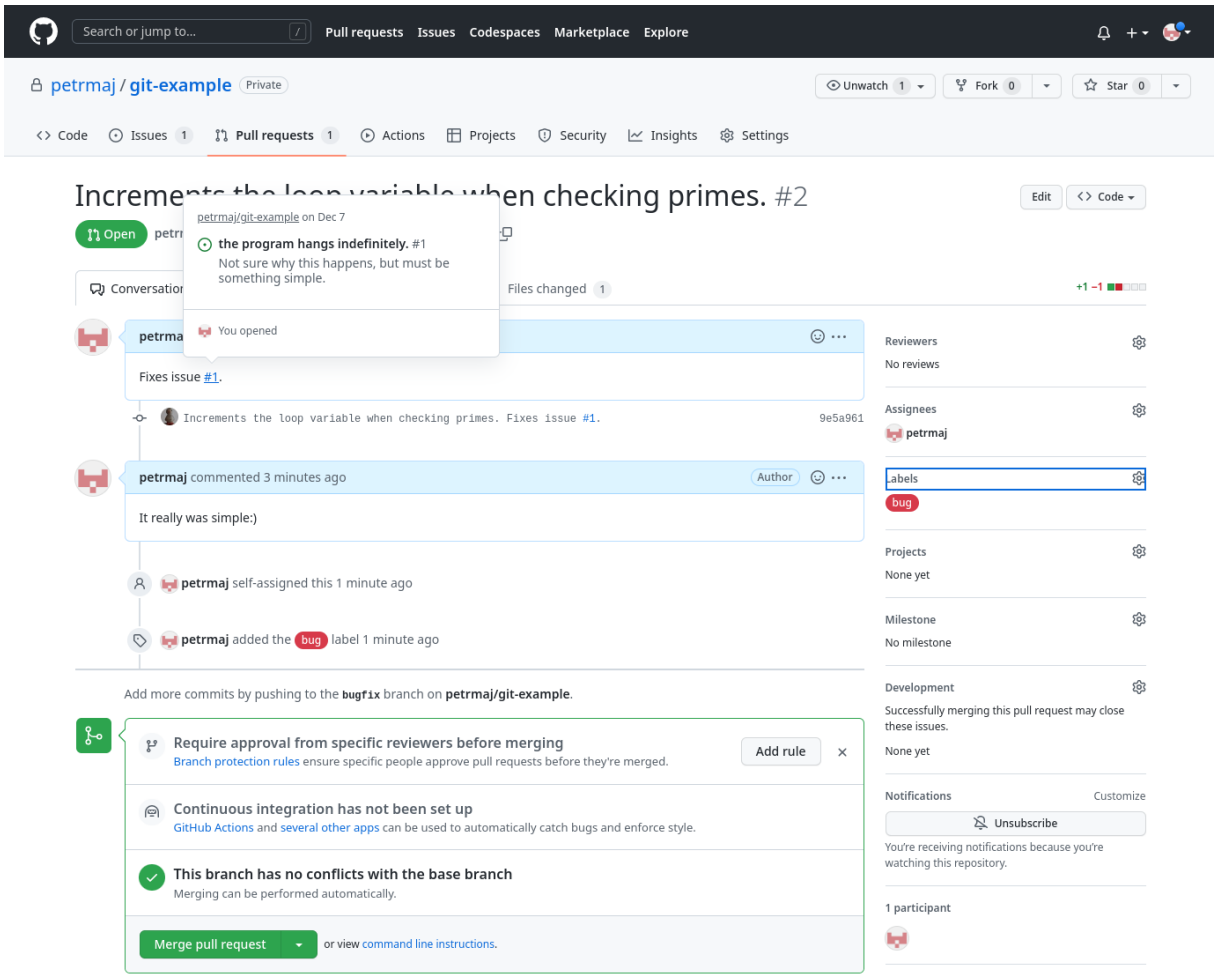


Figure 1.7: Pull request, as visualized by GitHub. Note that the pull request is automatically linked to an issue describing the problem, can be labelled as a fix, contains a discussion about the features, lists outputs of tests and so on. All this information and the links are available for mining.

1.3 Motivation

Researchers recognized the potential of analyzing large software repositories and the method has been used in many papers. In 2004, it even gave rise to a specialized conference, Mining Software Repositories. The initial research was merely setting the stage, operating on smaller datasets and exploring the boundaries of what can be analyzed. But in 2014, a paper titled *A Large Scale Study of Programming Languages and Code Quality in GitHub* appeared [12]. The paper is notable as being one of the first attempts to utilize the wealth of available projects and their development information to answer questions that are too costly to answer by classical experiments. The study had a wide impact with over 470 citations recorded on Google Scholar (December 2022) as well as numerous mentions outside the academia. It was further selected in 2017 for a reprint in the prestigious *CACM Research Highlights* [11].

The paper aimed to settle the debate about programming language design qualities using the approach we have arrived to previously. Instead of artificially creating new software in a

```
{
    "id": 1151891156,
    "number": 2,
    "state": "open",
    "title": "Increments the loop variable when checking primes.",
    "user": { "login": "petrmaj", ... },
    "body": "Fixes issue #1.",
    "created_at": "2022-12-07T16:04:07Z",
    "closed_at": null,
    "merged_at": null,
    "assignee": { "login": "petrmaj", ... },
    "requested_reviewers": [],
    "labels": [
        { "name": "bug", ... }
    ],
    "head": {
        "label": "petrmaj:bugfix",
        "ref": "bugfix",
        "sha": "9e5a96188f1f7944dcfc4fbed3fa87efcceff69e",
        "user": { "login": "petrmaj", ... },
        "repo": { "id": 575482550, ... }
        ...
    },
    "base": {
        "label": "petrmaj:main",
        "ref": "main",
        "sha": "461254f0168b710c3ed398326e061a5cdecefbe5",
        ...
    },
    "comments": 1,
    "review_comments": 0,
    "commits": 1,
    "additions": 1,
    "deletions": 1,
    "changed_files": 1,
    ...
         }
```


Figure 1.8: Pull request obtained in JSON format, abridged.

tightly controlled experiment, the study gathers a large set of existing projects (50 projects per language, 17 programming languages, including C++ and Haskell) and their commits. Instead of manually analyzing the bugs, the study turns to the data itself for this classification: It determines for each commit whether it fixes a bug (based on the presence of selected keywords in the commit message). This gives a ratio of bug fixing commits for different languages.

Clever statistics is then used to verify that all uncertainties have indeed been controlled for and do not influence the outcome. For each language, a coefficient determining its propensity towards bugs is calculated. Positive coefficient means the language does, on average, introduce more bugs, while negative coefficient indicates a language that spares its users some of the programming pitfalls. For the two languages we are interested in, the paper reports coefficients 0.23 for C++ and -0.23 for Haskell, both with very high statistical confidence of 99.9%. Those results tell us C++ projects contain generally more bugs than average programming language,

while Haskell projects are marred by much less errors. All other factors being averaged or controlled for, this difference is due to the language selection alone, i.e. C++ is worse language than Haskell. Informally put, if we choose C++ for an implementation of our project, we should be prepared to deal with more bugs than we would have to if Haskell was chosen.

But let us not jump to conclusions. While the power of big data analysis is tempting, big data comes with its own, equally big problems: The bigger the dataset, the noisier it is. The noise comes from various sources: version control systems are advantageous not only to source code, but to a wide variety of tasks that store their information predominantly in a textual representation. Software repositories such as GitHub thus consist of not just programs, but also books, examples, code snippets, programming tutorials, webpages, research papers (including this thesis) and so on. Even when focusing on actual programs, the noise levels are significant. The projects are developed by people of varying skill levels and teams of varying sizes. The majority of programming projects are short-lived and abandoned personal projects or student assignments. Such projects are much less likely to follow the software development discipline. Large software repositories are also full of copies of both the actual code, and of entire histories of programs. Apart from the practice of copying verbatim source code snippets or entire libraries into projects that use them, GitHub allows users to copy any project with its entire history for either collaboration or customization purposes. Such copies are called *forks* and they comprise up to a half of all projects hosted on the platform.

For analyses such as the language quality paper outlined above, this is bad news: If the results are obtained from student projects, the results will likely not generalize to experienced developers. Analyzing projects that are not rigorously developed will invalidate the methods of the paper (bugs not labelled as such and mentioned in commit messages). Analyzing copies of same project will skew the results towards those observed in that particular project as opposed to the general trend. To make matters worse, such projects comprise the bulk of GitHub so if a simple random sample were chosen, any analysis would be garbage-in, garbage-out.

To mitigate those issues, datasets obtained through large software repositories (same as any big data source) must be thoroughly filtered and cleaned before the analysis, otherwise they risk invalidation of their results. This warning is not hypothetical, the very paper on which we demonstrated the benefits of large software repositories suffered exactly this fate in [A.2] and [A.1] (those papers are part of this thesis and are discussed in greater detail in chapter 3).

But why is it that even popular research papers focused on data mining fail in data filtering and cleaning? We might brush the problem off as negligence, but before we do so, let us imagine what a reasonable data preprocessing step would be for our motivation example and how it can be accomplished with existing tools.

We need projects in the languages we are interested in that are developed rigorously. This notion of rigorous development has to be mapped to known characteristics, such as:

- majority of changes in one of the programming languages we are interested in,
- more than one developer to increase the likelihood of proper development process and descriptive commit messages

- at least 50 commits and more than six months of active development so that there is enough chance to observe the bugs being discovered and fixed
- the projects should not contain copies of each other to rule out over-representation
- active use of issues that implies bugs are detected and their fixes are tracked

Could this be done? The data exists. GitHub alone consists of 230M publicly available projects we can download with all their metadata as well. The filtering and cleanup steps are simple and easy to describe. Had the data been in a relational database, a few lines of relatively simple SQL code would do. But GitHub and other source code hosting providers are not relational databases. Their entire workflow focuses on working with a few projects such as those one contributes to, not project discovery and analysis.

The only way to get the projects we need is to discover all projects on GitHub through the existing API (a few weeks of work thanks to rate limiting) and get their metadata (many years!). The projects we are interested in would then need to be cloned (some more months), deduplicated, random sampled and only then can they be passed to further analysis. While there are tools that can help with some of the tasks, such as deduplication, they require considerable effort to make them work with the data at hand. Furthermore, at the end of the year+ effort the data will be old, some projects more than others, which will likely be a problem.[2] If our filter needs some tweaking, such as longer lifetime, or different language, some of the time consuming stages above will have to be repeated unless we have kept the vast quantities of data, a problem of its own.

Simply put, mining software repositories for precisely selected projects in reasonable amounts of time is impossible. This lack of functionality is what drives many researchers to compromise. Instead of carefully curated subsets, they use readily available samples, such as the most popular projects, one of the very few project properties GitHub allows semi-deterministic searches for. This method, known as *convenience sampling* produces inferior results as the analyzed datasets are likely biased.

And this lack of functionality is also the motivation for this thesis.

1.4 Thesis

Thesis statement: **It is possible to create an infrastructure that automates precise, scalable and reproducible selection of software projects from repositories.**

All the above features are requirements for reliable and targeted data acquisition of software projects without bias:

- *precise* - no information should be lost in the infrastructure and any information can be queried for so that the queries can be as precise as possible for the best signal to noise ratio.
- *scalable* - as the number of publicly available projects reaches hundreds of millions, scalability is a primary concern. Tens of thousands of new projects are created every day, while existing projects are evolving as their bugs are fixed and new features added.

[2] novel features, such as JavaScript ES6 classes, can be introduced in the middle of the data gathering, skewing the information about its adoption rate, etc.

- *reproducible* - asking the same question over the same dataset should yield the same result. Despite the continuously evolved dataset, it should be always possible to revert the dataset to any previous moment so that filtering queries can be repeated and modified long time after they were used.
- *automated* - the enormous data volumes involved require that the entire infrastructure must be able to perform without any human intervention other than specifying the filtering criteria.

This thesis introduces four research papers that together advance our understanding and practical usability of large software repositories:

1. *A Map of Code Duplicates on GitHub* [A.3] analyses source code clones present in GitHub projects. It verifies the existence of one of the most common biases and shows its scale. Our findings signify the necessity for dedicated project selection and filtering steps in big code analyses.
2. *On the Impact of Programming Languages on Code Quality* [A.2] is a reproduction study focusing on the data filtering, reproducibility, and statistical interpretation of large corpora analyses. The paper shows the problems pointed out by this thesis are present in contemporary research and that they affect our results.
3. *Reproducible Queries over Large-Scale Software Repositories* [A.1] introduces the infrastructure that forms the statement of this thesis: a scalable, precise, deterministic, up-to-date and reproducible project selection pipeline.
4. *How to Design Reproducible Large-scale Code Analysis Experiments* [A.4] then devises and argues for an explicit and rigorous project filtering step and demonstrates how it can be done with the tool presented in the previous paper.

1.5 Structure of the Dissertation Thesis

1. The *Introduction* describes the motivation behind the thesis and states its goals.
2. *Background and State of the Art* surveys the past and current solutions available for mining software repositories.
3. *Overview of Contributions* summarizes the author's work towards more scalable and reproducible large software repository mining.
4. *Relevant Papers* presents a collection of the author's papers that form the basis of the thesis' contributions. Each paper comes with a detailed list of the author's contributions.
5. *Conclusions* summarizes the results of the thesis, hints at possible improvements and future work.

Chapter 2

Background and State-of-the-Art

At its heart, a software repository comprises of two essential features: a data source containing the projects and associated metadata, and an interface allowing querying and retrieval of the dataset. The size and composition of the repository define its theoretical usefulness, while the query precision and retrieval speed of the interface determine its practical usability. This chapter presents an overview of current and past software repositories from those angles.

Creating and maintaining a large database of software projects is a complex undertaking. Therefore, there exists only a relatively small number of such data sources. They are not built to support a niche task like data mining but focus on aiding the software development process itself. A larger number of software repositories follow a different approach and instead of maintaining their own unique source, to reduce the costs, they choose to mirror an existing source, or its portion (in both number of projects and kinds of data stored). Those secondary repositories then provide a more complex interface to data querying that is better suited for data mining.

This chapter splits the discussion about existing software repositories into first discussing the primary data sources available in terms of their composition and volume. It then looks at the software repositories from the querying and retrieval perspective. A software repository that also maintains its own data source thus appears in both sections.

2.1 Sources

A software source can be characterized by the following main attributes:

- *Size and bias* - the number of projects available in the source and any bias associated with it. For all the sources mentioned in this section, one has to accept the bias towards publicly available open source software, but other biases, such as programming language, licensing terms, project popularity and so on are mentioned where appropriate.

- *Contents* - data sources differ in the data they store. Historically the first sources were built around version control systems hosting providers and therefore contained the most recent code along with a history of changes. As development switched to open distributed model, extra metadata about the software, such as code reviews, regression test results and so on become available.

The rest of this section describes the main software sources in the above terms.

2.1.1 GitHub

Established in 2008 as an online hosting for `git` projects, its key advantage was the inclusion of a free plan that allowed individuals and companies to host unlimited number of open source projects with paid plans for private and closed projects. Over the years, this policy and the rise of `git` itself has made GitHub extremely popular amongst programmers. This success has made GitHub the hegemon in terms of number of software projects stored: as of 2022, there are over 230M of publicly available projects and the total number of projects hosted might well attack half a billion.[1] The service is used by over 73M developers [8].

As its popularity increased, GitHub expanded into a complete platform for distributed software development, integrating a plethora of extra services under its name. GitHub provides project web hosting, issue tracking, code review process, continuous integration builds, release management, deployment and much more. This vast portfolio of extra services and their widespread use, as most are still offered free of charge for open source projects has made GitHub a treasure trove of both software projects' code and extensive details about their entire development process.

All this makes GitHub the single most complete source currently available. But this popularity comes at a price, especially when mining software repositories. The notoriety of GitHub and the amount of extra services it sports has led its use to transcend the original software development niche. Large amount of projects hosted on GitHub are not software per se (including, but not limited to hosted web pages, book manuscripts and documentation). Even larger amounts are pieces of software that were never intended to be developed. GitHub is a popular vehicle for conducting computer science courses (with repositories automatically generated in vast numbers for all enrolled students), showcasing demo applications and even a dump site for abandoned projects for archival purposes.

Reproducibility is also an issue. Being oriented towards the development process itself, GitHub does not provide any guarantees about future availability of its data. Projects can be deleted or made private, their histories can be altered, or even purged, and none of these changes are archived. The new content, or lack thereof, simply overwrites the past data with no going back.

2.1.2 Bitbucket

Bitbucket was founded in 2008 as a hosting service for Mercurial, another version control system. Following the upsurge in `git`'s popularity, it was added as option to Bitbucket as well. Then in 2020, Mercurial support was removed, cementing `git`'s dominance. Bitbucket is much smaller and thanks to its policy of allowing privately hosted repositories for free from the very beginning, it contained much less open source software. In terms of repository contents, Bitbucket is similar to GitHub, it is a primary repository, offering own bug trackers, discussions, pull requests and continuous integration.

It is much harder to determine Bitbucket's size - the ratio of private projects is likely higher than in GitHub, but no yearly reports are published. Furthermore, it is impossible to guess the total number of projects as Bitbucket uses unique text identifiers instead of

[1] Since it is impossible to distinguish between deleted and private projects on GitHub, we can only determine the total number of projects created.

consecutive ids. A reasonable estimate of 3.4M public projects can be obtained from archival sites' records.[2]

2.1.3 Other Version Control Systems Hosts

Numerous other services similar to GitHub exist. As their usefulness for software repository mining pales with the comparison of GitHub, they are only briefly discussed in the following paragraphs:

GitLab GitLab is a service bearing striking similarity to GitHub itself, with a major twist: GitLab is geared towards a self-hosted deployment, making itself less appealing to repository mining as large amount of GitLab instances would have to be scanned for reasonable number of projects to be acquired. In addition to self-hosted option, GitLab also provides cloud hosting, with the number of projects available reaching 4M.[3]

SourceForge Created to support open software projects and their development directly with no paid options, SourceForge provides capabilities similar to already mentioned services. Its membership stands at over 500K projects.[4] SourceForge supports multiple version control systems: in addition to the popular `git`, Mercurial, CVS and others are supported as well.

2.1.4 Package Managers

Package managers provide software developers with access to large numbers of third-party libraries available for their programs that can be easily integrated into applications. Instead of the continuous development process supported by version control systems, package managers focus on the releases - updates to the libraries made explicit by the developers, usually following the semantic versioning pattern.

Package managers contain less noise in the form of non-software projects, and often provide extra metadata about the usage and downloads analysis of the packages that can be used to further filter interesting projects. Historic reproducibility is also better than version control systems as older versions of packages are kept for backwards compatibility. Unfortunately, a package can still be withdrawn by its developers, such as the infamous withdrawal of `leftpad` in 2016.[5]

In terms of size, package managers for the most popular platforms, such as JavaScript, reach millions of libraries, while less widespread languages such as the R programming language used mainly for statistics with its CRAN package manager provides only 20K packages.

Their biggest weakness is strong bias towards library code as virtually no applications (standalone executables) are part of any package manager. Package managers are rarely the primary source of the code as most of their content is available from version control systems, notably GitHub.

[2] Number of Bitbucket projects in Software Heritage corpus is about 2M projects, compared to 136M for GitHub. Using the same ratio of completeness for both providers would give us 3.4M public projects.

[3] Via Software Heritage

[4] `https://sourceforge.net/about`

[5] `https://www.theregister.com/2016/03/23/npm_left_pad_chaos/`

2.1.5 Software Heritage

Started in 2015 by Inria, the Software Heritage Project [2] aims to preserve the large code base mankind has created. It archives software projects from various primary sources including GitHub, Bitbucket, GitLab, CRAN, SourceForge and Debian source packages. The project is actively maintained and updated via means of automated crawlers or direct access by partners. As of Fall 2022 the Software Heritage has archived over 184M projects. GitHub is the major source with 136M projects, followed from a distance by GitLab (4M), Bitbucket (2M) and NPM (1.8M). All other sources contribute less than 1M projects. Compared to development-oriented platforms such as GitHub or BitBucket, Software Heritage stores only version control systems' data (file changes over time and commit messages).

Although technically not a primary software repository itself, we classify Software Heritage as one for the purposes of this thesis due to its archival nature. It is the only such source that provides historical reproducibility - when a new version of a software project is found a new snapshot is added to the repository. Knowing the snapshot identifier, one can always access the previous data.

Software Heritage has two major drawbacks: (a) the rate of snapshot updates is unpredictable and long, which biases the dataset towards historical studies, not current trends. (b) it archives the source code and its history only, not the extra metadata such as issues, discussions, etc.

2.2 Software Repositories

After describing the available sources, we focus on software repositories from the querying precision and retrieval performance point of view, where a repository can be characterized by the following key attributes:

- *Source* - whether a repository maintains its own primary source, creates a mirror of one, or simply provides a frontend to another repository.
- *Activity* - active repositories are accessible and can be used. Inactive repositories are included for their historical significance.
- *Updated* - Some repositories offer a single view of the projects they store, while others are regularly updated at varying intervals, or are the primary sources themselves.
- *Query precision* - while each repository provides *some* form of querying the projects it contains, the expressiveness of the queries is a limiting factor.
- *Deterministic* - a deterministic repository will, for a given query, return always the same answer as long as its underlying dataset remains unchanged.
- *Reproducible* - a reproducible repository goes beyond simple determinism by requiring that a query can be constructed in such a way that identical results are returned even if the underlying data gets updated in the meantime. Non-updating repositories achieve reproducibility trivially; for updating repositories, especially the primary ones, reproducibility is usually sacrificed as projects may be deleted or their history altered using version control systems.

To better illustrate the usefulness of the described repositories, we roughly describe how each can be used to obtain a dataset that could be used for our motivation example. Recall that we need projects that have enough development to make the bug fixing commits appear in reasonable numbers. We have thus searched for Haskell and C++ projects that actively use issues, have more than one developer, more than six months of active development and at least 50 commits. We will need their commit messages, the changes to the source code the commits made and their issues. As this is a rather low bar, we expect that more projects fulfill our criteria than we can reasonably analyze and would therefore require a random sample of 10K such projects.

2.2.1 GitHub

As well as the largest primary source, GitHub is also widely used as a software repository for querying and retrieval. GitHub is constantly updated as projects are changed, or indeed created. Those projects can also be deleted, made private, or their histories may be overwritten destructively. As GitHub does not keep any historic records other than those maintained by `git`, it is not reproducible. Multiple ways of accessing the stored data are offered:

- *Git* The simplest method is to use `git`, the underlying version control system, to retrieve the contents and history of hosted projects. Such access exists so that developers can obtain the projects they are involved with and upload their changes. It therefore features no project filtering, or even discovery capabilities as the project to be downloaded must be known by other means. Only information maintained by `git` itself is accessible using this method, i.e. no metadata. GitHub imposes no official data rate limits on downloading repository contents via this API, but we have observed heavy throttling for continuous multi-process accesses. Despite this, cloning GitHub projects remains the fastest method of getting projects and their commits en masse. The `git` access is deterministic, but not reproducible.[6]

- *REST API* Provides the complete access to all data available on GitHub. The API is geared towards programmatic inspection and manipulation of the few projects one develops, such as automatic releasing, code scanning, pull request alerts and summaries, etc. The API also provides an endpoint capable of searching for projects based on simple queries that allow filtering based on user or organization name, project description and readme file contents, project size in bytes, number of followers, forks, stars, creation and last update time, programming language used, topics associated with the project, issues ready for contribution, and other project properties (mirrors, forks, archived projects, sponsors). Limited sorting is supported (by stars, forks and help-wanted issues). Its intended use is to promote community involvement, not any form of mining. This is further exacerbated by the limitation of at most 1000 search results per query, which makes constructing larger datasets impossible. A query must not be longer than 256 characters and can contain at most 5 clauses. The query results are not guaranteed to be deterministic - if a query times out, partial results found so far are returned. It also provides a special endpoint exempt from the 1000 results limitation, which allows listing of public projects in the order of their creation, providing the discovery of all

[6] modulo history overwriting changes and project deletions, the access can be made reproducible by keeping the latest commit returned and then pruning the newer results.

public GitHub projects. The API has rate limits of no more than 5000 requests per hour by a single user, whereas the search queries incur additional rate limit of no more than 30 requests per minute.[7]

- *GraphQL API* This API offers more complex queries to be formed, but its main advantage is the precise control over the data returned and the ability to construct a single query that would have required multiple REST API endpoints. For instance, to return all mergeable pull requests of a repository, the REST API would need a call to determine pull-requests of a repository first and then a call per pull request to determine whether it can be merged. The GraphQL API can achieve the same result in a single query. For the purposes of data mining though, it suffers from the same drawbacks: limited query power as it is not designed for project discovery, severe rate limiting, inability to fetch *all* results (GraphQL queries are limited to 500000 nodes, the meaning of a node depends on the exact query), and non-determinism.

- *Web search* Not intended for automated use, the web search is a search bar within GitHub's webpage. On top of searching for repositories via same queries as the REST API, the web search also allows searching for particular files and even matching over the file contents. Web search results are non-deterministic even for simple queries that do not timeout, presumably due to load balancing of the requests.

In order to download the random sample from the motivating example, we would first need to list all the public projects available, then obtain their metadata, such as major language and number of commits, and then their code and issues themselves. The strict rate limiting policy means that getting all projects would take 19 days (230M projects, 100 projects per request, 5000 requests per hour). Getting the metadata at a cost of a request per project would take 5 years, providing us with some information about presence of issues as well.[8] Downloading the project's source code and commit history would best be done by the `git` access as this avoids the rate limits. We can use the commits history to determine if there is enough commits and enough time has passed between the first and last one. Issues for the sampled projects can be obtained through the REST API in hours (assuming 100 issues per repository on average).

Those timelines are squarely outside of the realm of practicality. One can be clever and, for instance, download all projects first (roughly 2 years of work), or sample earlier (needs to be carefully designed to prevent bias and only works for popular languages), but the amount of work and time required remains too high. Instead, researchers often compromise in the dataset description. Limiting ourselves to a mere thousand projects per language, using popularity instead of number of commits as an indication for project development and selecting top projects instead of random sample would give us results in mere minutes as we would be able to use the search API. It would be wrong, but tantalizingly easy.

[7] GitHub also uses secondary rate limiting which may be activated at any time GitHub suspects overuse of its resources, details of which are not publicly disclosed.

[8] Not ideal, since GitHub would only report the number of open issues, not whether issues are actively used.

2.2.2 Software Heritage

Software Heritage is another primary source. Like GitHub, it is active and updated. Unlike GitHub, the update rates are not instantaneous, but occur at irregular intervals that can take years.[9] Software Heritage is both deterministic and reproducible as different visits for the same project are all archived and can be retrieved separately.

As the repository contains projects from various primary sources, it offers a special API, called *Vault* that can asynchronously collect any archived project and export it as a bundle in a variety of formats, such as `git`. In this regard the vault service is essentially equivalent to downloading projects from GitHub.

Each piece of software is assigned a unique identifier and this identifier, project name, URL, and assigned tags are the only things Software Heritage can be queried on. The API is limited to 1000 results per request. It offers a REST API similar to that of GitHub with endpoints geared towards retrieval of known items, not advanced search and filtering. Rate limiting is more severe at 1200 requests per hour per authenticated user. Software Heritage is work in progress and it is likely the querying capabilities will increase in the future.

Using Software Heritage for our example would therefore suffer from the same weaknesses GitHub did, namely limited querying capabilities and rate limits and retrieval could take years. Since Software Heritage only captures the code and its history, we would not be able to obtain the related issues.

2.2.3 GH Torrent

GHTorrent started in 2012 as a scalable, queryable, offline mirror of data offered through the GitHub APIs. It monitors the GitHub public timeline, a special API endpoint that publishes many GitHub public events in a single stream. Those include project creation, new commits, starring a project, creating or closing an issue and many more. The stream allows GHTorrent to observe each such event in real-time and store them in its own database.

GHTorrent is currently inactive, there have been no updates to the project since 2020. At its peak, it archived activity for more than 150M projects. For some projects the events obtained through the public timeline were augmented with data outside of GHTorrent's lifetime, acquired via GitHub REST API. GHTorrent's database exists in two versions, a MongoDB dump of the raw GitHub public event timeline records, and a SQL version that contains the information processed into tables, such as basic project information, popularity, commits, messages and comments, issues and so on. Both MongoDB and SQL databases were searchable online, and can still be downloaded offline for local use. Rate limits are moot in the local download scenario and determinism and reproducibility were guaranteed via the monthly released snapshots.

The databases also form the querying and retrieval API for the repository. All of the archived information can be searched, filtered and ordered easily by complex queries that far surpass the ability of GitHub or Software Heritage. Even more complex queries can be calculated offline as GHTorrent archives enough metadata to allow calculation of various aggregated software engineering metrics.

[9] As an anecdotal evidence, this thesis author's own software *terminalpp* has been discovered and archived, but not updated in two years. Software Heritage offers manual trigger for selected projects in such cases, but this approach does not scale.

GHTorrent suffered various drawbacks. It lacked consistency; due to downtimes in the GitHub public timeline the dataset integrity is not guaranteed. While one could patch the dataset by crawling the projects, this is not enough as we have shown in our research [A.4]. Furthermore, it has bias in favor of projects that are actively updated; those which are not are simply absent. Last, it does not have source code, so it cannot be the ultimate source of truth.

Due to the lack of source code, GHTorrent alone cannot be used for our example. When used together with GitHub it greatly speeds up the process: The latest snapshot of GHTorrent can be downloaded and then queried for Haskell and C++ projects with sufficient amount of commits in a matter of mere hours. Since GitHub public events timeline also contains information about issues, these too can be obtained from GHTorrent at virtually no additional costs. The URLs obtained can then be used to download the projects' source code directly from GitHub. Thanks to this speedup, GHTorrent was used extensively in research, including own [A.3, A.2, A.1]. But the dataset has errors and holes, increasing the risk of bias in any subsets obtained by it as issues, stars, users, commits or indeed entire projects can be missing or wrong. Finally, while GHTorrent is deterministic and reproducible, GitHub is not. Over time, the source code that is not part of GHTorrent itself will differ or become inaccessible.

2.2.4 Orion

Orion [1] is a software repository targeted specifically to data mining. Its dataset is modest, consisting of 185K projects downloaded from GitHub, Sourceforge and GoogleCode (other version control system providers popular at the time). Complete file contents, commits history, metadata available at the time and a wealth of synthesized software engineering metrics are stored for each project. However, Orion's main focus lies in the design of a domain specific language for querying large software repositories and the implementation of its search engine allowing advanced search and filtering over all data items stored in Orion's database. The database does not support updates and therefore Orion achieved both consistency and reproducibility trivially. The project is no longer maintained.

Answering our example question would be very easy with Orion. Had the project been maintained still, the results would suffer from two main problems: (a) it is not clear the relatively small corpus would contain enough projects fitting our criteria, (b) since updates are not supported, the results would age quickly.

2.2.5 Boa

Like Orion, Boa [4] addresses the need for an efficient searching over large software repositories. Boa goes even further and strives to provide tools to mine specifically the source code itself. It supports not only searching the aggregated attributes, but also parsed abstract syntax trees of the stored source files. It then uses its own domain specific language based on the visitor pattern to allow constructing efficient queries over the syntax trees and project attributes. It offers the biggest expressive power from the tools reviewed. The queries are executed in parallel on a hadoop cluster.

Boa's dataset consists of 380K GitHub projects since 2015 and was extended to support Python (2020) and Kotlin (2021).[10] On top of the fully queryable 380K projects, the dataset contains additional 7.5M projects without the source code with aggregate metrics only.

The project is active, but the dataset is updated very infrequently with no changes to the Java dataset since 2015. Boa is both deterministic and reproducible via update snapshots.

Boa provides much more than our relatively simple example query requires. But that extra expressiveness comes at a cost: Adding a new language to Boa is a substantial effort as its files have to be syntactically understood. Neither C++, nor Haskell are supported. But even if they were, the infrequent updates (it's Java corpus is now 7 years old) make the results quickly obsolete, similar to Orion.

2.2.6 Other Repositories

We briefly mention other software repositories that are either much smaller, or only vaguely linked to the task of this thesis, but played an important role historically. None of them can be used for our example query reasonably as they are either too small in scope to provide relevant data, or not fit for the purpose.

Bitbucket Like GitHub, Bitbucket offers a REST API for accessing its data with only slightly better querying capabilities. Notably, anything than can be filtered can also be sorted. However, randomization of results is not supported and due to the use of textual unique identifiers for projects, random project acquisition is not possible either.

Flossmetrics This work analyzed 2800 open source projects and computed statistics about various aspects of their development process, such as number of commits and developers [6]. Information from additional sources such as project mailing lists and issue trackers was included. Queries could be formulated on metrics such as COCOMO effort, core team members, evolution and dynamics of bugs. Filtering based on these criteria was supported. The project is inactive, and it did not support updates.

Black Duck Open Hub A public directory of open-source software that offers search services for discovering, evaluating, tracking, and comparing projects.[11] It bears similarity to the older Flossmetrics projects upon which it improves on both quality and quantity, including continuous updates. It analyzes both the code's history and ongoing updates to provide reports about the composition and activity of code bases. The Open hub does not store any contents of the analyzed projects, nor does it keep historic data other than the aggregated metrics. However, since the links between Open Hub projects and their repositories are kept, and the querying capabilities over the analyzed attributes are extensible, the Open Hub can be used to bootstrap an analysis by selecting projects whose contents will be downloaded from a primary source. Open Hub does not support randomization of the results, but given its relatively small size, getting all the data first and then doing own randomization is indeed possible.

[10] Boa started with 490K Java projects obtained from SourceForge, but later switched to GitHub

[11] `https://www.openhub.net`

SourcererCC The single aim of this project is to detect code clones [14]. The tool scales to large datasets and can detect near-identical code at various granularity. It has been used to analyze cloning across large corpora of Java, JavaScript, Python, C and C++ projects on GitHub [A.3]. SourcererCC does not keep the metadata or code of the analyzed projects but keeps a hash of each file contents as well as a fingerprint obtained by tokenizing each file and remembering the token counts. Only source files in the four analyzed languages are kept. It could be used by researchers to detect duplication in their samples specified by links to GitHub projects, after which a report of cloned files found within the dataset was provided. The project is no longer active.

Stress One of the first attempts at reproducibility of project selection, Stress works either locally, or online [5]. Its accompanying paper surveys the reproducibility of project selections in 68 studies and finds none to be completely reproducible. It then proposes a selection tool that allows extensive filtering based on the project information and 100 synthesized arguments from the projects' version control data and metadata, such as project lifetime or open tickets. The tool is verified on a corpus of 211 Apache projects. Stress supports queries to be stored and repeated later. Querying over source code is not supported. Stress is no longer active.

2.3 Summary

In Table 2.1, a summary of the software repositories discussed in this chapter is given in terms of their numbers of projects (size), whether they are a primary source of data (sources), if they are actively maintained (active), whether queries can be run again with identical results (reproducible), whether queries only have the power to express basic project access, filter or are full-featured (query), and lastly, what attributes of a project are stored, these can include the code, versions, and metadata (contents).

For our data mining purposes, we need a software repository to scale in the number of projects and languages supported and to be up to date, to allow for powerful queries to be expressed, and to yield deterministic and reproducible results. For those requirements, none of the existing solutions are adequate.

	Size	Sources	Active	Updates	Reproducible	Query	Contents
GitHub	210M	primary	Y	continuous	–	basic	code, ver, meta
Software Heritage	175M	many	Y	continuous	Y	basic	code, ver
GHTorrent	157M	GitHub	–	continuous	–	full	ver, meta
Orion	185K	many	–	–	Y	full	code, ver, meta
Boa	980K	GitHub	Y	–	Y	full	code, ver, meta
Bitbucket		primary	Y	continuous	–	basic	code, ver, meta
Flossmetrics	2800	many	–	–	Y	filter	other
Black Duck	1.4M	many	Y	continuous	–	filter	other
Stress	211	Apache	–	–	Y	full	other
SourcererCC	4.5M	GitHub	–	–	Y	basic	other

Table 2.1: Comparison of repositories.

Chapter 3

Overview of Contributions

As we have argued in the previous section, analyzing large code bases requires identifying which software projects to study. Any mistake in the choice of projects to look at can introduce unwanted bias in the process and, eventually, even invalidate or skew the results of the analysis. This chapter provides an overview of the contributions of this thesis towards our understanding of the key challenges in the practice of big code analysis, and towards an automated, scalable, precise and reproducible infrastructure for selecting software projects from large scale open source software repositories.

Hundreds of millions of software projects with their histories and machine-readable metadata about many aspects of their development process offer an unprecedented wealth of information that, when mined properly, allows us to infer software engineering insights that would simply be out of reach using traditional experiments.

But large software repositories, as many other instances of big data, come with their own challenges. The enormous sizes of repositories such as GitHub, make gathering, and later analyzing the data, a complex process. Every step must be fully automated as these steps may be repeated many times by different researchers. Validating that those steps yield the expected result is hard as errors in big data analysis do not fail in a visible manner, but rather may corrupt the dataset or lead to unsound results. The reason for this is that insights are not directly observed by inspection of unique data points, but rather aggregated from millions of rows of results using statistical techniques that may obscure the errors further.

Torture the data long enough, and it'll confess, Ronald Coase famously cautioned. No big code step tortures the data more than project selection. The problem is that large-scale software repositories are extremely noisy. Most non-trivial analyses of software repositories give insights based on development patterns that can be observed. Chapter 1, which provides context and motivation for this thesis mentions a paper that tried to show that choosing a particular programming language impacts the number of bugs programmers will commit. To answer whether languages affect bugs, researchers make an implicit, unstated, hypothesis: namely that the software projects they study are representative and meaningful. We assume that the projects are typical software projects developed by professionals trying to follow best practices, and that the data is meaningful. In the particular example at hand, this meant that the commit messages contain descriptions that match the content of the commit, that bugs are looked for and eventually fixed, and that the `git` history is rich enough for us to be able to observe patterns. We will often use the, somewhat informal, term *developed project*

to denote such projects. But this hypothesis is wrong. The crux of the problem is that large scale software repositories also contain garbage. Or at least, to be charitable, projects that are entirely irrelevant to the analysis at hand.

Dealing with garbage is a problem in many areas. In our context, garbage is a byproduct of the success and usefulness of GitHub. Hobbyists use it for efforts that do not progress past a few commits before the project is abandoned and left to bit rot. Most of the world's students and their educational institutions use software repositories for their class assignments. Popular projects are forked (i.e. duplicated) many times over with only a few changes applied to their code. Going beyond software, repositories are also used to store text, configuration files, research papers, and even the thesis you are reading. Taking a random sample of GitHub will be disappointing. The overwhelming majority of projects one will find will be extremely small, have few commits and no meaningful metadata. Garbage.

And garbage is what this thesis concerns itself with. Namely, how to deal with it in our analyses. If it is true that most projects hosted by GitHub are irrelevant for virtually any software engineering analysis one could imagine, then it is clear that what is required is extremely careful selection to ensure only developed projects, as we call them, are returned. Garbage will only add noise to any code analysis task and possibly skew the results.

3.1 Mapping code duplication

Our first contribution in the thesis is documented in the OOPSLA 2017 paper titled *DejaVu: A Map of Code Duplicates on GitHub*, in which we have mapped duplication in GitHub projects written in four popular programming languages. We have analyzed file-level duplication, both exact and approximate and reported on the source and composition of the clones we found.

A proper project selection must not only ensure the absence of garbage, but it must also correct for any bias that might be present in the selection. One source of bias comes from code duplication which causes duplicated code to be over-represented in the analysis. This is no surprise, as code reuse is encouraged practice in software development. Common software functionality is packed into reusable libraries which are then included in other projects. And while some libraries serve only a niche of applications, others have become extremely ubiquitous, such as the *jQuery* library found in almost every webpage. Code duplication is not limited to libraries. Repositories often support a one-click copy of an entire project, including all its history into a *fork*, a new and essentially independent project. This practice is widely used by external developers wanting to work on a project they do not have authorization to update.

In a sense, duplication in software engineering is a feature, not a problem. This is only true if the duplication is explicit, such as in the case of forks, where the source is explicitly linked to the clone and therefore can trivially be removed from any analysis. We thus went after the more insidious, implicit duplication.

Our paper analyzed all non-forked repositories with code in four languages: C++, Java, Python and JavaScript. We found duplication rampant. Table 3.1 shows the summary of our findings in terms of exact and approximate clones. Java projects being the least affected at 40% of files being exact copies of others, whereas JavaScript was affected the worst with an eye-popping 94% files being cloned. We have attributed this high rate to the presence of node_modules directory where *npm*, the Node.js package manager downloads any library dependencies. While only a small portion of the projects included this folder, the files within

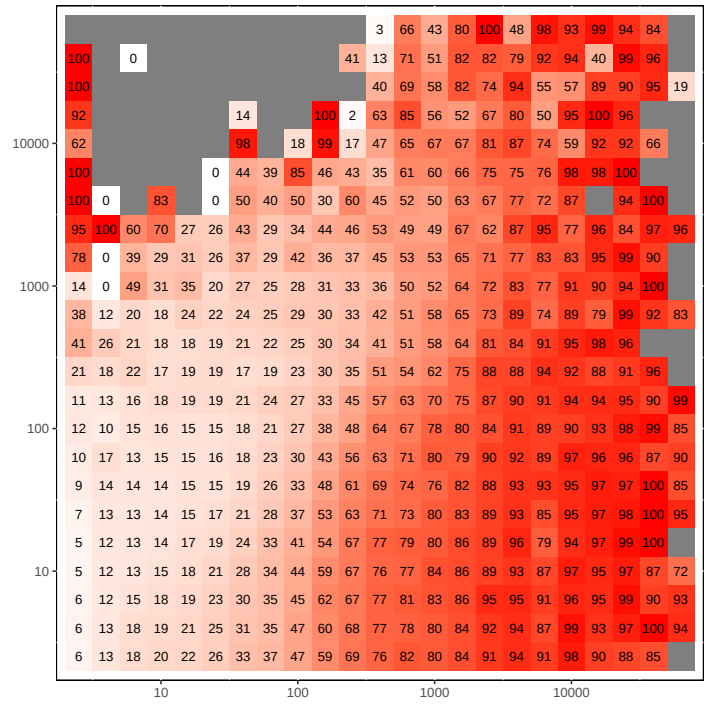


Figure 3.1: Map of code duplication in C++. The y-axis is the number of commits per project, the x-axis is the number of files in a project. The value of each tile is the percentage of duplicated files for projects in the tile. Darker means more clones.

accounted for a whopping 70% of the JavaScript dataset. Even after those files were removed from the corpus, JavaScript duplication rate was still the highest at 89%.

	Java	C++	Python	JavaScript
Total Files	72,880,615	61,647,575	31,602,780	261,676,091
File hashes	43,713,084 (60%)	16,384,801 (27%)	9,157,622 (29%)	15,611,029 (6%)
Token Hashes	40,786,858 (56%)	14,425,319 (23%)	8,620,326 (27%)	13,587,850 (5%)
>80% similar	22,085,265 (30%)	8,225,018 (13%)	5,887,579 (19%)	8,342,380 (3%)

Table 3.1: Code duplication on GitHub across four languages and three duplication thresholds.

Figure 3.1 gives a map of duplicates in one particular language, C++, and its relationship with two notions of project size. The y-axis is the number of commits per project, the x-axis is the number of files in a project. The value of each tile is the percentage of duplicated files for projects in the tile. Darker means more clones. The heatmaps show that as project size increases so does the fraction of duplicates and that projects with fewer commits tend to have more clones. Of particular interest are the very high percentages of cloned files found – including many projects with *no* unique files. The heatmap also shows various pathologies such as the unusually high duplication found in the cluster of very small projects with large number of commits. Those are a product of automatically generated commits with fake timestamps.

Another angle of our investigation was the search for any patterns in the duplicated code:

Manual classification of a sample of the cloned files revealed that most file-level duplication comes from the inclusion of libraries, while the near-identical files are often due to code generation (with the exception of C++ where the rate in the sample was only 15%, hinting to the limited use of code generation in the ecosystem).

Our paper has provided a proof of the enormous levels of duplication and confirmed it as a major source of bias that must be explicitly dealt with. Analyzing duplicates not only wastes resources, but also skews results. If, for instance, we analyzed JavaScript, the presence of *jQuery* clones would ensure that the results of our analysis would be a perfect representation of the characteristics of *jQuery* and little else. Furthermore, our analysis of duplication shed some light on the enormous number of often unexpected forms of garbage present in GitHub, such as the included *npm* packages, or the autogenerated fake commits.

3.2 Producing wrong data without doing anything obviously right!

Our second contribution, the TOPLAS 2019 paper entitled *On the impact of programming languages on code quality: a reproduction study* shows that garbage in software repositories not only exists but has an impact on our research unless dealt with properly. The paper is a reproduction study, an attempt to independently validate a previously published result. In our case, the original publication, which happens to be the paper we mentioned in our motivation, appeared at a software engineering venue. We will refer to it as the *original study*, and our work as the *reproduction*. We documented in excruciating details how easy it is to make mistakes in all phases of a big code analysis and how those the mistakes invalidate the result of analyzing a large corpus of programs.

The original study aimed to establish a correlation between the choice of programming language and the number of software defects in a project. In other words, to answer the question: *is* Haskell *better than* C++*?* Technically, the original study did not claim a causal linkage between language and bugs, but that is how the paper was widely interpreted, even by some of its authors.

The setup of the original study is, briefly, as follows. Select fifty most popular projects, as measured by GitHub stars, in seventeen influential languages. Then go over the commit history stored in GitHub for these projects and count bugs. Finally, use a statistical model to correlate bugs to project language. The results of the original study were statistically significant for most languages and comforting to our prejudices. Haskell is indeed better than C++. Generally, functional languages have fewer bugs than imperative ones.

The reproduction reviewed how the dataset was selected, as well as the data cleaning steps, and the statistical model. Our work identified multiple errors, mistakes and slip ups. When we corrected the errors, the results of the paper were affected, leaving very little conclusive evidence, and definitely nothing that would support the strong conclusions that were drawn from the original study.

To begin with, duplication in the dataset was not accounted for. As the data included the unique commit identifiers from GitHub, cleanup was as simple as removing rows with already seen commit ids. 1.8% of rows were removed in total, but some languages were affected disproportionally. Of the 33 projects with duplicate commits, 18 were related to Bitcoin (litecoin, megacoin, memorycoin, anoncoin, ppcoin, zetacoin and so on).

To attribute bugs to languages, programming languages used in a commit were identified by extensions of files touched by the commits. This attribution was not without problems

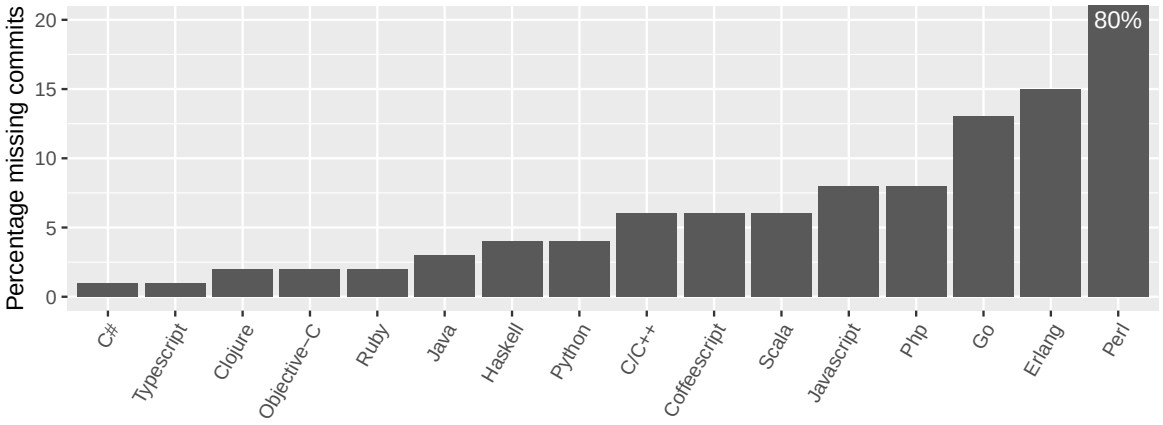


Figure 3.2: Percentage of missing commits in the original paper's dataset per analyzed language. Note that Perl is not to scale with the rest.

either. In case of C++, header files (*.h*) were classified as C irrespective of the language used and several frequently used extensions such as *.cxx* or *.cc* were omitted entirely leading to mistakes such as classifying V8, Google's JavaScript Engine written in C++, to only contain 7 C++ commits and be classified as the largest JavaScript project in the dataset. Other languages fared worse: translation files for internationalization were often classified as TypeScript, while its remaining files were mostly type definitions. As those contain no bugs, the original study reported TypeScript as the *best* language. Without a tool to acquire more data and label commits properly, that was missing from the artifact, we could only correct those by removing the seriously affected projects and languages.

We also embarked on a costly reproduction of parts of the data gathering step. We could not feasibly validate that the projects selected for the original study were indeed the most popular projects as of its writing: no exact time was given and even if it were, it is not possible to query GitHub for historic popularity efficiently. We focused on the completeness of the dataset instead - i.e. given the projects selected for the study, were all of their commits included in the dataset? This too, was surprisingly hard to verify as the artifact omitted project owners' names, necessary for project identification. Through careful reconstruction using the project names alone and their commit hashes we were able to identify 423 out of original 729 projects. Some were no longer available, some could not be identified from the data. We found significant number of commits from those projects to be missing in the study, 19.95% of the dataset. Figure 3.2 shows their distribution per language, with Perl being affected the worst with 80% of commits missing.

Taking a step back from the reproduction, our work identified four categories of challenges for analyzing large code bases:

1. **Selection**. Getting the right projects is tricky, as we argued earlier most repositories are littered with garbage. Finding developed projects that are representative of the variety in each language ecosystem is hard. In our case study, the failure points were the use of popularity as measured by stars and not accounting for duplication.

2. **Cleaning**. Data found in software repositories is surprisingly noisy. Mistakes in data cleaning accounted for a significant part of the issues our reproduction uncovered. Examples are misclassification of files due to errors in recognizing file extensions, and misclassification of commits as "buggy" due to imprecision of the bug detection oracles that were used.

3. **Modelling**. Statistical modelling is critical to analyze big data. We have observed an over-reliance on p-values, which are mostly of use for small data sets, and a general statistical naiveté. Collaboration with a practicing statistician was essential to build a sound model of the data and that model revealed how few of the results are, in fact significant.

4. **Reproducibility**. Being able to reproduce results is a key feature of good science. It is essential for big code analysis as any manual step is likely to become a bottle neck and a source of obscure errors. While we were lucky to have an artifact with code and data, it turns out that the code was missing some key functions and the data did not capture key attributes needed for reproduction.

3.3 Precise Project Selection

Our third contribution, the ECOOP 2021 paper entitled *CodeDJ: Reproducible Queries over Large-Scale Software Repositories* designed and implemented CodeDJ, a tool for precise, scalable and reproducible project selection from large software repositories. Our tool allows researchers to precisely specify the attributes of projects of interest and obtain their samples, including source code, history and metadata. CodeDJ is fully deterministic and supports historical reproducibility, i.e., each query can specify a time and the results will be returned as if the query has been asked at that time. As a first example of the usefulness of our tool and to motivate its use we have used CodeDJ to investigate the effects of project selection on the original study discussed in the previous section. [A.1]

In our second contribution in section 3.2, we have identified reliance on project popularity, instead of intrinsic project attributes, as selectors for developed projects as a problem. The high duplication rates in projects related to popular themes at the time, and projects like $PHPDesignPatterns$ suggested the correlation between popular and developed projects is not without issues. But although we have argued earlier that precise project selection is a key step in dealing with garbage present in large software repositories, we could not confirm its effects on the claim of the original study as GitHub does not support random selection, complex queries, or historical records as detailed in chapter 2 of this thesis.

CodeDJ is our answer to this crucial lack of functionality. Its goal is to allow researchers to formulate complex queries that evaluate attributes of software hosted on GitHub and return data about matching projects. We have selected GitHub as our sole information provider, but the tool is designed to support additional project sources in the future.

Being reproducible effectively means that CodeDJ must keep its own mirror of projects, their metadata and contents, as GitHub itself gives no guarantees about immutability or indeed availability of previously accessed data. The enormous sizes of data involved, both in absolute and relative terms, mandated that instead of simply repurposing existing data storage systems, such as relational databases used by GHTorrent, we opted for a bespoke solution to minimize overhead. The design of CodeDJ flows from four high-level principles:

- *Consistent, eventually*: The sheer size and churn in GitHub means that obtaining a snapshot of the whole data source is not practical. But, it is often the case that a slightly out-of-date view is sufficient for most investigations. We choose to refresh entire projects atomically at irregular intervals. Thus, any individual project is consistent, but for any group of projects, the lower bound on their refresh times is the last consistent time point.

- *Code-centric, language agnostic*: We aim to support queries on project metadata and file contents written in any programming language. To reduce space requirements, the only source artifacts we store is code, deduplication is used to remove redundancy, and metadata is trimmed where possible without loss of information.

- *Flexible query interface*: CodeDJ is split between two components - Parasite and Djanco. Parasite is a continuously evolving datastore that tracks changes in GitHub projects and adds them to its database. Parasite provides a simple querying API (index for random access and sequential iterator). This simple but complete querying mechanism can be extended by clients, one of which is Djanco. Djanco builds a domain specific language for writing complex queries over the Parasite datastore that resemble modern data manipulation pipelines.

- *Reproducible by design*: The importance of reproducibility cannot be overstated. CodeDJ is designed so it is possible to run any query with the information the datastore had at an arbitrary point in the past. For this purpose, the datastore is time-indexed, strictly append-only.

Let us illustrate how CodeDJ works on the same task we subjected the existing software repositories to in the previous chapter. Recall that we were after a random sample of 10K projects, their commits and the source code changes. We require projects with at least 50 commits in either Haskell or C++ languages, active issues use, at least two developers, and more than 6 months of development.

As CodeDJ is only a mirror of GitHub, we must first ensure that we have enough such projects. If that were not the case, we would have to instruct Parasite to look for C++ and Haskell projects and add them to the datastore. In the paper we bootstrapped CodeDJ with project information from GHTorrent, but have since moved towards our own scanner that uses a continuously updated list of all public repositories and randomly scans them for project metadata. The random scan ensures that while the process continues, the data we have always make an unbiased random sample of GitHub. Projects in the languages of our choice will then be cloned and analyzed for file contents and other information Parasite keeps track of (history, commit messages, project metadata, etc.).

All this information is deduplicated and stored in append-only storage files. Our storage files have immutable schema, support variable length records and are designed with minimal disk overhead to ensure we scale to hundreds of millions of projects. When newer versions of the scrapper download new kinds of information, new storage files are created instead of changing the schema of existing ones. The scrapper alternates between adding new projects and revisiting existing projects for any changes to be added. The append-only nature of the storage files ensure that we never lose information, i.e., if a project gets deleted, becomes private, or its history is altered, we record the change as a new latest state without removing the old one.

The storage files, required internally for deduplication, or for any serious queries, are extremely inefficient for random access. They only support forward search, that together with the append only nature, makes it impossible to reason about validity of any entry until the whole file is scanned. To mitigate, Parasite uses index files heavily. Similarly to databases, but with much lower overhead, the index files store offsets to the storage files. Different index files for direct, indirect, or linked access, where new entries link not only to the information in the storage file, but also to the index of the previous entry, are supported.

Going back to our example, after we gathered the projects we wish to filter and sample, one important step remains before we can query the datastore. To ensure historical reproducibility, CodeDJ requires each query to specify a *savepoint* of the datastore on which the query will be executed. This feature is semantically similar to existing tools such as GHTorrent and Orion who provide their database snapshots for download. Instead of a full dump, which would be prohibitively expensive, we exploit the append-only nature of the storage files. A savepoint for CodeDJ is nothing more than a list of storage files forming the dataset and their current sizes. Savepoints are complicated by the need for our datastore to remain always consistent, e.g., when commit A with parent commit B and a change to file C is in the dataset, so must be commit B and the new contents of file C. We ensure this by enforcing an order on analyzed new commits.

Specifying the savepoint gives us a queryable constant view into the full datastore, while the store itself can continue to be updated. Parasite provides a Rust library that allows iterator and index-based access to all of the storage files that can be directly used to run queries. As an implementation detail, the CodeDJ datastore is partitioned into substores based on project languages to simplify the default use case. Listing 3.3 shows an example program in abridged Rust that gives a random subset of 10K Haskell projects fulfilling our needs.

Rust might not be the language best suited for further analysis, nor is the low-level API using iterators and indices particularly efficient at expressing more complex queries. We have designed CodeDJ and particularly the Parasite datastore to work with multiple clients with varying capabilities. One such client, which did not make it into our paper is mistletoe, a command-line utility that can list projects and their attributes and export the projects and their file contents on disk, similarly to the GHTorrent vault. It can be used in combination with other tools researchers are already familiar with, such as the R programming language to provide efficient access to the datastore. Another Parasite client, Djanco introduces a domain specific language (DSL) that is suited for analysis and filtering of projects based on their intrinsic values. Figure 3.4 shows the same query expressed in Djanco. The DSL provides several useful abstractions over the Parasite API such as synthesized attributes like the number of commits and developers, project age, or more explicit sampling functions that also provide deduplication out of the box.[1]

For a more detailed description of CodeDJ functionality and additional information, we refer the reader to our paper.

Pitfalls of Project Selection. To showcase the value of CodeDJ, our paper demonstrates how it can be used not only to perform precise project selection, but to assist in its validation as well. We have reused the original study from our previous paper in section 3.2 as we were already familiar with it, and CodeDJ allowed us to perform the last missing piece of the reproduction, the validation of its project selection criteria.

[1]The main author of Djanco is Konrad Siek, not the author of this thesis. We simply use to it demonstrate the capabilities of CodeDJ architecture.

```
// select the datastore, specified by the root folder of its storage files
let cdj = DatastoreView::from("path/to/datastore").substore();
// select the savepoint and substore
let savepoint = cdj.savepoints().latest().unwrap();
let substore = Substore::Haskell;
// list all projects
let projects = ds
    .projects(substore, & savepoint)
    .filter_map(|project_id| {
        // for each project obtain info and heads (latest commits in its branches)
        let pinfo = cdj.project_updates(substore, & savepoint).get(project_id).
            unwrap();
        // filter projects with no issues
        if pinfo.issues < 1 {
            return None;
        }
        let heads = cdj.project_heads(substore, & savepoint).get(project_id).unwrap
            ();
        let commits = ProjectCommitsIterator::new(& heads, cdj.commits_info(substore
            )).collect::<Vec<_>>();
        // filter projects with too few commits
        if commits.len() < 50 {
            None
        // filter projects with less than 6 months lifetime
        } else if (commits.iter().map(|c| c.author_time).max() - commits.iter().map
            (|c| c.author_time).min()) < 31 * 6 {
            None
        // filter projects with less than 2 users
        } else if (commits.iter().map(|c| x.author).distinct().len() == 1) {
            None
        } else {
            Some((pinfo, commits))
        }
    })
    // sample 10K projects randomly and store them to a vector
    .choose_multiple(10000)
    .collect::<Vec<_>>();
// do what needs to be done with the returned projects
// ...
```

Figure 3.3: Getting a random sample of 10K developed projects written in Haskell using the low level Parasite API

First, to determine how well the reported results generalize, we performed a simple experiment: Using CodeDJ we were able to quickly get 1000 sets of 50 random projects per language (similar to the original study). For each of the thousand datasets we calculated the coefficients as obtained by the paper. The distribution of those compared to the single coefficient per language given by the paper is shown in figure 3.5. The spread of each distribution is a measure of the sensitivity of the analysis to its inputs. One could argue that picking close to the median of the distribution is more likely to give a representative and robust answer. Selecting outliers, such as the C#, Perl, or most strikingly TypeScript, should definitely raise some eyebrows, ideally leading to further investigation.

The observed sensitivity to the input selection can usually be mitigated by increasing the

```
// open the datastore
let projects = Djanco::from(PATH)
    .projects()
    // use only haskell projects
    .filter_by(Equals(project::Language, "Haskell")
    // select those with at least 50 commits
    .filter_by(AtLeast(Count(project::Commits), 50))
    // at least two users
    .filter_by(AtLeast(Count(project::Users), 2))
    // at least 6 months
    .filter_by(AtLeast(Count(project::Age), 31 * 6))
    // uses issues
    .filter_by(AtLeast(Count(project::Issues), 1))
    // sample 10K projects and ensure that each added project will have at least 90% unique
         commits wrt the rest of the subset
    .sample(Distinct(Random(10000, Seed(42)), MinRatio(project::Commits, 0.9)));
// process the projects
// ...
```

Figure 3.4: Getting a random sample of 10K developed projects written in Haskell using the high level Djanco API

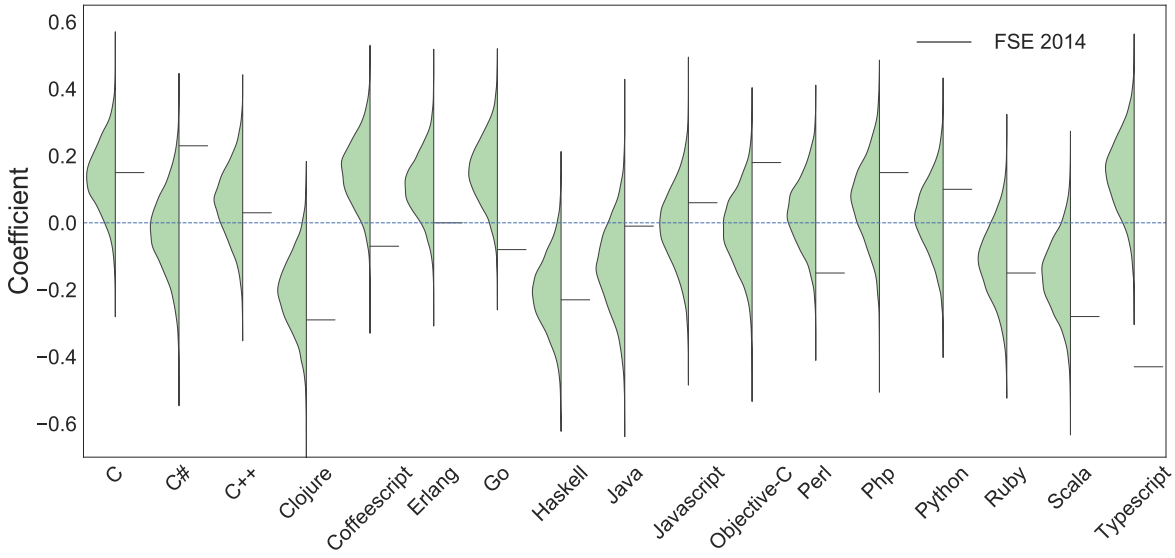


Figure 3.5: Distribution of coefficients calculated from 1000 random subsets compared to those provided in the original study

sample sizes, i.e., analyzing more than 50 projects per language the original study did. But this only works if the actual selection makes sense, e.g., if the most popular projects form a representative subset of developed projects. We tried to verify this assumption by an experiment that compared the predictions made by the most popular projects to other, equally plausible selections. Our subsets were obtained from CodeDJ by sampling the population if interest defined explicitly by project attributes expected to correlate with project development, such as:

- *Touched Files:* compute number of files changed by commits, pick projects that changed the most files. *Rationale:* indicative of projects where commits represent larger units of work.
- *Experienced Author:* experienced developers are those on GitHub for at least two years; pick a sample of projects with at least one experienced contributor. *Rationale:* less likely to be throw-away projects.
- *50% Experienced:* projects with two or more developers, half of which experienced. *Rationale:* focus on larger teams.
- *Message Size:* Compute size in bytes of commit messages; pick projects with the largest size. *Rationale:* empty or trivial commit messages indicate uninteresting projects.
- *Number of Commits:* Compute the number of commits; pick projects with the most commits. *Rationale:* larger projects are more mature.
- *Issues:* Pick projects with the most issues. *Rationale:* issues indicate a more structured development process.

Figure 3.6 shows, for each language, the value of the coefficients (recall that higher means more bugs). Coefficients that are not statistically significant are shown in faded colors. If the input set did not matter for the model used for the analysis, one could expect the different queries to give roughly the same coefficients with the same significance. That is not the case. The touched files query is highly predictive, 14 of the languages are significant, but the coefficients are frequently opposite from those of other queries. This is striking as it goes against expectations. The stars query is the least informative. It only gives 7 statistically significant coefficients with remarkably low values, with multiple disagreements with the majority as well (e.g., CoffeeScript, Erlang, Java).

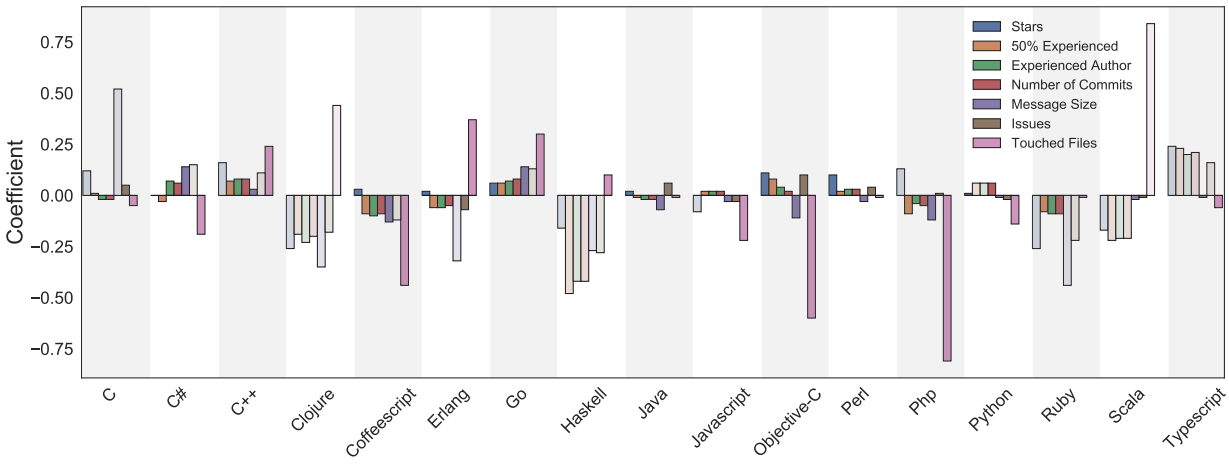


Figure 3.6: Domain knowledge

In our paper, we stress the importance of understanding the selection criteria and its impact, as statistical significance should not be confused with validity. To help, CodeDJ can easily provide distributions of various measures in the data. Figure 3.7 shows the distribution

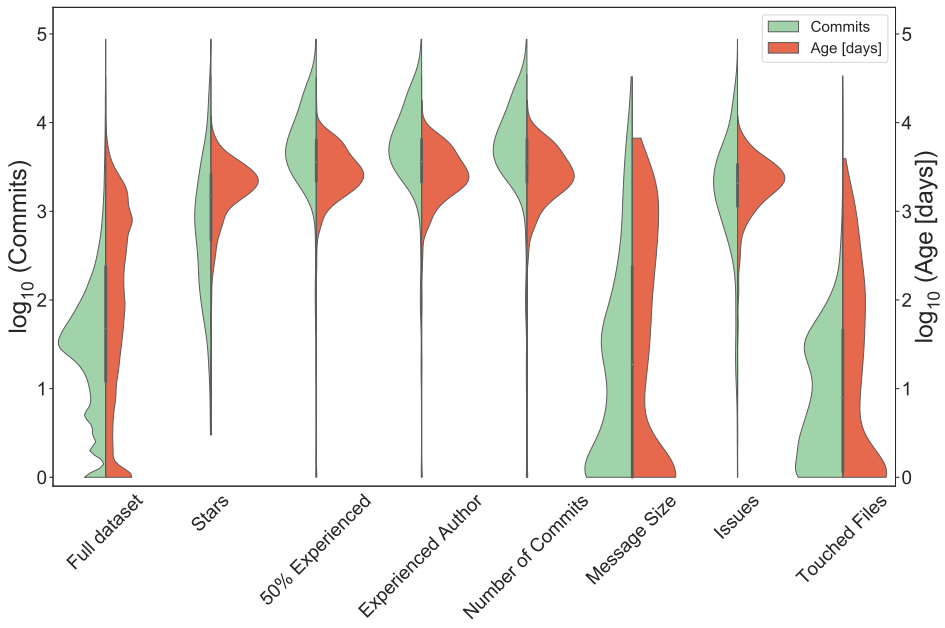


Figure 3.7: Distribution of project age and number of commits over various project selections.

of project sizes (left) and project age (right) for the entire dataset and for the various queries. Looking at these distributions makes it clear that they return quite different projects. The experienced author and number of commits are remarkably similar and return projects that meet our expectations. The issues distribution is similar, which should raise red flags given that it frequently disagrees. The stars query returns many smaller projects. Finally, message sizes and touched files show distributions opposite to those expected. They favor degenerate young projects with few commits that are either verbose, or disproportionately large (touching over 100K files). This is reflected in the input sizes, ranging from 8M rows for the experienced author query to mere 79K rows of the touched files query. It is likely that these queries are "wrong" in the sense they do not return the population of interest, exacerbated by their reliance on sorting, as opposed to random sampling.

In summary, our paper has provided the research community with CodeDJ, a tool for precise, scalable and reproducible project selection, promised by this thesis. Our tool is essential for the data gathering steps of big code analyses and we have shown some of its usefulness and motivated its adoption by performing the analysis of project selection based on project popularity used in an influential paper. Our analysis confirmed problems with such extrinsic selection criteria.

3.4 Designing Reproducible Big Code Experiments

Our last contribution, a paper entitled *The Fault in Our Stars: How to Design Reproducible, Large-scale Code Analysis Experiments*, submitted to ECOOP 2023 advocates in favor of standardized experimental design methodology for big code analysis. In particular, we analyze the shortcomings of using popularity related convenience sampling methods and provide a simple methodology for experiment design based on explicit definition of projects of interest

enabled by CodeDJ. [A.4]

We have analyzed two years' worth of Mining Software Repositories (MSR) conference and found out that a surprisingly high number of published studies - 48%, used stars as their sole selection criteria. Why do GitHub stars play such a central role in our experimental methodology? Most of the papers we have reviewed were not interested in popular projects per se, but used popularity as a proxy to the already familiar, albeit vague notion of developed projects. As the previous chapter showed in detail, large software repositories are not built for data mining. GitHub does not provide an easy-to-use index of hosted projects that can be searched. Selecting the most popular projects is an example of convenience sampling. Without specialized tools, such as CodeDJ, it is next to impossible to select the real projects of interest. Selecting the popular ones is the closest criteria we can practically get.

The reader of this thesis should already raise their eyebrows, since at this point, we have demonstrated the implied correlation between popularity and quality to be dubious at best. To convince those having only the paper to read, it analyzes the top starred projects and how well they remove common biases, such as duplication, how much they preserve the composition of the projects of interest and how efficient they are at filtering garbage. We then follow by arguing for a big code experiments methodology that improves generalizability and reproducibility of our results.

Stars v. All. For most big code analyses, one wants to find projects of interest while avoiding the duplicates that litter most language ecosystems and weeding out obviously uninteresting projects - the garbage. But do stars really correlate with *some* notion of interest, filter out uninteresting projects and remove duplicates? Unfortunately, the answer to all three questions is *no*.

Most popular projects certainly do not remove duplication in significant numbers. While analyzing the entire GitHub for duplicates in C++, Java, Python and JavaScript, we have also checked top 1K most popular projects in each language. For C++ and JavaScript, 41% and 44%, respectively, of the most popular projects were duplicates. Python was only slightly better with 28%, but even in Java, the most popular projects contained 9% duplication (note that much lower duplication rate in our case study was already affecting the results).

To judge how well stars perform in selecting the projects of interest we turn to project attributes corresponding to developed projects, such as project size (in terms of commits, lines of code, etc.), age, or the number of contributing developers. The exact values of those attributes that make a project interesting enough are hard to quantify generally, so we choose the opposite: any project that has fewer than 100 lines of code, fewer than 10 commits, or has been alive for less than a week is obviously uninteresting. This rather low bar already removes copious amounts of the dataset, for Java only 29% of projects remain. Figure 3.8 contrasts the distribution of some project attributes related to its development on the whole dataset (gray), the dataset with uninteresting projects removed (black), and 1K most popular projects (deliberately chosen as what is available from GitHub). Ideally, one would expect the distributions of the interesting and most popular projects to be skewed slightly towards older, larger, more committed to projects, with generally similar shape to avoid introduction of bias. The interesting projects follow this expectation, but the distribution patterns in the top starred projects are rather different. The contrast is particularly stark for the project age. It takes a while for a project to become popular, and since stars are rarely removed, once a project becomes popular, it stays popular for many years after its development cedes, rendering top starred projects, on top of all their problems, particularly

unsuitable for analyzing the latest trends.

We might be willing to forgive the most popular projects their high duplication, or the heavy skew towards large, old and community developed projects. One can always perform deduplication separately, albeit at a cost of severely reduced dataset, and we might even attempt to convince ourselves this is what we want, despite the rather different makeup of the entire repository. But at the very least stars should be effective in removing garbage. Yet not even this is the case. By manually examining the most popular projects for outliers in the attributes associated with project development, we have found that at least 17% of the thousand most popular projects in Java and Python should in fact be considered uninteresting as they are either too small toy projects, large archives without significant development, or not even software.

Despite their demonstrated shortcomings, the expected correlation between popularity and quality (for any meaning of it) is deeply embedded in much of our thinking. Time and time again have reviewers of the papers in this thesis raised the question of whether popularity is really as bad as we claim. Perhaps, for some particular notion of quality, stars make perfect sense. Maybe. But even then, selecting the most popular projects is the wrong answer that

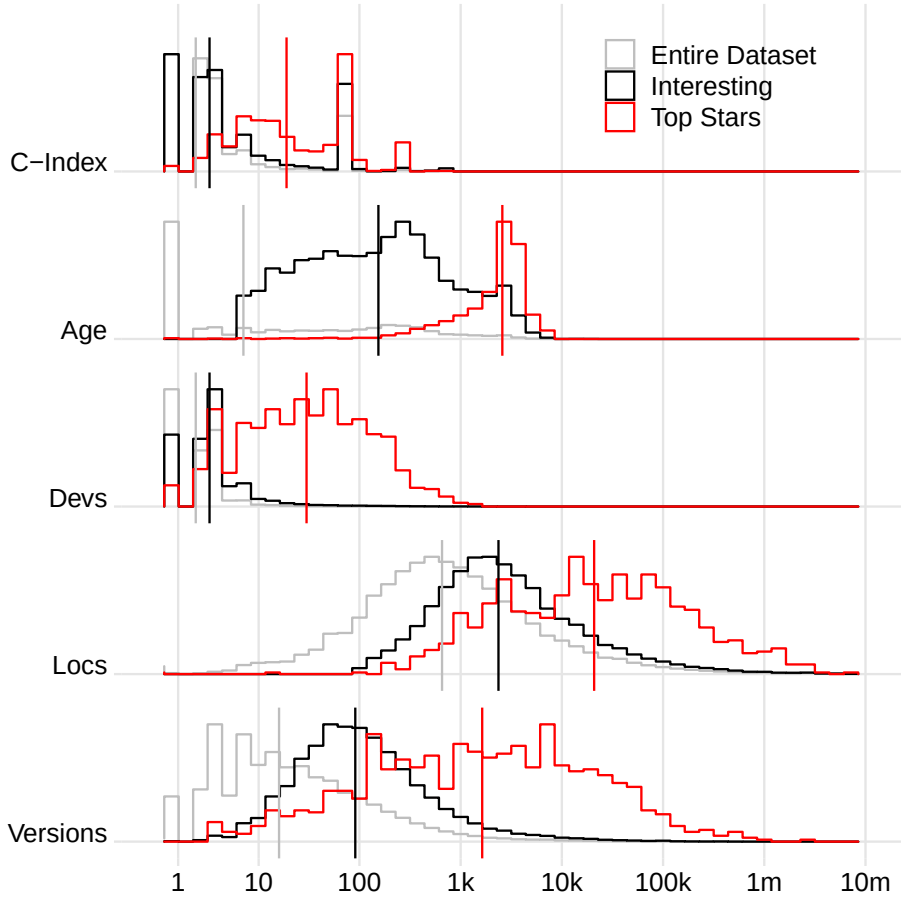


Figure 3.8: Comparing developed and starred projects to the entire dataset in various software engineering metrics. See [A.4] for their detailed descriptions.

reveals perhaps the biggest fault in our stars: one of the benefits of random sampling is that each sample is different. Other studies analyzing different subsets will eventually point out any discrepancies. But if everyone is using the same subset of the *most* popular projects, we will never know, as new studies will keep analyzing the same data over.

Methodology. The task of filtering projects from large software repositories we designed CodeDJ for is only one, albeit crucial, step towards a more robust and generalizable data gathering for big code analyses. We thus conclude our contributions with a proposal of methodology for designing reproducible experiments with the explicit goal of improving the generalizability of the results. The methodology is in line with evolving community standards [10], but specific to large-scale code analysis. Our approach builds on our ability to precisely and reproducibly select projects based on their measurable attributes and takes the form of the following protocol:

- *Population Hypothesis*: A brief description of the population of interest, what the research should generalize to, which may be a narrow slice such as "programs written by students learning JavaScript as their first language" or a broader one such as "commercial code".
- *Frame Oracle*: A procedure for deciding if a project belongs to the population. Ideally, an algorithm efficiently computed over intrinsic attributes of a project. An oracle could, e.g., return GitHub projects with one JavaScript file which were created by a user with no previous commits.
- *Sampling Strategy*: A strategy for selecting a subset of the values of the population. Ideally, specified algorithmically. An example is random sampling without replacement from a known seed.
- *Validity*: An argument about the oracle's and sampling strategy's validity as means to obtain representative samples from the population. A discussion of attempts to validate result quality, such as manual inspection of a sample to check if JavaScript code was actually written by beginners.
- *Reproduction Artifacts*: The artifact should allow to reproduce exactly the reported results as well as to change either the input or the experiment.

We have picked four research papers and shown how our methodology can improve their results by either confirming or eliminating some of the threats to their validity and improving the ability of our research community to review and trust our results.

3.5 Summary

We identify precise project selection, data filtering, and reproducibility as the key challenges faced by researchers analyzing software repositories that are too large and too noisy to be analyzed whole. Through an analysis of GitHub projects in four major programming languages, we give evidence of the widespread bias and noise present in software repositories, and through reproductions we show that mistakes made in those areas, and lack of tooling to support the tasks has direct impact on the research claims.

We also introduce CodeDJ, a tool for precise, scalable and reproducible project selection. Building on the abilities of CodeDJ, we propose a methodology to replace the widespread, but grossly inadequate, practice of convenience sampling in the form of project popularity with carefully designed reproducible experiments to improve the robustness and generalizability of our results.

Chapter 4

Relevant Papers

This chapter presents the four research papers that form the main results of the thesis. Each paper is accompanied by a description of author's contributions and a list of citations. All our papers come with functional and reusable artifacts that were submitted to the corresponding artifact evaluation committees.

4.1 Paper 1 - DejaVu: A Map of Code Duplicates on GitHub

Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek.

In: Proc. ACM Program. Lang. 1, OOPSLA, Article 84 (October 2017), 28 pages. https://doi.org/10.1145/3133908

4.1.1 Author's Contributions

I was responsible for almost all of the work on the JavaScript pipeline. Specifically the data acquisition, tokenization and the analysis of duplicates. Since the JavaScript dataset was the largest, I had to rewrite previously used programs from scratch to improve their scalability. Those programs were then used for the other three languages as well. I was responsible for almost all data visualization and interpretation in the paper, notably the heatmaps and their analysis. I also implemented a library in the R programming language that automatically generated all data results used in the paper to ensure reproducibility and prepared the artifact, which was awarded the *Distinguished Artifact Award* at OOPSLA. The library was re-used in our subsequent papers.

Jakub Zitny developed the first version of a JavaScript tokenizer and identified the NPM modules as a source of duplication.

Cristina Lopes, Pedro Martins, Vaibhav Saini and Di Yang, all from the University of California, Irvine, were responsible for gathering the Python, Java and C/C++ datasets, running the SourcererCC analysis on them, and for constructing the file level duplication graph in the paper. Hitesh Sajnani is the author of the SourcererCC tool [14] that was used in the paper.

I presented the paper at OOPSLA 2017 and performed the additional dataset and heatmap analysis that was part of the presentation, but did not make it in the paper.

4.1.2 Citations

1. Rabin, M., Hussain, A., Alipour, M. & Hellendoorn, V. Memorization and generalization in neural code intelligence models. *Information And Software Technology.* **153** (2023)

2. Kang, W., Son, B. & Heo, K. TRACER: Signature-based Static Analysis for Detecting Recurring Vulnerabilities. *Proceedings Of The ACM Conference On Computer And Communications Security.* pp. 1695-1708 (2022)

3. Dyer, R. & Chauhan, J. An exploratory study on the predominant programming paradigms in Python code. *ESEC/FSE 2022 - Proceedings Of The 30th ACM Joint Meeting European Software Engineering Conference And Symposium On The Foundations Of Software Engineering.* pp. 684-695 (2022)

4. Yu, Y., Huang, Z., Shen, G., Li, W. & Shao, Y. ASTENS-BWA: Searching partial syntactic similar regions between source code fragments via AST-based encoded sequence alignment. *Science Of Computer Programming.* **222** (2022)

5. Dann, A., Plate, H., Hermann, B., Ponta, S. & Bodden, E. Identifying Challenges for OSS Vulnerability Scanners-A Study & Test Suite. *IEEE Transactions On Software Engineering.* **48**, 3613-3625 (2022)

6. Kang, H. & Lo, D. Active Learning of Discriminative Subgraph Patterns for API Misuse Detection. *IEEE Transactions On Software Engineering.* **48**, 2761-2783 (2022)

7. Hannousse, A., Nait-Hamoud, M. & Yahiouche, S. A deep learner model for multi-language webshell detection. *International Journal Of Information Security.* (2022)

8. Jesse, K. & Devanbu, P. ManyTypes4TypeScript: A Comprehensive TypeScript Dataset for Sequence-Based Type Inference. *Proceedings - 2022 Mining Software Repositories Conference, MSR 2022.* pp. 294-298 (2022)

9. Ciniselli, M., Pascarella, L. & Bavota, G. To What Extent do Deep Learning-based Code Recommenders Generate Predictions by Cloning Code from the Training Set?. *Proceedings - 2022 Mining Software Repositories Conference, MSR 2022.* pp. 167-178 (2022)

10. Mir, A., Latoskinas, E., Proksch, S. & Gousios, G. Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python. *Proceedings - International Conference On Software Engineering.* **2022-May** pp. 2241-2252 (2022)

11. Huang, Y., Xu, F., Zhou, H., Chen, X., Zhou, X. & Wang, T. Towards Exploring the Code Reuse from Stack Overflow during Software Development. *IEEE International Conference On Program Comprehension.* **2022-March** pp. 548-559 (2022)

12. Misu, M. & Satter, A. An Exploratory Study of Analyzing JavaScript Online Code Clones. *IEEE International Conference On Program Comprehension.* **2022-March** pp. 94-98 (2022)

13. Liang, J., Zimmermann, T. & Ford, D. Towards Mining OSS Skills from GitHub Activity. *Proceedings - International Conference On Software Engineering.* pp. 106-110 (2022)

14. Ahmed, T. & Devanbu, P. Multilingual training for Software Engineering. *Proceedings - International Conference On Software Engineering.* **2022-May** pp. 1443-1455 (2022)

15. Coupette, C., Hartung, D., Beckedorf, J., Böther, M. & Katz, D. Law Smells: Defining and Detecting Problematic Patterns in Legal Drafting. *Artificial Intelligence And Law.* (2022)

16. Papathomas, E., Diamantopoulos, T. & Symeonidis, A. Semantic Code Search in Software Repositories using Neural Machine Translation. *Lecture Notes In Computer Science (including Subseries Lecture Notes In Artificial Intelligence And Lecture Notes In Bioinformatics).* **13241 LNCS** pp. 225-244 (2022)

17. Yang, L., Ren, Y., Guan, J., Li, B., Ma, J., Han, P. & Tan, Y. FastDCF: A Partial Index Based Distributed and Scalable Near-Miss Code Clone Detection Approach for Very Large Code Repositories. *Lecture Notes In Computer Science (including Subseries Lecture Notes In Artificial Intelligence And Lecture Notes In Bioinformatics).* **13148 LNCS** pp. 210-222 (2022)

18. Crawford, R. & Sloss, A. Is Expressiveness the Future of Software?. *Computer.* **55**, 43-52 (2022)

19. Sonnekalb, T., Heinze, T. & Mäder, P. Deep security analysis of program code: A systematic literature review. *Empirical Software Engineering.* **27** (2022)

20. Nguyen, P., Di Rocco, J., Iovino, L., Di Ruscio, D. & Pierantonio, A. Evaluation of a machine learning classifier for metamodels. *Software And Systems Modeling.* **20**, 1797-1821 (2021)

21. Amit, I. & Feitelson, D. Corrective commit probability: a measure of the effort invested in bug fixing. *Software Quality Journal.* **29**, 817-861 (2021)

22. Mathew, G. & Stolee, K. Cross-language code search using static and dynamic analyses. *ESEC/FSE 2021 - Proceedings Of The 29th ACM Joint Meeting European Software Engineering Conference And Symposium On The Foundations Of Software Engineering.* pp. 205-217 (2021)

23. Bogomolov, E., Kovalenko, V., Rebryk, Y., Bacchelli, A. & Bryksin, T. Authorship attribution of source code: A language-agnostic approach and applicability in software engineering. *ESEC/FSE 2021 - Proceedings Of The 29th ACM Joint Meeting European Software Engineering Conference And Symposium On The Foundations Of Software Engineering.* pp. 932-944 (2021)

24. Liang, H. & Ai, L. AST-path Based Compare-Aggregate Network for Code Clone Detection. *Proceedings Of The International Joint Conference On Neural Networks.* **2021-July** (2021)

25. Bogart, C., Kästner, C., Herbsleb, J. & Thung, F. When and How to Make Breaking Changes: Policies and Practices in 18 Open Source Software Ecosystems. *ACM Transactions On Software Engineering And Methodology.* **30** (2021)

26. He, J., Lee, C., Raychev, V. & Vechev, M. Learning to find naming issues with big code and small supervision. *Proceedings Of The ACM SIGPLAN Conference On Programming Language Design And Implementation (PLDI).* pp. 296-311 (2021)

27. Hata, H., Kula, R., Ishio, T. & Treude, C. Same file, different changes: The potential of meta-maintenance on GitHub. *Proceedings - International Conference On Software Engineering.* pp. 773-784 (2021)

28. Mir, A., Latoskinas, E. & Gousios, G. ManyTypes4Py: A benchmark python dataset for machine learning-based type inference. *Proceedings - 2021 IEEE/ACM 18th International Conference On Mining Software Repositories, MSR 2021.* pp. 585-589 (2021)

29. Svyatkovskiy, A., Lee, S., Hadjitofi, A., Riechert, M., Franco, J. & Allamanis, M. Fast and memory-efficient neural code completion. *Proceedings - 2021 IEEE/ACM 18th International Conference On Mining Software Repositories, MSR 2021.* pp. 329-340 (2021)

30. Woo, S., Park, S., Kim, S., Lee, H. & Oh, H. CENTRIS: A precise and scalable approach for identifying modified open-source software reuse. *Proceedings - International Conference On Software Engineering.* pp. 860-872 (2021)

31. Tang, Y., Khatchadourian, R., Bagherzadeh, M., Singh, R., Stewart, A. & Raja, A. An empirical study of refactorings and technical debt in machine learning systems. *Proceedings - International Conference On Software Engineering.* pp. 238-250 (2021)

32. Golubev, Y., Poletansky, V., Povarov, N. & Bryksin, T. Multi-threshold token-based code clone detection. *Proceedings - 2021 IEEE International Conference On Software Analysis, Evolution And Reengineering, SANER 2021.* pp. 496-500 (2021)

33. Chen, X., Abdalkareem, R., Mujahid, S., Shihab, E. & Xia, X. Helping or not helping? Why and how trivial packages impact the npm ecosystem. *Empirical Software Engineering.* **26** (2021)

34. Nguyen, P., Di Ruscio, D., Pierantonio, A., Di Rocco, J. & Iovino, L. Convolutional neural networks for enhanced classification mechanisms of metamodels. *Journal Of Systems And Software.* **172** (2021)

35. Seker, A., Diri, B., Arslan, H. & Amasyali, M. Open Source Software Development Challenges: A Systematic Literature Review on GitHub. *Research Anthology On Agile Software, Software Development, And Testing.* **4** pp. 2134-2164 (2021)

36. Lachaux, M., Roziere, B., Szafraniec, M. & Lample, G. DOBF: A Deobfuscation Pre-Training Objective for Programming Languages. *Advances In Neural Information Processing Systems.* **18** pp. 14967-14979 (2021)

37. Seker, A., Diri, B., Arslan, H. & Amasyalı, M. Open Source Software Development Challenges: A Systematic Literature Review on GitHub. *Research Anthology On Usage And Development Of Open Source Software.* **1** pp. 33-62 (2021)

38. Källén, M., Sigvardsson, U. & Wrigstad, T. Jupyter Notebooks on GitHub: Characteristics and Code Clones. *Art, Science, And Engineering Of Programming.* **5** (2021)

39. Villmow, J., Depoix, J. & Ulges, A. CONTEST: A Unit Test Completion Benchmark featuring Context. *NLP4Prog 2021 - 1st Workshop On Natural Language Processing For Programming, Proceedings Of The Workshop.* pp. 17-25 (2021)

40. Golubev, Y. & Bryksin, T. On the Nature of Code Cloning in Open-Source Java Projects. *Proceedings - 2021 IEEE 15th International Workshop On Software Clones, IWSC 2021.* pp. 22-28 (2021)

41. Farmahinifarahani, F., Lu, Y., Saini, V., Baldi, P. & Lopes, C. D-REX: Static Detection of Relevant Runtime Exceptions with Location Aware Transformer. *Proceedings - IEEE 21st International Working Conference On Source Code Analysis And Manipulation, SCAM 2021.* pp. 198-208 (2021)

42. David, Y., Alon, U. & Yahav, E. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings Of The ACM On Programming Languages.* **4** (2020)

43. Devore-Mcdonald, B. & Berger, E. Mossad: Defeating software plagiarism detection. *Proceedings Of The ACM On Programming Languages.* **4** (2020)

44. Brody, S., Alon, U. & Yahav, E. A structural model for contextual code changes. *Proceedings Of The ACM On Programming Languages.* **4** (2020)

45. Kondo, M., Oliva, G., Jiang, Z., Hassan, A. & Mizuno, O. Code cloning in smart contracts: a case study on verified contracts from the Ethereum blockchain platform. *Empirical Software Engineering.* **25**, 4617-4675 (2020)

46. Seker, A., Diri, B., Arslan, H. & Amasyalı, M. Open source software development challenges: A systematic literature review on GitHub. *International Journal Of Open Source Software And Processes.* **11**, 1-26 (2020)

47. Spinellis, D., Kotti, Z. & Mockus, A. A Dataset for GitHub Repository Deduplication. *Proceedings - 2020 IEEE/ACM 17th International Conference On Mining Software Repositories, MSR 2020.* pp. 523-527 (2020)

48. Golubev, Y., Eliseeva, M., Povarov, N. & Bryksin, T. A Study of Potential Code Borrowing and License Violations in Java Projects on GitHub. *Proceedings - 2020 IEEE/ACM 17th International Conference On Mining Software Repositories, MSR 2020.* pp. 54-64 (2020)

49. Allamanis, M., Barr, E., Ducousso, S. & Gao, Z. Typilus: Neural type hints. *Proceedings Of The ACM SIGPLAN Conference On Programming Language Design And Implementation (PLDI).* pp. 91-105 (2020)

50. Tang, W., Luo, P., Fu, J. & Zhang, D. LibDX: A Cross-Platform and Accurate System to Detect Third-Party Libraries in Binary Code. *SANER 2020 - Proceedings Of The 2020 IEEE 27th International Conference On Software Analysis, Evolution, And Reengineering.* pp. 104-115 (2020)

51. Couto, M., Saraiva, J. & Fernandes, J. Energy Refactorings for Android in the Large and in the Wild. *SANER 2020 - Proceedings Of The 2020 IEEE 27th International Conference On Software Analysis, Evolution, And Reengineering.* pp. 217-228 (2020)

52. Li, G., Wu, Y., Roy, C., Sun, J., Peng, X., Zhan, N., Hu, B. & Ma, J. SAGA: Efficient and Large-Scale Detection of Near-Miss Clones with GPU Acceleration. *SANER 2020 - Proceedings Of The 2020 IEEE 27th International Conference On Software Analysis, Evolution, And Reengineering.* pp. 272-283 (2020)

53. Allamanis, M. The adverse effects of code duplication in machine learning models of code. *Onward! 2019 - Proceedings Of The 2019 ACM SIGPLAN International Symposium On New Ideas, New Paradigms, And Reflections On Programming And Software, Co-located With SPLASH 2019.* pp. 143-153 (2019)

54. Křikava, F., Miller, H. & Vitek, J. Scala implicits are everywhere a large-scale study of the use of scala implicits in the wild. *Proceedings Of The ACM On Programming Languages.*

55. Luan, S., Yang, D., Barnaby, C., Sen, K. & Chandra, S. Aroma: Code recommendation via structural code search. *Proceedings Of The ACM On Programming Languages.* **3** (2019)

56. Mastrangelo, L., Hauswirth, M. & Nystrom, N. Casting about in the dark an empirical study of cast operations in Java programs. *Proceedings Of The ACM On Programming Languages*

57. Kessel, M. & Atkinson, C. Automatically curated data sets. *Proceedings - 19th IEEE International Working Conference On Source Code Analysis And Manipulation, SCAM 2019*. pp. 56-61 (2019)

58. Ragkhitwetsagul, C. & Krinke, J. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering*. **24**, 2236-2284 (2019)

59. Rigger, M., Marr, S., Adams, B. & Mössenböck, H. Understanding GCC builtins to develop better tools. *ESEC/FSE 2019 - Proceedings Of The 2019 27th ACM Joint Meeting European Software Engineering Conference And Symposium On The Foundations Of Software Engineering*. pp. 74-85 (2019)

60. Davis, J., Michael, L., Coghlan, C., Servant, F. & Lee, D. Why arent regular expressions a lingua franca? An empirical study on the re-use and portability of regular expressions. *ESEC/FSE 2019 - Proceedings Of The 2019 27th ACM Joint Meeting European Software Engineering Conference And Symposium On The Foundations Of Software Engineering*. pp. 443-454 (2019)

61. Baltes, S. & Diehl, S. Usage and attribution of Stack Overflow code snippets in GitHub projects. *Empirical Software Engineering*. **24**, 1259-1295 (2019)

62. Theeten, B., Vandeputte, F. & Van Cutsem, T. Import2vec: Learning embeddings for software libraries. *IEEE International Working Conference On Mining Software Repositories*. **2019-May** pp. 18-28 (2019)

63. Rua, R., Couto, M. & Saraiva, J. GreenSource: A large-scale collection of android code, tests and energy metrics. *IEEE International Working Conference On Mining Software Repositories*. **2019-May** pp. 176-180 (2019)

64. Perez, D. & Chiba, S. Cross-language clone detection by learning over abstract syntax trees. *IEEE International Working Conference On Mining Software Repositories*. **2019-May** pp. 518-528 (2019)

65. Zhang, T., Yang, D., Lopes, C. & Kim, M. Analyzing and Supporting Adaptation of Online Code Examples. *Proceedings - International Conference On Software Engineering*. **2019-May** pp. 316-327 (2019)

66. Thongtanunam, P., Shang, W. & Hassan, A. Will this clone be short-lived? Towards a better understanding of the characteristics of short-lived clones. *Empirical Software Engineering*. **24**, 937-972 (2019)

67. Buch, L. & Andrzejak, A. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. *SANER 2019 - Proceedings Of The 2019 IEEE 26th International Conference On Software Analysis, Evolution, And Reengineering*. pp. 95-104 (2019)

68. Alon, U., Zilberstein, M., Levy, O. & Yahav, E. Code2vec: Learning distributed representations of code. *Proceedings Of The ACM On Programming Languages.* **3** (2019)

69. Fernandes, P., Allamanis, M. & Brockschmidt, M. Structured neural summarization. *7th International Conference On Learning Representations, ICLR 2019.* (2019)

70. Brockschmidt, M., Allamanis, M., Gaunt, A. & Polozov, O. Generative code modeling with graphs. *7th International Conference On Learning Representations, ICLR 2019.* (2019)

71. Cvitkovic, M., Singh, B. & Anandkumar, A. Open vocabulary learning on source code with a graph-structured cache. *36th International Conference On Machine Learning, ICML 2019.* **2019-June** pp. 2662-2674 (2019)

72. Liu, J., Wang, T., Feng, C., Wang, H. & Li, D. A Large-Gap Clone Detection Approach Using Sequence Alignment via Dynamic Parameter Optimization. *IEEE Access.* **7** pp. 131270-131281 (2019)

73. Moore, J., Gelman, B. & Slater, D. A convolutional neural network for language-agnostic source code summarization. *ENASE 2019 - Proceedings Of The 14th International Conference On Evaluation Of Novel Approaches To Software Engineering.* pp. 15-26 (2019)

74. Javed, O., Villazón, A. & Binder, W. JUniVerse: Large-scale jUnit-test analysis in the wild. *Proceedings Of The ACM Symposium On Applied Computing.* **Part F147772** pp. 1768-1775 (2019)

75. Chatley, R., Donaldson, A. & Mycroft, A. The next 7000 programming languages. *Lecture Notes In Computer Science (including Subseries Lecture Notes In Artificial Intelligence And Lecture Notes In Bioinformatics).* **10000** pp. 250-282 (2019)

76. Liu, Z., Wu, Z., Cao, Y. & Wei, Q. Software vulnerable code reuse detection method based on vulnerability fingerprint. *Zhejiang Daxue Xuebao (Gongxue Ban)/Journal Of Zhejiang University (Engineering Science).* **52**, 2180-2190 (2018)

77. Palsberg, J. & Lopes, C. NJR: A normalized Java resource. *Companion Proceedings For The ISSTA/ECOOP 2018 Workshops.* pp. 100-106 (2018)

78. Javed, O. & Binder, W. Large-Scale Evaluation of the Efficiency of Runtime-Verification Tools in the Wild. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC.* **2018-December** pp. 688-692 (2018)

79. Gottschlich, J., Solar-Lezama, A., Tatbul, N., Carbin, M., Rinard, M., Barzilay, R., Amarasinghe, S., Tenenbaum, J. & Mattson, T. The three pillars of machine programming. *MAPL 2018 - Proceedings Of The 2nd ACM SIGPLAN International Workshop On Machine Learning And Programming Languages, Co-located With PLDI 2018.* pp. 69-80 (2018)

80. Alon, U., Zilberstein, M., Levy, O. & Yahav, E. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices.* **53**, 404-419 (2018)

81. Alon, U., Zilberstein, M., Levy, O. & Yahav, E. A general path-based representation for predicting program properties. *Proceedings Of The ACM SIGPLAN Conference On Programming Language Design And Implementation (PLDI)*. pp. 404-419 (2018)

82. Martins, P., Achar, R. & Lopes, C. 50K-C: A dataset of compilable, and compiled, Java projects. *Proceedings - International Conference On Software Engineering*. pp. 1-5 (2018)

83. Markovtsev, V. & Long, W. Public git archive: A big code dataset for all. *Proceedings - International Conference On Software Engineering*. pp. 34-37 (2018)

84. Horschig, S., Mattis, T. & Hirschfeld, R. Do Java programmers write better python? Studying off-language code quality on GitHub. *ACM International Conference Proceeding Series*. **Part F137691** pp. 127-134 (2018)

85. Rigger, M., Marr, S., Kell, S., Leopoldseder, D. & Mössenböck, H. An analysis of x86-64 inline assembly in C programs. *ACM SIGPLAN Notices*. **53**, 84-99 (2018)

86. Rigger, M., Marr, S., Kell, S., Leopoldseder, D. & Mössenböck, H. An analysis of x86-64 inline assembly in C programs. *VEE 2018 - Proceedings Of The 2018 International Conference On Virtual Execution Environments*. pp. 84-99 (2018)

87. Bagly, A. Developing code factoring transformation for FPGA. *CEUR Workshop Proceedings*. **2260** pp. 55-62 (2018)

DéjàVu: A Map of Code Duplicates on GitHub

CRISTINA V. LOPES, University of California, Irvine, USA
PETR MAJ, Czech Technical University, Czech Republic
PEDRO MARTINS, University of California, Irvine, USA
VAIBHAV SAINI, University of California, Irvine, USA
DI YANG, University of California, Irvine, USA
JAKUB ZITNY, Czech Technical University, Czech Republic
HITESH SAJNANI, Microsoft Research, USA
JAN VITEK, Northeastern University, USA

Previous studies have shown that there is a non-trivial amount of duplication in source code. This paper analyzes a corpus of 4.5 million non-fork projects hosted on GitHub representing over 428 million files written in Java, C++, Python, and JavaScript. We found that this corpus has a mere 85 million unique files. In other words, 70% of the code on GitHub consists of clones of previously created files. There is considerable variation between language ecosystems. JavaScript has the highest rate of file duplication, only 6% of the files are distinct. Java, on the other hand, has the least duplication, 60% of files are distinct. Lastly, a project-level analysis shows that between 9% and 31% of the projects contain at least 80% of files that can be found elsewhere. These rates of duplication have implications for systems built on open source software as well as for researchers interested in analyzing large code bases. As a concrete artifact of this study, we have created DéjàVu, a publicly available map of code duplicates in GitHub repositories.

CCS Concepts: • **Information systems → Near-duplicate and plagiarism detection**; • **Software and its engineering → Ultra-large-scale systems**;

Additional Key Words and Phrases: Clone Detection, Source Code Analysis

ACM Reference Format:
Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 84 (October 2017), 28 pages. https://doi.org/10.1145/3133908

1 INTRODUCTION

The advent of web-hosted open source repository services such as GitHub, BitBucket and SourceForge have transformed how source code is shared. Creating a project takes almost no effort and is free of cost for small teams working in the open. Over the last two decades, millions of projects have been shared, building up a massive trove of free software. A number of these projects have been widely adopted and are part of our daily software infrastructure. More recently there have been attempts to treat the open source ecosystem as a massive dataset and to mine it in the hopes of finding patterns of interest.

Authors' addresses: Cristina V. Lopes, University of California, Irvine, USA; Petr Maj, Czech Technical University, Czech Republic; Pedro Martins, University of California, Irvine, USA; Vaibhav Saini, University of California, Irvine, USA; Di Yang, University of California, Irvine, USA; Jakub Zitny, Czech Technical University, Czech Republic; Hitesh Sajnani, Microsoft Research, USA; Jan Vitek, Northeastern University, USA.

2475-1421/2017/10-ART84
https://doi.org/10.1145/3133908

When working with software, one may want to make statements about applicability of, say, a compiler optimization or a static bug finding technique. Intuitively, one would expect that a conclusion based on a software corpus made up of thousands of programs randomly extracted from an Internet archive is more likely to hold than one based on a handful of hand-picked benchmarks such as [Blackburn et al. 2006] or [SPEC 1998]. For an example, consider [Richards et al. 2011] which demonstrated that the design of the Mozilla optimizing compiler was skewed by the lack of representative benchmarks. Looking at small workloads gave a very different picture from what could be gleaned by downloading thousands of websites.

Scaling to large datasets has its challenges. Whereas small datasets can be curated with care, larger code bases are often obtained by random selection. If GitHub has over 4.5 million projects, how does one pick a thousand projects? If statistical reasoning is to be applied, the projects must be independent. Independence of observations is taken for granted in many settings, but with

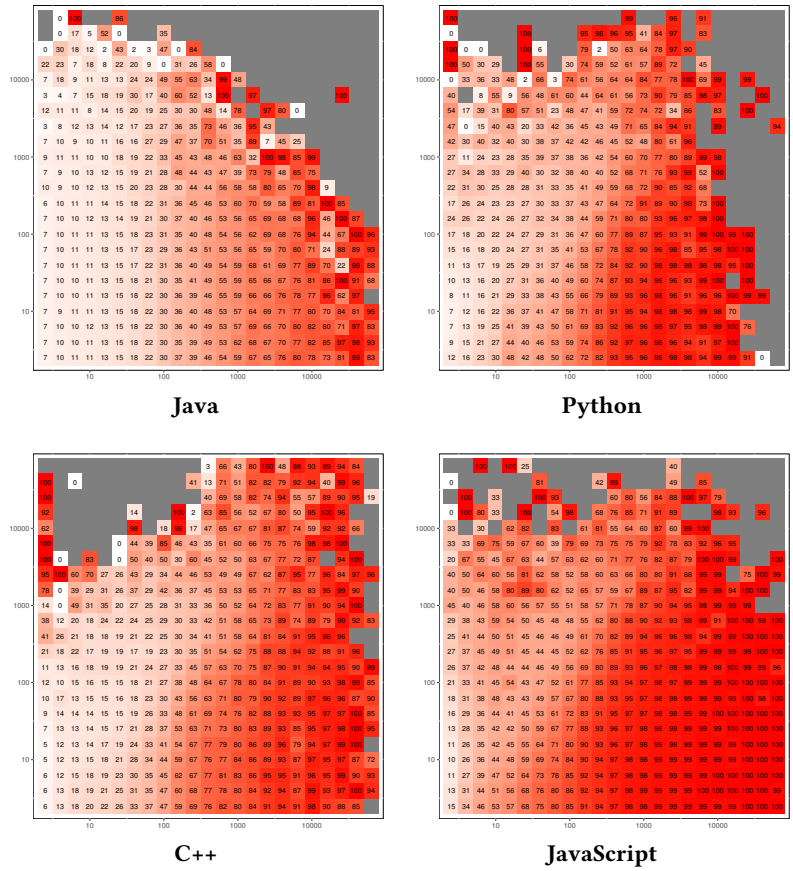


Fig. 1. Map of code duplication. The y-axis is the number of commits per project, the x-axis is the number of files in a project. The value of each tile is the percentage of duplicated files for all projects in the tile. Darker means more clones.

software there are many ways one project can influence another. Influences can originate from the developers on the team, for instance the same people will tend to write similar code. Even more common are the various means of software reuse. Projects can include other projects. Apache Commons is used in thousands of projects, Oracle's SDK is universally used by any Java project, JQuery by most websites. StackOverflow and other discussion forums encourage the sharing of code snippets. Cut and paste programming where code is lifted from one project and dropped into another is another way to inject dependencies. Lastly, entire files can be copied from one project to the next. Any of these actions, at scale, may bias results of research.

Several published studies either neglected to account for duplicates, or addressed them before analysis. [Casalnuovo et al. 2015] studied the use of assertions in the top 100 most popular C and C++ projects in GitHub. [Ray et al. 2014] studied software quality using the top 50 most popular projects in 17 languages. Neither addressed file duplication. Conversely, [Hoffa 2016] studied the old "tabs v. spaces" issue in 400K GitHub projects; file duplication was identified as an issue and eliminated before analysis. [Cosentino et al. 2016] present a meta-analysis of studies on GitHub projects where trends and problems related to dataset selection are identified.

This paper provides a tool to assist selecting projects from GitHub. DéjàVu is a publicly available index of file-level code duplication. The novelty of our work lies partly in its scale; it is an index of duplication for the entire GitHub repository for four popular languages, Java, C++, Python and JavaScript. Figure 1 illustrates the proportion of duplicated files for different project sizes and numbers of commits (section 5 explains how these heatmaps were generated). The heatmaps show that as project size increases the proportion of duplicated files also increases. Projects with more commits tend to have fewer project-level clones. Finally JavaScript projects have the most project-level clones, while Java projects have the fewest.

Table 1. File-hash duplication in subsets.

	10K Stars	10K Commits
Java	9%	6%
C/C++	41%	51%
Python	28%	44%
JavaScript	44%	66%

The clone map from which the heatmaps were produced is our main contribution. It can be used to understand the similarity relations in samples of projects or to curate samples to reduce duplicates. Consider for instance a subset that focuses on the most active projects, as done in [Borges et al. 2016], by filtering on the number of stars or commits a project has. For example, the clones for the 10K most popular projects are summarized in Figure 1. In Java, this filter is reasonably efficient at reducing the number of clones. In other languages clones remain prevalent. DéjàVu can be used to curate datasets, i.e. remove projects with too many clones. Besides applicability to research, our results can be used by anyone who needs to host large amounts of source code to avoid storing duplicate files. Our clone map can also be used to improve tooling, e.g. being queried when new files are added to projects to filter duplicates.

At the outset of this work, we were planning to study different granularities of duplication. As the results came in, the staggering rate of file-level duplication drove us to select three simple levels of similarity. A *file hash* gives a measure of file that are copied across projects without changes. A *token hash* captures minor changes in spaces, comments and ordering. Lastly, SourcererCC captures files with 80% token-similarity. This gives an idea of how many files have been edited after cloning. Our choice of languages was driven by the popularity of these languages, and by the fact that two are statically typed and two have no type annotations. This can conceivably lead to differences in the way code is reused. We expected to answer the following questions: How much code cloning is there, how does cloning affect datasets of software written in different languages, and through which processes does duplication come about? This paper describes our methodology, details the corpus that we have selected and gives our answers to these questions. Along with the quantitative analysis, we provide a qualitative analysis of duplicates on a small number of examples.

Artifacts. The lists of clones, code for gathering data, computing clones, data analysis and visualization are at: http://mondego.ics.uci.edu/projects/dejavu. Processing was done on a Dell PowerEdge R830 with 56 cores (112 threads) and 256G of RAM. The data took 2 months to download and 6 weeks to process.

2 RELATED WORK

Code clone detection techniques have been documented in the literature since the early 90s. Readers interested in a survey of the early work are referred to [Koschke 2007; Roy and Cordy 2007]. There are also benchmarks for assessing the performance of tools [Roy and Cordy 2009; Svajlenko and Roy 2015]. The pipeline we used includes SourcererCC, a token-based code clone detection tool that is freely available and has been compared to other similar tools using those benchmarks [Sajnani 2016; Sajnani et al. 2016].[1] SourcererCC is the most scalable tool so far for detecting Type 3 clones. Type 3 clones are syntactically similar code fragments that differ at the statement level. The fragments have statements added/modified/removed with respect to each other.

One of the earliest studies of inter-project cloning, [Kamiya et al. 2002] analyzed clones across three different operating systems. They found evidence of about 20% cloning between FreeBSD and NetBSD and less than 1% between Linux and FreeBSD or NetBSD. This is explained by the fact that Linux originated and grew independently. [Mockus 2007] performed an analysis of popular open source projects, including several versions of Unix and several popular packages; 38K projects and 5M files. The concept of duplication there was simply based on file names. Approximately half of the file names were used in more than one project. Furthermore, the study also tried to identify components that were duplicated among projects by detecting directories that share a large fraction of their files. Both [Mockus 2007] and [Mockus 2009] use only a fraction of our dataset and a single similarity metric, as opposed to the 3 metrics we provide.

A few studies have focused on block-level cloning, i.e. portions of code smaller than entire files. [Roy and Cordy 2010] analyzed clones in twenty open source C, Java and C# systems. They found 15% of the C files, 46% of the Java files, and 29% of C# files are associated with exact block-level clones. Java had a higher percentage of clones because of accessors methods in Swing. [Heinemann et al. 2011] computed block-level clones consisting of at least 15 statements between 22 commonly reused Java frameworks consisting of more than 6 MLOC and 20 open source Java projects. They did not find any clones for 11 projects. For 5 projects, they found cloning to be below 1% and for the remaining 4, they found up to 10% cloning. These two studies give conflicting accounts of block-level code duplication.

Closer to our study, an analysis of file-level code cloning on Java projects is presented by [Ossher et al. 2011]. This work, analyzed 13K Java projects with close to 2M files. The authors created a system that merges various clone detection techniques with various degrees of confidence, starting on the highest: MD5 hashes; name equivalence through Java's full-qualified names. They report 5.2% file-hash duplication, considerably lower than what we found. Our corpus is three orders of magnitude larger than Ossher's. Furthermore, intra-project duplication meant to deal with versioning was excluded. They looked at subversion, which may have different practices than git, especially related to versioning. We speculate that the practice of copying source code files in open source has become more pervasive since that study was made, and that sites like GitHub simplify copying files among projects, but we haven't reanalyzed the dataset as it is not relevant to the DéjàVu map.

[1] http://github.com/Mondego/SourcererCC

Over the past few years, open source repositories have turned out to be useful to validate beliefs about software development and software engineering in general. The richness of the data and the potential insights that it represents has created an entire community of researchers. [Kochhar et al. 2013] used 50K GitHub repositories to investigate the correlation between the presence of test cases and various project development characteristics, including the lines of code and the size of development teams. They removed toy projects and included famous projects such as JQuery and Rails in their dataset. [Vendome et al. 2016] study how licensing usage and adoption changes over a period of time on 51K repositories. They choose repositories that (i) were not forks; and (ii) had at least one star. [Borges et al. 2016] analyze 2.5K repositories to investigate the factors that impact their popularity, including the identification of the major patterns that can be used to describe popularity trends.

The software engineering research community is increasingly examining large number of projects to test hypotheses or derive new knowledge about the software development process. However, as [Nagappan et al. 2013] point out, more is not necessarily better, and selection of projects plays an important role – more so now than ever, since anyone can create a repository for any purpose at no cost. Thus, the quality of data gathered from these software repositories might be questionable. For example, as we also found out, repositories often contain school assignments, copies of other repositories, images and text files without any source code. [Kalliamvakou et al. 2014] manually analyzed a sample of 434 GitHub repositories and found that approximately 37% of them were not used for software development. As a result, researchers have spent significant effort into collecting, curating, and analyzing data from open source projects around the world. Flossmetrics [Gonzalez-Barahona et al. 2010] and Sourcerer [Ossher et al. 2009] collect data and provide statistics. [Dyer et al. 2013] have curated a large number of Java repositories and provide a domain specific language to help researchers mine data about software repositories. Similarly [Bissyande et al. 2013] have created Orion, a prototype for enabling unified search to retrieve projects using complex search queries linking different artifacts of software development, such as source code, version control metadata, bug tracker tickets, developer activities and interactions extracted from hosting platform. Black Duck Open Hub (www.openhub.net) is a public directory of free and open source software, offering analytics and search services for discovering, evaluating, tracking, and comparing open source code and projects. It analyzes both the code's history and ongoing updates to provide reports about the composition and activity of project code bases. These platforms are useful for researchers to filter out repositories that are interesting to study a given phenomenon by providing various filters. While these filters are useful to validate the integrity of the data to some extent, certain subtle factors when unaccounted for can heavily impact the validity of the study. Code duplication is one such factor. For example, if the dataset consists of projects that have hundreds and thousands of duplicate projects that are part of the same dataset, the overall lack of diversity in the dataset might lead to incorrect observations, as pointed out by [Nagappan et al. 2013].

3 ANALYSIS PIPELINE

Our analysis pipeline is outlined in Figure 2. The pipeline starts with local copies of the projects that constitute our corpus. From here, code files are scanned for fact extraction and tokenization. Two of the facts are the hashes of the files and the hashes of the tokens of the files. File hashes identify exact duplicates; token hashes allow catch clones up with minor differences. While permutations of same tokens may have the same hash, they are unlikely. Clones are dominated by exact copies, and we did not observe any such collision in randomly sampled pairs. Files with distinct token hashes are used as input to the near-miss clone detection tool, SourcererCC. While our JavaScript

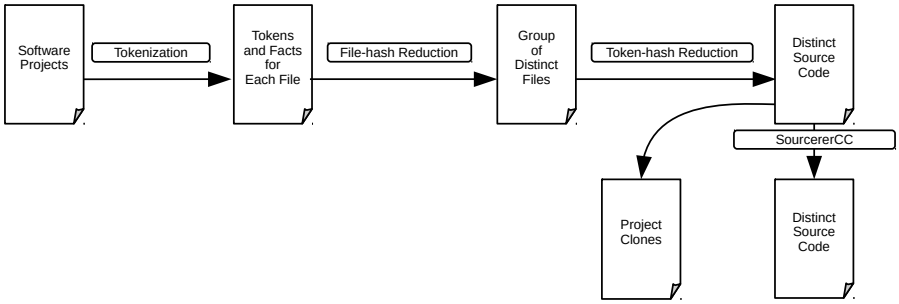


Fig. 2. Analysis pipeline.

pipeline was developed independently, data formats, database schema and analysis scripts are identical.

3.1 Tokenization

Tokenization transforms a file into a "bag of words," where occurrences of each word are recorded. Consider, for instance, the Java program:

```
package foo;
public class Foo {  // Example Class
  private int x;
  public Foo(int x) { this.x = x; }
  private void print() { System.out.println("Number: " + x)  }
  public static void main() {  new FooNumber(4).print();  } }
```

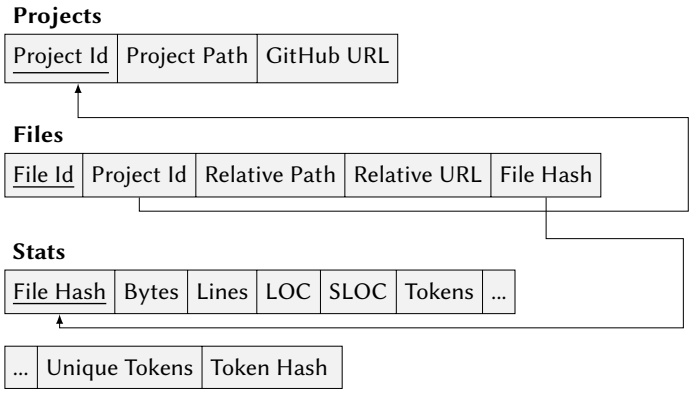
Tokenization removes comments, white space, and terminals. Tokens are grouped by frequency, generating:

```
Java Foo:[(package,1),(foo,1),(public,3),(class,1),(Foo,2),(private,2),(int
    ,2),(x,5),
(this,1),(void,2),(print,2),(System,1),(out,1),(println,1),(Number,1),(
    static,1),
(main,1),(new,1),(FooNumber,1),(4,1)]
```

The tokens package and `foo` appear once, `public` appears three times, etc. The order is not relevant. During tokenization we also extract additional information: (1) *file hash* – the MD5 hash of the entire string that composes the input file; (2) *token hash* – the MD5 hash of the string that constitutes the tokenized output; (3) *size* in bytes; (4) *number of lines*; (5) *number of lines of code* without blanks; (6) *number of lines of source* without comments; (7) *number of tokens*; and (8) *number of unique tokens*. The tokenized input is used both to build a relational database and as input to SourcererCC. The use of MD5 (or any hashing algorithm) runs the risk of collisions, given the size of our data they are unlikely to skew the results.

3.2 Database

The data extracted by the tokenizer is imported into a MySQL database. The table `Projects` contains a list of projects, with a unique identifier, a path in our local corpus and the project's URL. `Files` contains a unique id for a file, the id of the project the file came from, the relative paths and URLs of the file and the file hash. The statistics for each file are stored in the table `Stats`, which contains the information extracted by the tokenizer. The tokens themselves are not imported. The



`Stats` table has the file hash as unique key. With this, we get an immediate reduction from files to hash-distinct files. Two files with distinct file hashes may produce the exact same tokens, and, therefore the same token hash. This could happen when the code of one file is a permutation of another. The converse does not hold: files with distinct token hashes must have come from files with distinct file hashes. For source code analysis, file hashes are not necessarily the best indicators of code duplication; token hashes are more robust to small perturbations. We use primarily token hashes in our analysis.

3.3 Project-Level Analysis

Besides file-level analysis, we also look for projects with significant overlap with other projects. This is done with a script that queries the database making an intersection of the project files' distinct token hashes. This script produces pairs of projects that have significant overlap in at least one direction. The results are of the form: *A cloned in B at x%, B cloned in A at y%*, where *x%* of project A's files (in tokenized form) are found also in project B, and *y%* of project B's files (in tokenized form) are found in project A. Calculating project-level information is done in two steps. First, collect all the files from a project A, say, for example there are 4 files in A: Then find the token-hash duplicates for each of these files in other projects. It might be something like:

```
project A
  File1 - B, B, C
  File2 - B
  File3 -
  File4 - B, D, F
```

There are 3 files from A with duplicates in B, making A a clone of B at 75%. Conversely, there are 4 files in B with duplicates in A; assuming B has a total of 20 files, then B is cloned in A at 20%. A file can be in other project multiple times (e.g. in different directories) as is File 1.

3.4 SourcererCC

The concept of inexact code similarity has been studied in the code cloning literature. Blocks of code that are similar are called near-miss clones, or near-duplication [Cordy et al. 2004]. SourcererCC estimates the amount of near-duplication in GitHub with a "bag of words" model for source code rather than more sophisticated structure-aware clone detection methods. It has been shown to have good precision and recall, comparable to more sophisticated tools [Sajnani 2016]. Its input consists of non-empty files with distinct token hashes. SourcererCC finds clone pairs between

Table 2. GitHub Corpus.

		Java	C++	Python	JavaScript
Counts	# projects (total)	3,506,219	1,130,879	2,340,845	4,479,173
	# projects (non-fork)	1,859,001	554,008	1,096,246	2,011,875
	# projects (downloaded)	1,481,468	369,440	909,290	1,778,679
	# projects (analyzed)	1,481,468	364,155	893,197	1,755,618
	# files (analyzed)	72,880,615	61,647,575	31,602,780	261,676,091
Medians	Files/project	9 (σ = 600)	11 (σ = 1304)	4 (σ = 501)	6 (σ = 1335)
	SLOC/file	41 (σ = 552)	55 (σ = 2019)	46 (σ = 2196)	28 (σ = 2736)
	Stars/project	0 (σ = 71)	0 (σ = 119)	0 (σ = 99)	0 (σ = 324)
	Commits/project	4 (σ = 336)	6 (σ = 1493)	6 (σ = 542)	6 (σ = 275)

these files at a given level of similarity. We have selected 80% similarity as this has given good empirical results. Ideally one could imagine varying the level of similarity and reporting a range of results. But this would be computationally expensive and, given the relatively low numbers of near-miss clones, would not affect our results.

4 CORPUS

The GitHub projects were downloaded using the GHTorrent database and network [Gousios 2013] which contains meta-data such as number of stars, commits, committers, whether projects are forks, main programming language, date of creation, etc., as well as download links. While convenient, GHTorrent has errors: 1.6% of the projects were replicated entries with the same URL; only the youngest of these was kept for the analysis.

Table 2 gives the size of the different language corpora. We skipped forked projects as forks contain a large amount of code from the original projects, retaining those would skew our findings. Downloading the projects was the most time-consuming step. The order of downloads followed the GHTorrent projects table, which seems to be roughly chronological. Some of the URLs failed to produce valid content. This happened in two cases: when the projects had been deleted, or marked private, and when development for the project happens in branches other than master. Thus, the number of downloaded projects was smaller than the number of URLs in GHTorrent. For each language, the files analyzed were files whose extensions represent source code in the target languages. For Java: `.java`; for Python: `.py`; for JavaScript: `.js`, for C/C++: `.cpp` `.hpp` `.HPP` `.c` `.h` `.C` `.cc` `.CPP` `.c++` and `.cp`. Some projects did not have any source code with the expected extension, they were excluded.

The medians in Table 2 give additional properties of the corpus, namely the number of files per (non-empty) project, the number of Source Lines of Code (SLOC) per file, the number of stars and the number of commits of the projects. In terms of files per project, Python and JavaScript projects tend to be smaller than Java and C++ projects. C++ files are considerably larger than any others, and JavaScript files are considerably smaller. None of these numbers is surprising. They all confirm the general impression that a large number of projects hosted in GitHub are small, not very active, and not very popular. Figures 3 and 4 illustrate the basic size-related properties of the projects we analyzed, namely the distribution of files per project and the distribution of Source Lines of Code (SLOC) per file. For JavaScript we give data with and without NPM (it is a cause of a large number of clones). Without NPM means that we ignored files downloaded by the Node Package Manager.

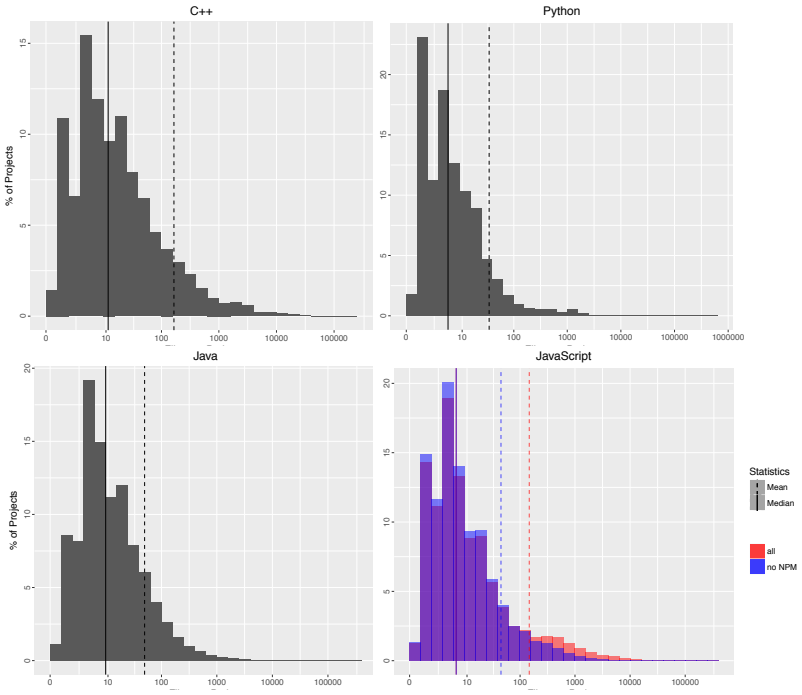


Fig. 3. Files per project.

5 QUANTITATIVE ANALYSIS

We present analyses of the data at two levels of detail: file and project level. This section focuses exclusively on quantitative analysis; the next section delves deeper into qualitative observations.

5.1 File-Level Analysis

Table 3 shows a summary of the findings for files. "SCC dup files" is the number of files, out of the distinct token-hash files, that SourcererCC has identified as clones; similarly, "SCC unique files" is the number of files for which no clones were detected. Figure 5 (top row) charts the numbers in Table 3. The duplicated files (dark grey) are the files that are duplicate of at least one of the distinct token-hash files (light grey); further, the distinct token-hash files are split between those for which SourcererCC found at least one similar file (cloned files, grey) and those for which SourcererCC did not find any similar file (unique files, in white).

These numbers show a considerable amount of code duplication, both exact copies of the files (file hashes), exact copies of the files' tokens (token hashes), and near-duplicates of files (SourcererCC). The amount of duplication varies with the language: the JavaScript ecosystem contains the largest amount of duplication, with 94% of files being file-hash clones of the other 6%; the Java ecosystem contains the smallest amount, but even for Java, 40% of the files are duplicates; the C++ and Python ecosystems have 73% and 71% copies, respectively. As for near-duplicates, Java contains the largest percentage: 46% of the files are near-duplicate clones. The ratio of near-miss clones is 43% for Java, 39% for JavaScript, and 32% for Python.

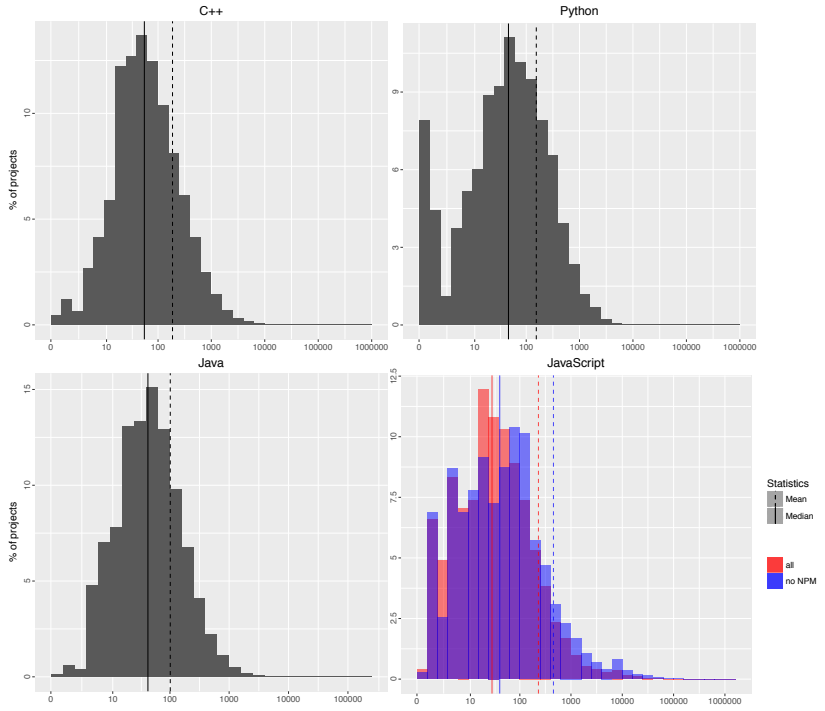


Fig. 4. SLOC per file.

The heatmaps (Figure 1) shown in the beginning of the paper were produced using the number of commits shown in Table 2, the number of files in each project, and the file hashes. The heat intensity corresponds to the ratio of file hashes clones over total files for each cell.

Duplication can come in many flavors. Specifically, it could be evenly or unevenly distributed among all token hashes. We found these distributions to be highly skewed towards small groups of files. In Java 1.5M groups of files with the same token-hash have either 2 or 3 files in them; the number of token hash-equal groups with more than 100 files is minuscule. The same observation holds for the other languages. Another interesting piece of information about clone groups is given by the largest extreme. In Python, the largest group of file-hash clones has over 2.5M files. In Java,

Table 3. File-Level Duplication.

	Java	C++	Python	JavaScript
Total files	72,880,615	61,647,575	31,602,780	261,676,091
File hashes	43,713,084 (60%)	16,384,801 (27%)	9,157,622 (29%)	15,611,029 (6%)
Token hashes	40,786,858 (56%)	14,425,319 (23%)	8,620,326 (27%)	13,587,850 (5%)
SCC dup files	18,701,593 (26%)	6,200,301 (10%)	2,732,747 (9%)	5,245,470 (2%)
SCC unique files	22,085,265 (30%)	8,225,018 (13%)	5,887,579 (19%)	8,342,380 (3%)

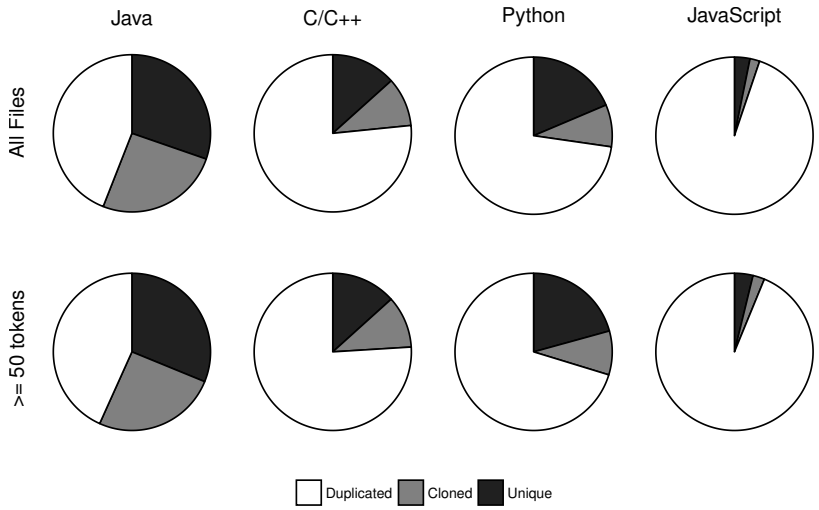


Fig. 5. File-level duplication for entire dataset and excluding small files.

the largest group of SourcererCC clones has over 65K files. In the next section we show which files these are.

5.2 File-Level Analysis Excluding Small Files

One observation that emerged immediately from all the language ecosystems was that the most duplicated file is the empty file – a file with no content, and size 0. In the Python corpus alone, there are close to 2.2M occurrences of this trivial file, and in the JavaScript corpus there are 986K occurrences of that same file. Another frequently occurring trivial file in all ecosystems is a file with 1 empty line. Indeed, a common pattern that emerged was that the most duplicated files tend to be very small. Once we detected that, we redid the analysis excluding small files. Specifically, we excluded all files with less than 50 tokens.[2] Table 4 and Figure 5 (bottom row) show the results.

Although the absolute number of files and hashes change significantly, the changes in ratios of the hashes and SCC results are small. When they are noticeable, they show that there is slightly less duplication in this dataset than in the entire dataset. Comparing Table 4 with Table 3 shows

[2]This threshold is arbitrary. It is based on our observations of small files; other values can be used.

Table 4. File-level duplication excluding small files.

	Java	C++	Python	JavaScript
# of files	57,240,552	49,507,006	23,382,050	162,136,892
% of corpus	79%	80%	74%	62%
File hashes	34,617,736 (60%)	13,401,948 (27%)	7,267,097 (31%)	11,444,667 (7%)
Token hashes	32,473,052 (58%)	11,893,435 (24%)	6,949,894 (30%)	10,074,582 (6%)
SCC dup files	14,626,434 (26%)	5,297,028 (10%)	2,105,769 (9%)	3,896,989 (2%)
SCC unique files	17,848,618 (31%)	6,596,407 (13%)	4,844,125 (21%)	6,177,593 (4%)

that small files account for a slightly higher presence of duplication, but not that much higher than the rest of the corpus.

5.3 Inter-Project Analysis

So far, we investigated how code duplication is rampant at the file level. The next question is how this finding maps into projects: how many projects are exact and near-duplicates of other projects, even though they are not technically forks? This is called **inter**-project cloning. For that, and as explained in Section 3, we computed the overlap of files between projects, as given by the files' **token hashes**. We used the entire corpus, including the small files, as these are important for the projects. The results are shown in Table 5 and Figure 6.

Table 5. Inter-project cloning.

	Java	C++	Python	JavaScript
# projects (analyzed)	1,481,468	364,155	893,197	1,755,618
# clones ≥ 50%	205,663 (14%)	94,482 (25%)	159,224 (18%)	854,300 (48%)
# clones ≥ 80%	135,168 (9%)	58,906 (16%)	94,634 (11%)	546,207 (31%)
# clones 100%	87,220 (6%)	24,851 (7%)	51,589 (6%)	273,970 (15%)
# exact dups	73,869 (5%)	19,809 (5%)	43,501 (5%)	198,556 (11%)
# exact dups (≥ 10 files)	37,722 (3%)	10,286 (3%)	7,331 (1%)	78,972 (4%)

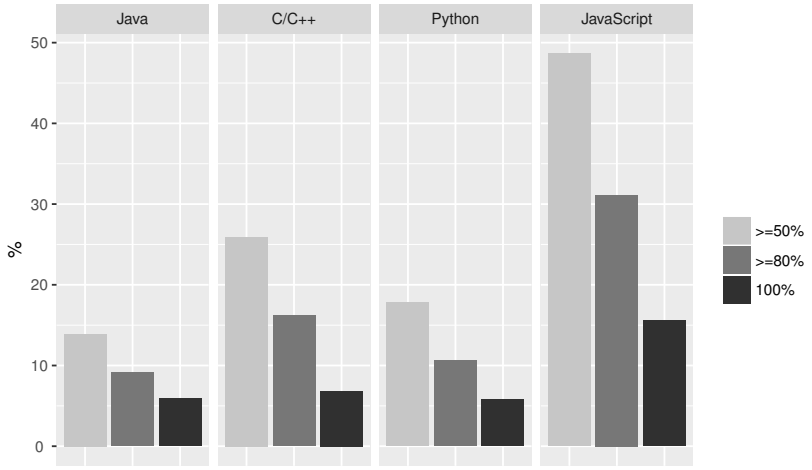


Fig. 6. Percentage of project clones at various levels of overlap.

Table 5 shows the number of projects whose files exist in other projects at some overlap threshold – 50%, 80% and 100%, respectively. A normalization of these numbers over the total number of projects for each language is shown in Figure 6. JavaScript comes on top with respect to the amount of project-level duplication, with 48% of projects having 50% or more files duplicated in some other project, and an equally impressive 15% of projects being 100% duplicated.[3] Not surprisingly, the

[3]Again, we remind the reader that our dataset does not contain forks.

percentage of project-level duplication tracks the percentage of file-level duplication, as shown in Figure 5. The differences seen between the language ecosystems do not seem to be related to the size of projects: Table 2 in Section 4 shows that the median files per project in JavaScript is slightly higher than in Python (so, JavaScript projects tend to have more files), but the inter-project cloning is much higher for JavaScript than for Python. We will dive more into this in the next section.

The last two rows of Table 5 show the number of projects that are token-hash clones of some other project (apart from differences in white space, comments, and terminal symbols). This is different, and more constrained, than being cloned at 100% elsewhere: it requires bidirectionality. With the exception of JavaScript at 11%, the ratios are all 5%, but it is still surprising that there are so many projects that are exact copies of each other. As the last row shows, though, many of those are very small projects, with less than 10 files. The number for projects with at least 10 files that are exact copies of some other project is considerably smaller, but still in the thousands for all languages.

6 MIXED METHOD ANALYSIS

The numbers presented in the previous section portray an image of GitHub not seen before. However, that quantitative analysis opens more questions. What files are being copied around, and why? What explains the differences between the language ecosystems? Why is the JavaScript ecosystem so much off the charts in terms of duplication? In order to answer these kinds of questions, we delve deeper into the data.

With so much data, our first heuristic was size. As seen in the previous section we noticed that the empty file was the most duplicated file in the entire corpus, among all languages. We also noticed that the top duplicated files tended to be very small and relatively generic. Although an intriguing finding, very small, generic files hardly provide any insightful information about the practice of code duplication. What about the non-trivial files that are heavily duplicated? What are they?

This section presents observations emerging from looking at specific files and projects using mixed methods. We divide the section into four parts: (1) an analysis of each language ecosystems looking for the most duplicated files in general; (2) file duplication at different levels (file hashes, token hashes and near duplicates with SourcererCC); (3) the most reappropriated projects in the four ecosystems; and (4) an in-depth analysis of the JavaScript ecosystem.

6.1 Most Duplicated Non-Trivial Files

As stated above, we wanted to find out if the size of the files had an effect on their duplication. For example, are small files copy-pasted from StackOverflow or online tutorials and blogs, and large files from well-known supporting libraries? In order to make sense of so much data, we needed to sample it first, so that interesting hypotheses could emerge, and/or we could find counter-examples that contradicted our initial expectations. This is territory of qualitative and mixed methods [Creswell 2014].

6.1.1 Methodology. We used a mixed method approach consisting of qualitative and quantitative elements. Based on our quantitative analysis, we hypothesized that size of the files, and whether the duplication was exact or token-based, might have an effect on the nature of duplication; for example, the empty file certainly is not being copy-pasted from one project to another, it simply is created in many projects, for a variety of reasons. Maybe we could see patterns emerge for files of different sizes. The following describes our methodology:

- **Quantitative Elements.** We split files according to the percentiles of the number of tokens per file within each language corpus, and create bins representing the ranges 20%-30% (small),

Table 6. Number of tokens per file within certain percentiles of the distribution of file size.

		20%-30%	45%-55%	70%-80%	90%+
Tokens	Java	46-71	120-167	279-419	751+
	C/C++	50-77	138-199	372-623	1284+
	Python	29-65	149-236	477-795	1596+
	JavaScript	19-32	68-114	238-431	1127+
Files	Java	7,670,926 (11%)	7,523,679 (10%)	7,335,067 (10%)	7,298,767 (10%)
	C/C++	6,381,850 (10%)	6,228,550 (10%)	6,204,943 (10%)	6,167,647 (10%)
	Python	3,282,957 (10%)	3,205,337 (10%)	3,169,316 (10%)	3,161,325 (10%)
	JavaSript	28,257,319 (11%)	27,306,195 (10%)	26,326,975 (10%)	26,134,513 (10%)

45%-55% (medium), 70%-80% (large), and greater than 90% (very large). So, the 45%-55% bin contains files that are between the 45% percentile and the 55% percentile on the number of tokens per file of a certain language. The number of tokens for the bins can be seen in Table 6. For example in Java, the first bin includes files containing 47 to 72 tokens, and so on. The gaps between these percentiles (for example, no file is observed between the 30% and the 45% percentile) ensure buffer zones that are large enough to isolate the differently-sized files, should differences in their characteristics be observed. For each of these bins, we analyzed the top 20 most cloned files; this grouping was performed twice, using file hashes and token hashes, and this was done for all the languages. In total, for each language, 80 files were analyzed.

- **Qualitative Elements.** Looking at names of most popular files, a first observation was that many of these files came from popular libraries and frameworks, like Apache Cordova. This hinted at the possibility that the origin of file duplication was in well-known, popular libraries copied in many projects; a qualitative analysis of file duplication was better understood from this perspective. Therefore, each file was observed from the perspective of the path relative to the project where it resides, and was then *hand coded* for its origin.[4] For example, `project_name/src/external/com/http-lib/src/file.java` was considered to be part of the external library `http-lib`. Each folder assumed to represent an external library was matched with an existing homepage for the library, if we could find it using Google. Continuing the running example, `http-lib` was only flagged as an external dependency if there was a clear pointer online for a Java library with that name. In some cases, the path name was harder to interpret, for example: p roject_name/external/include/internal/ftobjs.h. In those cases, we searched Google for the last part of the path in order to find the origin (in this particular case, we searched `i nclude/internal/ftobjs.h`). For JavaScript the situation was often simpler: many of the files came from NPM modules, in which case the module name was obvious from the file's location. Some of the files were also minified versions of libraries, in which case the name of the file gave the library name, often with its version (e.g. `jquery-3.2.1.min`). Using these methods, we were able to trace the origins of all the 320 files.

6.1.2 Observations. Contrary to our original expectation, we did not find any differences in the nature of file duplication related to either size of the files, similarity metric, or language in the 320 samples we inspected. We also didn't find any StackOverflow or tutorial files in these samples. Moreover, the results for these files show a pattern that crosses all of those dimensions: the most

[4]For a good tutorial on coding, see [Saldaña 2009]

duplicated files in all ecosystems come from a few well-known libraries and frameworks. The Java files were dominated by the ActionBarSherlock and Cordova. C/C++ was dominated by boost and freetype, and JavaScript was dominated by files from various NPM packages, only 2 cases were from jQuery library. For Python, the origins of file cloning for the 80 files sampled were more diverse, along 6 or 7 common frameworks.[5]

Because the JavaScript sample was so heavily (78 out of 80) dominated by Node packages, we have performed the same analysis again, this time excluding the Node files. This uncovered jQuery in its various versions and parts accounting for more than half of the sample (43), followed from a distance by other popular frameworks such as Twitter Bootstrap (12), Angular (7), reveal (4). Language tools such as modernizr, prettify, HTML5Shiv and others were present. We attribute this greater diversity to the fact that to keep connections small, many libraries are distributed as a single file. It is also a testament to the popularity of jQuery which still managed to occupy half of the list.

The presence of external libraries within the projects' source code shows a form of dependency management that occurs across languages, namely, some dependencies are source-copied to the projects and committed to the projects' repositories, independent of being installed through a package manager or not. Whether this is due to personal preference, operational necessity, or simple practicality cannot be inferred from our data.

Another interesting observation was the proliferation of libraries for being themselves source-included in other widely-duplicated libraries. Take Cordova, a common duplicated presence within the Java ecosystem. Cordova includes the source of okhttp, another common origin of duplication. Similarly, within C/C++, freetype2 was disseminated in great part with the help of another highly dispersed supporting framework, cocos2d. This not only exacerbates the problem, but provides a clear picture of the tangled hierarchical reliance that exists in modern software, and that sometimes is source-included rather than being installed via a package manager.

6.2 File Duplication at Different Levels

In this section, we look in greater detail at the duplication in the three levels reported: file hashes, token hashes and SCC clones:

6.2.1 File Hashes. Top cloned files of various sizes were already analyzed in 6.1. To complement, we have also investigated mostly cloned non-trivial files across all sizes to make sure no interesting files slipped between the bins, but we did not find any new information. Instead we tried to give more precise answer to question which files get cloned most often. Our assumption was that the smaller the file, the more likely it is to be copied. Figure 7 shows our findings. Each file hash is classified by number of copies of the file (horizontal axis) and by size of the file in bytes (vertical axis). Furthermore, we have binned the data into 100x100 bins and we have a logarithmic scale on both axes, which forms the artefacts towards the axes of the graph. The darker the particular bin, the more file hashes it contains. The graphs show that while it is indeed smaller files that get copied most often, with the exception of extremely small outliers (trivial files, such as the empty file), the largest duplication groups can be found for files with sizes in thousands of bytes, with maximum sizes of the clone groups gradually lowering for either larger, or smaller files.

6.2.2 Token Hashes. For a glimpse of the distribution of token hashes, we have investigated the relations between number of files within a token hash group and number of file hashes (i.e.

[5]The very small number of libraries and frameworks found in these samples is a consequence of having sampled only 80 files per language, and the most duplicated ones. Many of the files had the same origin, because those original libraries consist of several files.

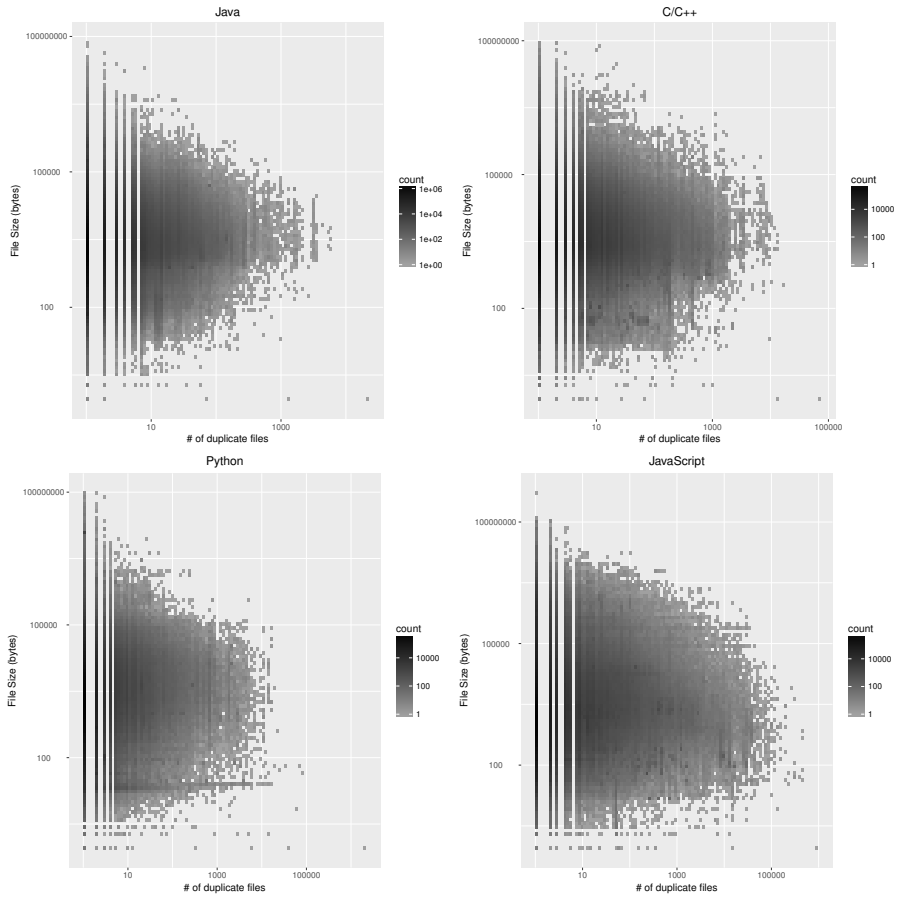


Fig. 7. Distribution of file-hash clones.

different files). These findings are summarized in Figure 8. The outlier in the top-right corner of each graph is the empty file. The number of different empty files is explained by the fact that when using token hash, any file that does not have any language tokens in it is considered empty. Given the multitude of sizes observed within token hash groups, the next step was to analyze the actual difference in sizes within the groups. The results shown in Figure 9 summarize our findings. As expected, for all four languages the empty file again showed very close to the top. For Java, the biggest empty file was 24.3MB and contains a huge number of comments as a compiler test. For C/C++ the empty files has the second largest difference and consists of a comment with ASCII art. Python's empty file was a JSON dump on a single line, which was commented, and finally for JavaScript the largest empty file consisted of thousands of repetitions of an identical comment line, totaling 36MB.

More interesting than largest empty files is the answer to the question: What other, non-trivial files display the greatest difference between sizes in the same group. Interestingly, the answer

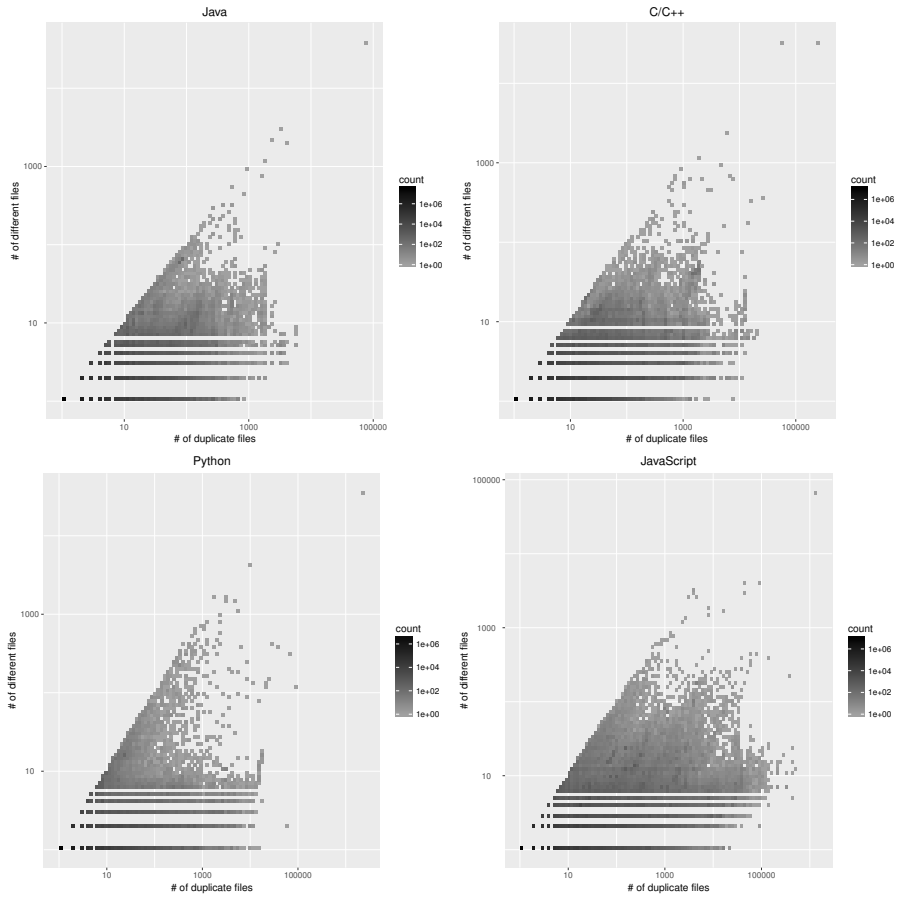


Fig. 8. Distribution of token-hash clones.

is slightly different for each language: for Java, the greatest size differences exist for binary files disguised as java files. In these files, very few tokens were identified by the tokenizer and therefore two unrelated binary files were grouped into a single token group with a small number of very different files. For C/C++ often, we have found source codes with and without hundreds of KB of comments as members of the same groups. An outlier was a file with excessive white-spaces at each line (2.42MB difference). In Python, formatting was most often the cause: a single file multiplied its size 10 times by switching from tabs to 8 spaces. For JavaScript, we observed minified and non-minified versions. Sometimes the files were false positives because complex Javascript regular expressions were treated as comments by the simple cross-language parser.

6.2.3 SourcererCC Duplicates. For SourcererCC, we randomly selected 20 clone pairs and we categorized them into three categories: i) *intentional copy-paste clones*; ii) *unintentional accidental clones*; and iii) *auto-generated clones*. It is interesting to note that the clones in categories ii) and iii) are both unavoidable and are created because of the use of the popular frameworks.

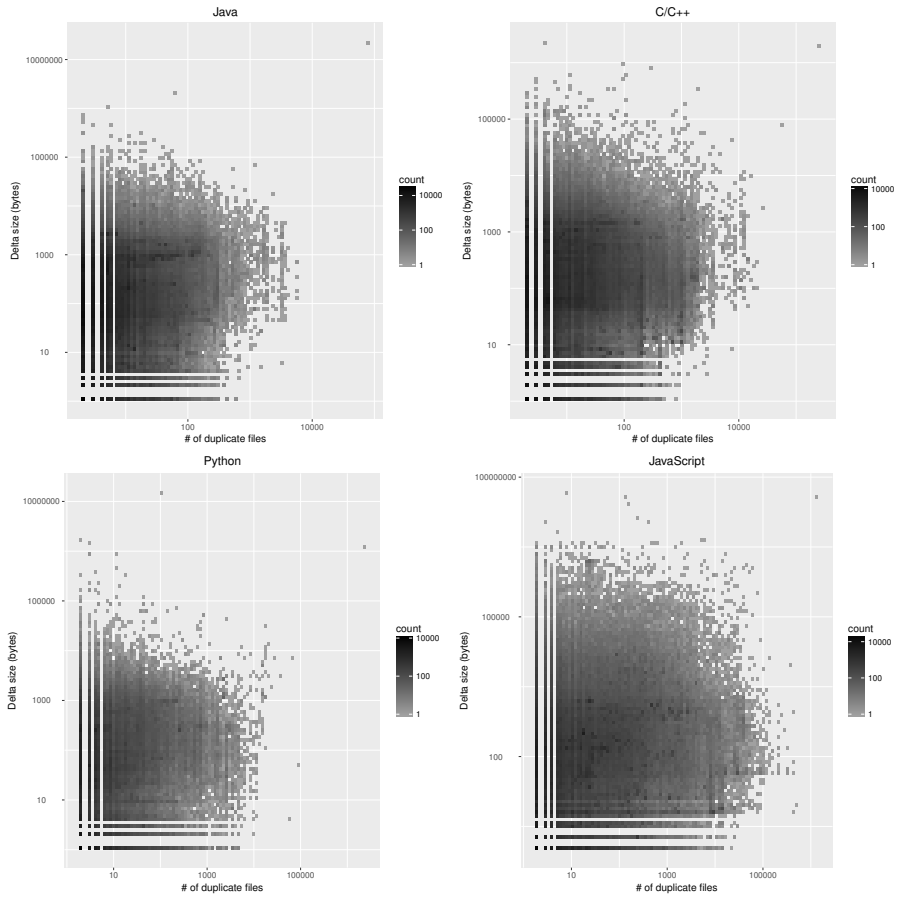


Fig. 9. Δ of file sizes in token hash groups.

Java. We have categorized 30% (6 out of 20) of the clone pairs into the *intentional copy-paste clones* category. It included instances of both inter-project and intra-project clones. Intra-project clones were created to test/implement functionalities that are similar while keeping them isolated and easy to maintain. Inter-project clones seemed to come from projects that look like class projects for a university course and from situations where one project was almost entirely copy-pasted into the other project. We found 2 instances of *unintentional cloning*, both inter-project. The files in such clone pairs implement a lot of similar boilerplate code necessary to create an Android activity class. We categorized the majority (12 out of 20) of the clone pairs into the *auto-generated clones* category. The files in this category are automatically generated from the frameworks like Apache Axis (6 pairs), Android (2 pairs), and Java Architecture for XML Binding (4 pairs). The unintentional and auto-generated clones together constitute 70% of the sample.

C/C++. The sample was dominated by *intentional copy-paste clones* (70%, 12 pairs). The origin for these file clone pairs seems to be the same, independent of these being inter of intra-project

clones, and relates to the reuse of certain pieces of source code after which they suffer small modification to cope with different setups or support different frameworks. Five pairs were classified as *unintentional cloning*. They represented educational situations (one file was composed in its large part by the skeleton of a problem, and the difference between the files clones was the small piece of code that implements the solution). Two different versions of the same file were also found (libpng 1.0.9 vs. libpng 1.2.30). Files from two projects sharing a common ancestor (bitcoin vs dotcoin) were also observed. The *auto-generated clones* were present in three pairs, 2 of them from the Meta-Object compiler.[6] The unintentional and auto-generated clones accounted for 40% of the sample.

Python. The sample was dominated by uses of the Django framework (17 pairs), all variants of auto generated code to initialize a Djagno application. We classified them as *auto-generated clones*. Two pairs were *intentional copy-paste clones* intra-project copy-paste of unittests. The last pair belonged to the same category was a model schema for a Django database.

JavaScript. Only one *intentional copy-paste clones* example has been found, which consisted of a test template with manually changed name, but nothing else. Five occurrences of *unintentional cloning* comprised of pairs of different file versions for jQuery(2), google maps opacity slider, modernizr, and angular socket service. The remaining 14 pairs (70%) have been classified as *auto-generated clones*. Dominated by Angular project files(7), project files for express(3), angular locales and different gruntfiles (builder files for Node projects) were present. All of the Angular project files are created with Yeoman, a tool for creating application skeletons with boilerplate code used also by the Angular Full Stack Generator. The last pair classified as autogenerated was also the only inter-project clone and consisted of two very similar JSON records in a federal election commission dump stored on Github. In total, 95% of the pairs were unintentional or auto-generated.

6.3 Most Reappropriated Projects

We look for projects duplicated in bulk without any addition or change, i.e. with 100% of their files present in a single host project. This captures the practice of reappropriation. Since versioning systems offer features that should be used instead of reappropriation (such as Git submodules) we were interested in how prevalent and for what purposes reappropriation exists. A simple query into the database gave us some insights. Note our analysis is not exhaustive; projects originating from outside GitHub may not be found unless an abandoned project that just reappropriated them exists. But if the project's exact copy will be missed, the files themselves will be identified as clones between projects using the same library.

For Java, we found that *Minecraft-API* and *PhoneGap* are the two most reappropriated projects. Looking for clues online, we found that the original Minecraft-API project was not hosted in GitHub until 2012, so the copies may have been from developers who used GitHub at the time. Also, on further inspection we found that *PhoneGap* is related to *Apache Cordova*. These frameworks might not have been in GitHub from the beginning.

For C++, GNU ISO C++ Library, homework templates, and Arduino examples have been reappropriated the most. The homework case is interesting: it seems that some instructor created a body of code that was then cloned by several dozen students, instead of being forked in GitHub, as one might expect. All clones were exactly the same, which seems to indicate the students didn't push their changes back. This an unorthodox, and somewhat abusive use of GitHub.

[6] http://doc.qt.io/qt-4.8/moc.html

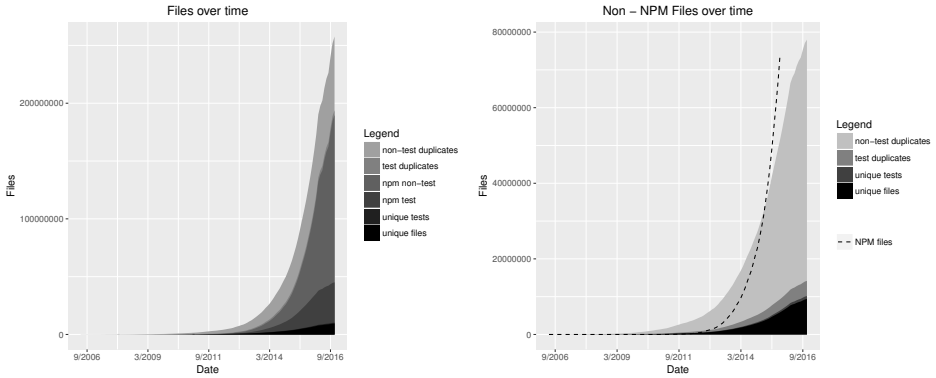


Fig. 10. JavaScript files over time, with and without NPM files.

For Python, the top 3 most reappropriated projects are Cactus, which is a static site generator using Django templates, Shadowsocks, a fast tunnel proxy that helps bypass firewalls, and Scons, a software construction tool.

Finally, for JavaScript the most reappropriated project is the Adobe *PhoneGap*'s Hello World Template[7], which has been found intact in total of 1746 projects. PhoneGap is a framework for building mobile applications using the web stack and it dominates the most frequently cloned projects - the top 15 most cloned projects are all different versions of its template. PhoneGap is followed by the *OctoPress*[8] blogging framework and by a template for *BlueMix*.[9]

These observations show that project reappropriation exists for a variety of reasons: simple reappropriations that could be addressed by Git submodules (e.g. Minecraft API, Arduino), seemingly abandoned derivative development (Cactus, PhoneGap), true forks with addition of non-source code content (OctoPress) and even unorthodox uses of GitHub (the C++ homework).

6.4 JavaScript

JavaScript has the highest clone ratio of the languages studied. Over 94% of the files are file-hash clones. We wanted to find out what is causing this bloat. After manually inspecting several files, we observed that many projects commit libraries available through NPM as if they are part of the application code.[10] As such, we analyzed the data with respect to the effect of NPM libraries, and concluded that this practice is the single biggest cause for the large duplication in JavaScript. What follows are some mostly quantitative perspectives on the effect of NPM libraries, along with some qualitative observations pulled from additional sources. Figure 10 on the left shows the composition of JavaScript repositories over time with respect to unique files and tests and token-hash clones and (we considered any file in `test` folder to be a test) compared with files & tests coming from unorthodox use of NPM. Figure 10 on the right shows the corpus in the same categories, but without the NPM files, whose number is indicated by the dashed line which quickly surpasses all other files in the corpus. The huge impact of NPM files can be seen not only in the sheer number of files, Figure 11 shows the percentage of token-hash clones for different subsets

[7] http://github.com/phonegap/phonegap-template-hello-world
[8] http://octopress.org/
[9] https://www.ibm.com/cloud-computing/bluemix/
[10] `npm` is the package manager used by the very popular Node framework.

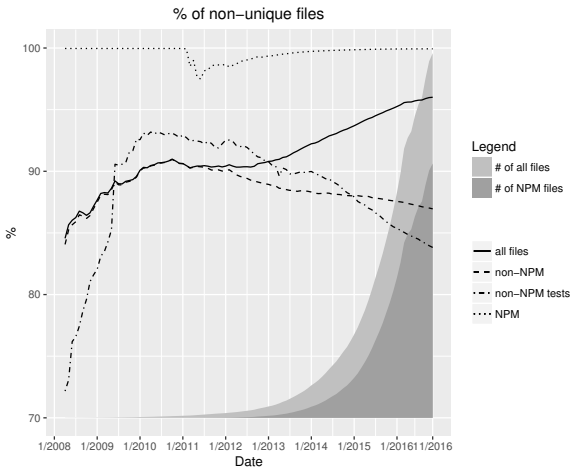


Fig. 11. Percentage of clones over time

of the files over time. To help assess influence, the background of the graph shows the numbers of total and NPM files at given times. Few files predate the NPM Manager itself (January 2010). We have found similar outliers in the rest of the files (small amount of them predating not just GitHub and Git, but even JavaScript itself). As soon as NPM files started to appear in the corpus, they took over the global ratio (solid line), while the rest of the files slowly added original content over time. Interesting is the higher originality of tests – when people copy and paste the entire files, they tend to ignore their tests.

6.4.1 NPM Files. When npm is used in a project, the package.json file contains the description of the project including its required packages. When the project is built, these packages, are loaded and stored in the 'node_modules' directory. If the packages themselves have dependencies, these are stored under the package name in another nested 'node_modules' directory. The 'node_modules' folder will be updated each time the project is built and a new version of some of the packages it transitively requires is available. Therefore it should not be part of the repository itself - a practice GitHub recommends.[11] Since NPM allows dependencies to link to specific versions of the package, there is no need to include the 'node_modules' directory even if the application requires specific package version. Even more surprising than the sheer number of NPM files in the corpus is the number of packages responsible for them. 41.21% (732991) projects use NPM package manager, but only 6% (106582) projects include their 'node_modules' directory. These 6% projects are ultimately responsible for almost 70% of the entire files. It is therefore not surprising that once a project includes its NPM dependencies, its file number is overwhelmed by the packages' files as shown in Figure 12 on the left.

There are even projects that seem to contain only NPM files. Often a project is created using an automated generator which installs various dependencies, pushed to Github with the node_modules directory and never used again. The largest of such projects[12] contains only NPM modules used in other project of the same author and has 46281 files. If the project is written

[11] https://github.com/github/gitignore/blob/master/Node.gitignore
[12] https://github.com/kuangyeheng/workflow-modules

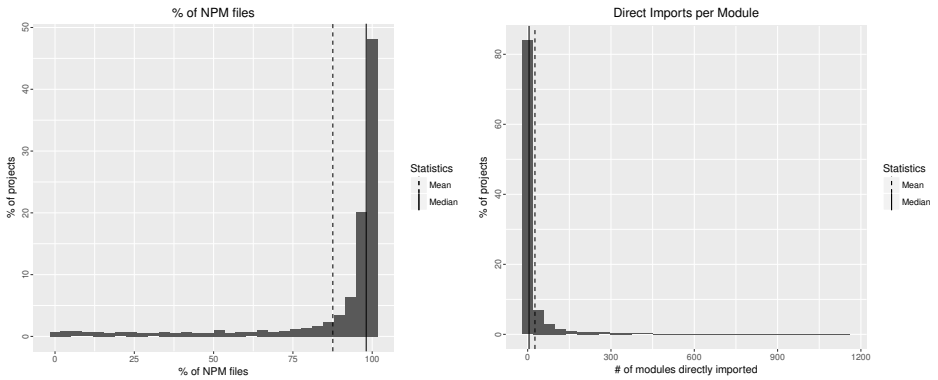


Fig. 12. % of NPM files in projects and directly imported NPM packages

a dialect of Javascript that does not use the `js` extension (such as `jsx` or `TypeScript`) it would appear all its files come from NPM. This is the case of the second largest npm-only project[13] consists of 16761 JS files from NPM and a handful of `jsx` files discovered by manual inspection.

We have also analyzed the depth of nested dependencies in the NPM packages. In the worst case we have observed this nesting to be 47 modules deep with median of 5. The number of unique projects included has median of 63 and maxes out at 1261, but this includes the nested dependencies as well. The direct imports, i.e. modules specified as dependencies in the `package.json` file is in general much smaller as shown in Figure 12 on the right. There are however outliers which come close to the max number of unique projects included. The largests of them has been created by the Angular Full Stack Generator,[14] an automated service for generating Angular applications.[15] Other projects with extraordinarily large direct dependencies are created using similar automated generators, such as Yeoman. In terms of module popularity (Figure 13) (note the log scale on y axis) most modules are imported by a small percent of projects, however there are some massively popular ones: `Express`[16] (59277 projects) is a minimalist web UI framework, body `parser`[17] (31807 projects) a HTTP response body parser and debug[18] (24413 projects), a debugging utility for Node applications. Surprisingly, many of the NPM packages contain a great deal of tests in them, as shown in Figure 10, which seems unnecessary, as these should be release versions of the packages for users, not package for developers.

7 CONCLUSIONS

The source control system upon which GitHub is built, Git, encourages forking projects and independent development of those forks. GitHub provides an easy interface for forking a project, and then for merging code changes back to the original projects. This is a popular feature: the metadata available from GHTorrent shows an average of 1 fork per project. However, there is a lot more duplication of code that happens in GitHub that does not go through the fork mechanism, and, instead, goes in via copy and paste of files and even entire libraries.

[13] https://github.com/george-codes/react-skeleton
[14] https://github.com/angular-fullstack/generator-angular-fullstack
[15] Ironically the project itself was created to let people “quickly set up a project following best practices”.
[16] https://www.npmjs.com/package/express
[17] https://www.npmjs.com/package/body-parser
[18] https://www.npmjs.com/package/debug

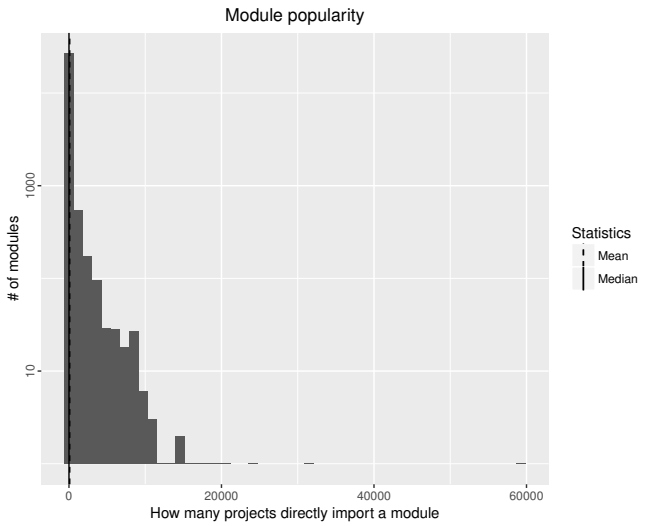


Fig. 13. Popularity of NPM modules.

We presented an exhaustive investigation of code cloning in GitHub for four of the most popular object-oriented languages: Java, C++, Python and JavaScript. The amount of file-level duplication is staggering in the four language ecosystems, with the extreme case of JavaScript, where only 6% of the files are original, and the rest are copies of those. The Java ecosystem has the least amount of duplication. These results stand even when ignoring very small files. When delving deeper into the data we observed the presence of files from popular libraries that were copy-included in a large number projects. We also detected cases of reappropriation of entire projects, where developers take over a project without changes. There seemed to be several reasons for this, from abandoned projects , to slightly abusive uses of GitHub in educational contexts. Finally, we studied the JavaScript ecosystem, which turns out to be dominated by Node libraries that are committed to the applications' repositories.

This study has some important consequences. First, it would seem that GitHub, itself, might be able to compress its corpus to a fraction of what it is. Second, more and more research is being done using large collections of open source projects readily available from GitHub. Code duplication can severely skew the conclusions of those studies. The assumption of diversity of projects in those datasets may be compromised. DéjàVu can help researchers and developers navigate through code cloning in GitHub, and avoid it when necessary.

A APPENDIX: SUMMARY STATISTICS

Table 7. Summary statistics for the entire dataset.

		Java	C++	Python	JavaScript
Files per project	Min	1	1	1	1
	1^{st} Qu	3	3	2	2
	Median	9	11	5	6
	Mean	49	169	35	147
	3^{rd} Qu	24	40	13	22
	Max	375,859	233,844	410,203	255,902
Bytes per file	Min	0	0	0	0
	1^{st} Qu	1,100	1,399	564	348
	Median	2,334	3,114	2,169	1,123
	Mean	5,714	10,340	8,784	11,326
	3^{rd} Qu	5,166	7,779	6,840	3,758
	Max	80M	100M	105M	576M
Lines per file	Min	1	1	1	1
	1^{st} Qu	38	47	20	10
	Median	75	100	66	33
	Mean	164	279	205	265
	3^{rd} Qu	158	237	195	106
	Max	1,437,949	8,129,599	5,861,049	5,105,047
LOC per file	Min	1	1	1	0
	1^{st} Qu	32	38	16	8
	Median	63	83	54	28
	Mean	142	240	174	232
	3^{rd} Qu	135	200	161	93
	Max	1,436,850	8,129,570	5,860,092	5,105,045
SLOC per file	Min	0	0	0	0
	1^{st} Qu	17	23	12	5
	Median	41	55	46	19
	Mean	101	188	156	173
	3^{rd} Qu	97	147	143	75
	Max	1,436,841	8,129,521	5,859,657	5,105,045
Distinct tokens per file	Min	0	0	0	0
	1^{st} Qu	30	31	27	15
	Median	56	57	77	37
	Mean	86	118	197	128
	3^{rd} Qu	100	111	180	92
	Max	1,320,501	7,338,821	3,525,840	5,945,029

Table 8. Summary statistics for the minimum set of files (distinct token hashes).

		Java	C++	Python	JavaScript
Files per project	Min	1	1	1	1
	1^{st} Qu	3	3	2	1
	Median	7	11	5	3
	Mean	25.12	148.2	29	7
	3^{rd} Qu	19	38	12	7
	Max	66,734	77,730	36,840	53,168
Bytes per file	Min	1	1	0	1
	1^{st} Qu	876	1,149	694	586
	Median	1,984	2,622	1,896	1,605
	Mean	5,189	10,557	10,005	22,711
	3^{rd} Qu	4,513	6,812	5,084	4,683
	Max	80,344,320	100,095,279	104,749,580	576,196,992
Lines per file	Min	1	1	1	1
	1^{st} Qu	33	43	25	20
	Median	67	90	60	50
	Mean	149.3	301	182	441
	3^{rd} Qu	142	221	146	130
	Max	1,437,949	8,129,599	5,861,049	5,105,047
LOC per file	Min	1	1	1	1
	1^{st} Qu	27	35	20	16
	Median	56	74	48	42
	Mean	128.4	260	155	394
	3^{rd} Qu	120	185	119	112
	Max	1,436,850	8,129,570	5,860,092	5,105,045
SLOC per file	Min	0	0	0	0
	1^{st} Qu	18	24	16	13
	Median	41	54	40	35
	Mean	97.4	216	138	320
	3^{rd} Qu	93	145	102	95
	Max	1,436,841	8,129,521	5,859,657	5,105,045
Distinct tokens per file	Min	0	0	0	0
	1^{st} Qu	31	33	34	26
	Median	56	60	70	51
	Mean	84.9	132	144	219
	3^{rd} Qu	99	122	137	105
	Max	1,320,501	7,338,821	3,525,840	5,945,029

Table 9. Summary statistics for the minimum set of files (distinct file hashes).

		Java	C++	Python	JavaScript
Files per project	Min	1	1	1	1
	1^{st} Qu	3	3	2	1
	Median	8	11	5	3
	Mean	28	150	29	8
	3^{rd} Qu	20	38	13	7
	Max	74,144	78,106	37,009	53,168
Bytes per file	Min	0	0	0	0
	1^{st} Qu	899	1,149	662	583
	Median	2,019	2,709	1,838	1,637
	Mean	5,207	10,490	9,792	23,013
	3^{rd} Qu	4,563	6,925	5,008	4,964
	Max	80.3M	100M	105M	576M
Lines per file	Min	1	1	1	1
	1^{st} Qu	34	44	24	18
	Median	68	92	58	48
	Mean	150	299	180	454
	3^{rd} Qu	143	223	143	131
	Max	1,437,949	8,129,599	5,861,049	5,105,047
LOC per file	Min	1	1	1	0
	1^{st} Qu	27	36	19	15
	Median	56	76	47	41
	Mean	129	257	153	405
	3^{rd} Qu	121	187	118	113
	Max	1,436,850	8,129,570	5,860,092	5,105,045
SLOC per file	Min	0	0	0	0
	1^{st} Qu	18	24	15	12
	Median	41	54	39	33
	Mean	97	212	136	324
	3^{rd} Qu	92	145	100	95
	Max	1,436,841	8,129,521	5,859,657	5,105,045
Distinct tokens per file	Min	0	0	0	0
	1^{st} Qu	31	33	32	25
	Median	55	60	68	52
	Mean	84	130	142	218
	3^{rd} Qu	98	121	136	108
	Max	1,320,501	7,338,821	3,525,840	5,945,029

ACKNOWLEDGEMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement 695412), from the United States Defense Advanced Research Agency under the MUSE program, and was partially support by NSF award 1544542 and ONR award 503353.

REFERENCES

T. F. Bissyande, F. Thung, D. Lo, L. Jiang, and L. Reveillere. 2013. Orion: A Software Project Search Engine with Integrated Diverse Software Artifacts. In *International Conference on Engineering of Complex Computer Systems*. https://doi.org/10.1109/ICECCS.2013.42

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. https://doi.org/10.1145/1167473.1167488

Hudson Borges, André C. Hora, and Marco Tulio Valente. 2016. Understanding the Factors that Impact the Popularity of GitHub Repositories. (2016). http://arxiv.org/abs/1606.04984

Casey Casalnuovo, Prem Devanbu, Abilio Oliveira, Vladimir Filkov, and Baishakhi Ray. 2015. Assert Use in GitHub Projects. In *International Conference on Software Engineering (ICSE)*. http://dl.acm.org/citation.cfm?id=2818754.2818846

James R. Cordy, Thomas R. Dean, and Nikita Synytskyy. 2004. Practical Language-independent Detection of Near-miss Clones. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. http://dl.acm.org/citation.cfm?id=1034914.1034915

V. Cosentino, J. L. C. Izquierdo, and J. Cabot. 2016. Findings from GitHub: Methods, Datasets and Limitations. In *Working Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1109/MSR.2016.023

John W. Creswell. 2014. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE.

Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories. In *International Conference on Software Engineering (ICSE)*. http://dl.acm.org/citation.cfm?id=2486788.2486844

Jesus M. Gonzalez-Barahona, Gregorio Robles, and Santiago Dueñas. 2010. Collecting Data About FLOSS Development: The FLOSSMetrics Experience. In *International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS)*. https://doi.org/10.1145/1833272.1833278

Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Working Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1109/MSR.2013.6624034

Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck. 2011. *On the Extent and Nature of Software Reuse in Open Source Java Projects*. Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-21347-2_16

Felipe Hoffa. 2016. 400,000 GitHub repositories, 1 billion files, 14 terabytes of code: Spaces or Tabs? (2016). https://medium.com/@hoffa/400-000-github-repositories-1-billion-files-14-terabytes-of-code-spaces-or-tabs-7cfe0b5dd7fd

Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Working Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1145/2597073.2597074

Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.* 28, 7 (2002). https://doi.org/10.1109/TSE.2002.1019480

P. S. Kochhar, T. F. BissyandÃľ, D. Lo, and L. Jiang. 2013. Adoption of Software Testing in Open Source Projects–A Preliminary Study on 50,000 Projects. In *European Conference on Software Maintenance and Reengineering*. https://doi.org/10.1109/CSMR.2013.48

R. Koschke. 2007. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software (Dagstuhl Seminar Proceedings 06301)*.

A. Mockus. 2007. Large-Scale Code Reuse in Open Source Software. In *First International Workshop on Emerging Trends in FLOSS Research and Development*. https://doi.org/10.1109/FLOSS.2007.10

A. Mockus. 2009. Amassing and Indexing a Large Sample of Version Control Systems: Towards the Census of Public Source Code History. In *Working Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1109/MSR.2009.5069476

Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2013. Diversity in Software Engineering Research. In *Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/2491411.2491415

J. Ossher, Sushil Bajracharya, E. Linstead, P. Baldi, and Crista Lopes. 2009. SourcererDB: An aggregated repository of statically analyzed and cross-linked open source Java projects. In *Working Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1109/MSR.2009.5069501

Joel Ossher, Hitesh Sajnani, and Cristina Lopes. 2011. File Cloning in Open Source Java Projects: The Good, the Bad, and the Ugly. In *International Conference on Software Maintenance (ICSM)*. https://doi.org/10.1109/ICSM.2011.6080795

Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A Large Scale Study of Programming Languages and Code Quality in Github. In *International Symposium on Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/2635868.2635922

Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011. Automated Construction of JavaScript Benchmarks. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/2048066.2048119

C. K. Roy and J. R. Cordy. 2007. *A survey on software clone detection research*. Technical Report 541. Queens University.

Chanchal K. Roy and James R. Cordy. 2009. A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools. In *International Conference on Software Testing, Verification, and Validation*. https://doi.org/10.1109/ICSTW.2009.18

C. K. Roy and J. R. Cordy. 2010. Near-miss Function Clones in Open Source Software: An Empirical Study. *J. Softw. Maint. Evol.* 22, 3 (2010). https://doi.org/10.1002/smr.v22:3

Hitesh Sajnani. 2016. *Large-Scale Code Clone Detection*. Ph.D. Dissertation. University of California, Irvine.

Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-code. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/2884781.2884877

Johnny Saldaña. 2009. *The Coding Manual for Qualitative Researchers*. SAGE.

SPEC. 1998. SPECjvm98 benchmarks. (1998).

J. Svajlenko and C. K. Roy. 2015. Evaluating clone detection tools with BigCloneBench. In *International Conference on Software Maintenance and Evolution (ICSME)*. https://doi.org/10.1109/ICSM.2015.7332459

Christopher Vendome, Gabriele Bavota, Massimiliano Di Penta, Mario Linares-Vásquez, Daniel German, and Denys Poshyvanyk. 2016. License usage and changes: a large-scale study on GitHub. *Empirical Software Engineering* (2016). https://doi.org/10.1007/s10664-016-9438-4

4.2 Paper 2 - On the Impact of Programming Languages on Code Quality: A Reproduction Study

Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek.

In: ACM Trans. Program. Lang. Syst. 41, 4, Article 21 (October 2019), 24 pages. https://doi.org/10.1145/3340571

Due to the sensitivity of the topic, the paper's authors were presented in alphabetical order.

4.2.1 Author's Contributions

I was responsible for almost all aspects of the paper: the analysis of the original paper, the review and reproduction of the paper's artifact, data acquisition and reporting. I helped design the experiment needed for validation of the bug classifier used in the original paper and I prepared our artifact.

Celeste Hollenbeck (from Northeastern University) was responsible for executing the experiment and for the implemention of its web interface. Olga Vitek, a statistics professor from Northeastern University, provided the statistical insights and validated my work. Emery Berger from University of Massachusetts, Amherst, helped with the text of the paper.

4.2.2 Citations

1. Li, Z., Qi, X., Yu, Q., Liang, P., Mo, R. & Yang, C. Exploring multi-programming-language commits and their impacts on software quality: An empirical study on Apache projects. *Journal Of Systems And Software*. **194** (2022)

2. Li, W., Li, L. & Cai, H. On the vulnerability proneness of multilingual code. *ESEC/FSE 2022 - Proceedings Of The 30th ACM Joint Meeting European Software Engineering Conference And Symposium On The Foundations Of Software Engineering*. pp. 847-859 (2022)

3. Li, W., Li, L. & Cai, H. PolyFax: a toolkit for characterizing multi-language software. *ESEC/FSE 2022 - Proceedings Of The 30th ACM Joint Meeting European Software Engineering Conference And Symposium On The Foundations Of Software Engineering*. pp. 1662-1666 (2022)

4. Xu, R., Tang, Z., Ye, G., Wang, H., Ke, X., Fang, D. & Wang, Z. Detecting code vulnerabilities by learning from large-scale open source repositories. *Journal Of Information Security And Applications*. **69** (2022)

5. Khan, F., Chen, B., Varro, D. & McIntosh, S. An Empirical Study of Type-Related Defects in Python Projects. *IEEE Transactions On Software Engineering*. **48**, 3145-3158 (2022)

6. Furia, C., Torkar, R. & Feldt, R. Applying Bayesian Analysis Guidelines to Empirical Software Engineering Data: The Case of Programming Languages and Code Quality. *ACM Transactions On Software Engineering And Methodology*. **31** (2022)

7. Tornhill, A. & Borg, M. Code Red: The Business Impact of Code Quality - A Quantitative Study of 39 Proprietary Production Codebases. *Proceedings - International Conference On Technical Debt 2022, TechDebt 2022*. pp. 11-20 (2022)

8. Bogner, J. & Merkel, M. To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub. *Proceedings - 2022 Mining Software Repositories Conference, MSR 2022*. pp. 658-669 (2022)

9. Tian, Y., Zhang, Y., Stol, K., Jiang, L. & Liu, H. What Makes a Good Commit Message?. *Proceedings - International Conference On Software Engineering*. **2022-May** pp. 2389-2401 (2022)

10. Mao, K., Kapus, T., Petrou, L., Hajdu, A., Marescotti, M., Loscher, A., Harman, M. & Distefano, D. FAUSTA: Scaling Dynamic Analysis with Traffic Generation at WhatsApp. *Proceedings - 2022 IEEE 15th International Conference On Software Testing, Verification And Validation, ICST 2022*. pp. 267-278 (2022)

11. Klima, M., Bures, M., Frajtak, K., Rechtberger, V., Trnka, M., Bellekens, X., Cerny, T. & Ahmed, B. Selected Code-Quality Characteristics and Metrics for Internet of Things Systems. *IEEE Access*. **10** pp. 46144-46161 (2022)

12. Amit, I. & Feitelson, D. Corrective commit probability: a measure of the effort invested in bug fixing. *Software Quality Journal*. **29**, 817-861 (2021)

13. Zhang, J., Li, F., Hao, D., Wang, M., Tang, H., Zhang, L. & Harman, M. A Study of Bug Resolution Characteristics in Popular Programming Languages. *IEEE Transactions On Software Engineering*. **47**, 2684-2697 (2021)

14. Sztwiertnia, S., Grübel, M., Chouchane, A., Sokolowski, D., Narasimhan, K. & Mezini, M. Impact of programming languages on machine learning bugs. *AISTA 2021 - Proceedings Of The 1st ACM International Workshop On AI And Software Testing/Analysis, Co-located With ECOOP/ISSTA 2021*. pp. 9-12 (2021)

15. Vogel, A., Griebler, D. & Fernandes, L. Providing high-level self-adaptive abstractions for stream parallelism on multicores. *Software - Practice And Experience*. **51**, 1194-1217 (2021)

16. Babii, H., Prenner, J., Stricker, L., Karmakar, A., Janes, A. & Robbes, R. Mining Software Repositories with a Collaborative Heuristic Repository. *Proceedings - International Conference On Software Engineering*. pp. 106-110 (2021)

17. Li, Z., Qi, X., Yu, Q., Liang, P., Mo, R. & Yang, C. Multi-Programming-Language Commits in OSS: An Empirical Study on Apache Projects. *IEEE International Conference On Program Comprehension*. **2021-May** pp. 219-229 (2021)

18. Bonifro, F., Gabbrielli, M. & Zacchiroli, S. Content-Based Textual File Type Detection at Scale. *ACM International Conference Proceeding Series*. pp. 485-492 (2021)

19. Ruohonen, J. The Similarities of Software Vulnerabilities for Interpreted Programming Languages. *Proceedings Of The 2021 IEEE International Conference On Progress In Informatics And Computing, PIC 2021*. pp. 304-307 (2021)

20. Hermann, B., Winter, S. & Siegmund, J. Community expectations for research artifacts and evaluation processes. *ESEC/FSE 2020 - Proceedings Of The 28th ACM Joint Meeting European Software Engineering Conference And Symposium On The Foundations Of Software Engineering*. pp. 469-480 (2020)

21. Gonzalez, D., Zimmermann, T. & Nagappan, N. The State of the ML-universe: 10 Years of Artificial Intelligence & Machine Learning Software Development on GitHub. *Proceedings - 2020 IEEE/ACM 17th International Conference On Mining Software Repositories, MSR 2020.* pp. 431-442 (2020)

On the Impact of Programming Languages on Code Quality: A Reproduction Study

21

EMERY D. BERGER, University of Massachusetts Amherst and Microsoft Research
CELESTE HOLLENBECK, Northeastern University
PETR MAJ, Czech Technical University in Prague
OLGA VITEK, Northeastern University
JAN VITEK, Northeastern University and Czech Technical University in Prague

In a 2014 article, Ray, Posnett, Devanbu, and Filkov claimed to have uncovered a statistically significant association between 11 programming languages and software defects in 729 projects hosted on GitHub. Specifically, their work answered four research questions relating to software defects and programming languages. With data and code provided by the authors, the present article first attempts to conduct an experimental repetition of the original study. The repetition is only partially successful, due to missing code and issues with the classification of languages. The second part of this work focuses on their main claim, the association between bugs and languages, and performs a complete, independent reanalysis of the data and of the statistical modeling steps undertaken by Ray et al. in 2014. This reanalysis uncovers a number of serious flaws that reduce the number of languages with an association with defects down from 11 to only 4. Moreover, the practical effect size is exceedingly small. These results thus undermine the conclusions of the original study. Correcting the record is important, as many subsequent works have cited the 2014 article and have asserted, without evidence, a causal link between the choice of programming language for a given task and the number of software defects. Causation is not supported by the data at hand; and, in our opinion, even after fixing the methodological flaws we uncovered, too many unaccounted sources of bias remain to hope for a meaningful comparison of bug rates across languages.

CCS Concepts: • **General and reference** → **Empirical studies**; • **Software and its engineering** → **Software testing and debugging**;

Additional Key Words and Phrases: Programming Languages on Code Quality

ACM Reference format:
Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. 2019. On the Impact of Programming Languages on Code Quality: A Reproduction Study. *ACM Trans. Program. Lang. Syst.* 41, 4, Article 21 (October 2019), 24 pages.
https://doi.org/10.1145/3340571

This work received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412), the NSF (awards 1518844, 1544542, and 1617892), and the Czech Ministry of Education, Youth and Sports (grant agreement CZ.02.1.010.00.015_0030000421).
Authors' addresses: E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, and J. Vitek, Khoury College of Computer Sciences, Northeastern University, 440 Huntington Ave, Boston, MA 02115; emails: emery.berger@gmail.com, majpetr@fit.cvut.cz, celeste.hollenbeck@gmail.com, o.vitek@northeastern.edu, vitekj@me.com.

0164-0925/2019/10-ART21 $15.00
https://doi.org/10.1145/3340571

1 INTRODUCTION

At heart, a programming language embodies a bet: the bet that a given set of abstractions will increase developers' ability to deliver software that meets its requirements. Empirically quantifying the benefits of any set of language features over others presents methodological challenges. While one could have multiple teams of experienced programmers develop the same application in different languages, such experiments are too costly to be practical. Instead, when pressed to justify their choices, language designers often resort to intuitive arguments or proxies for productivity such as numbers of lines of code.

However, large-scale hosting services for code, such as GitHub or SourceForge, offer a glimpse into the lifecycles of software. Not only do they host the sources for millions of projects, but they also log changes to their code. It is tempting to use these data to mine for broad patterns across programming languages. The article we reproduce here is an influential attempt to develop a statistical model that relates various aspects of programming language design to software quality.

What is the effect of programming language on software quality? is the question at the heart of the study by Ray et al. published at the 2014 Foundations of Software Engineering (FSE) conference [26]. The work was sufficiently well regarded in the software engineering community to be nominated as a Communication of the ACM (CACM) *Research Highlight*. After another round of reviewing, a slightly edited version appeared in journal form in 2017 [25]. A subset of the authors also published a short version of the work as a book chapter [24]. The results reported in the FSE article and later repeated in the followup works are based on an observational study of a corpus of 729 GitHub projects written in 17 programming languages. To measure quality of code, the authors identified, annotated, and tallied commits that were deemed to indicate bug fixes. The authors then fit a Negative Binomial regression against the labeled data, which was used to answer the following four research questions:

RQ1 "**Some languages have a greater association with defects than others**, although the effect is small." Languages associated with fewer bugs were TypeScript, Clojure, Haskell, Ruby, and Scala; while C, C++, Objective-C, JavaScript, PHP, and Python were associated with more bugs.

RQ2 "**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."

RQ3 "**There is no general relationship between domain and language defect proneness.**" Thus, application domains are less important to software defects than languages.

RQ4 "**Defect types are strongly associated with languages.** Some defect types like memory errors and concurrency errors also depend on language primitives. Language matters more for specific categories than it does for defects overall."

Of these four results, it is the first two that garnered the most attention both in print and on social media. This is likely the case, because those results confirmed commonly held beliefs about the benefits of static type systems and the need to limit the use of side effects in programming.

Correlation is not causality, but it is tempting to confuse them. The original study couched its results in terms of *associations* (i.e., correlations) rather than *effects* (i.e., *causality*) and carefully qualified effect size. Unfortunately, many of the article's readers were not as careful. The work was taken by many as a statement on the impact of programming languages on defects. Thus, one can find citations such as:

- "*…They found language design did have a significant, but modest effect on software quality*" [23].

Table 1. Citation Analysis

	Cites	Self
Cursory	77	1
Methods	12	0
	Cites	Self
Correlation	2	2
Causation	24	3

- "*... The results indicate that strong languages have better code quality than weak languages*" [31].
- "*... functional languages have an advantage over procedural languages*" [21].

Table 1 summarizes our citation analysis. Of the 119 articles that were retrieved,[1] 90 citations were either passing references (Cursory) or discussed the methodology of the original study (Methods). Of the citations that discussed the results, 4 were careful to talk about associations (i.e., correlation), while 26 used language that indicated effects (i.e., causation). It is particularly interesting to observe that even the original authors, when they cite their own work, sometimes resort to causal language. For example, Ray and Posnett write, "Based on our previous study [26] we found that the overall effect of language on code quality is rather modest" [24]; Devanbu writes, "We found that static typing is somewhat better than dynamic typing, strong typing is better than weak typing, and built-in memory management is better" [5]; and "Ray [...] said in an interview that functional languages were boosted by their reliance on being mathematical and the likelihood that more experienced programmers use them" [15]. Section 2 of the present article gives a detailed account of the original study and its conclusions.

Given the controversy generated by the CACM paper on social media, and some surprising observations in the text of the original study (e.g., that Chrome V8 is their largest JavaScript project—when the virtual machine is written in C++), we wanted to gain a better understanding of the exact nature of the scientific claims made in the study and how broadly they are actually applicable. To this end, we chose to conduct an independent reproduction study.

A reproduction study aims to answer the question *can we trust the papers we cite?* Over a decade ago, following a spate of refutations, Ioannidis argued that most research findings are false [13]. His reasoning factored in small effect sizes, limited number of experiments, misunderstanding of statistics, and pressure to publish. While refutations in computer science are rare, there are worrisome signs. Kalibera et al. reported that 39 of 42 PLDI 2011 papers failed to report any uncertainty in measurements [29]. Reyes et al. catalogued statistical errors in 30% of the empirical papers published at ICSE [27] from 2006 to 2015. Other examples include the critical review of patch generation research by Monperrus [20] and the assessment of experimental fuzzing evaluations by Klees et al. [14]. To improve the situation, our best bet is to encourage a culture of *reproducible research* [8]. Reproduction increases our confidence: an experimental result reproduced independently by multiple authors is more likely to be valid than the outcome of a single study. Initiatives such as SIGPLAN and SIGSOFT's artifact evaluation process, which started at FSE and spread widely [16], are part of a move toward increased reproducibility.

Methodology. Reproducibility of results is not a binary proposition. Instead, it spans a spectrum of objectives that provide assurances of different kinds (see Figure 1 using terms from References [9, 29]).

[1]Retrieval performed on 12/01/18 based on the Google Scholar citations of the FSE article; duplicates were removed.

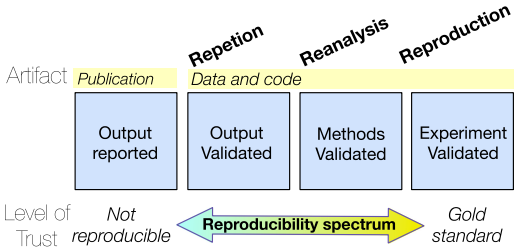


Fig. 1. Reproducibility spectrum (from Reference [22]).

Experimental repetition aims to replicate the results of some previous work with the same data and methods and should yield the same numeric results. Repetition is the basic guarantee provided by artifact evaluation [16]. *Reanalysis* examines the robustness of the conclusions to the methodological choices. Multiple analysis methods may be appropriate for a given dataset, and the conclusions should be robust to the choice of method. Occasionally, small errors may need to be fixed, but the broad conclusions should hold. Finally, *Reproduction* is the gold standard; it implies a full-fledged independent experiment conducted with different data and the same or different methods. To avoid bias, repetition, reanalysis, and reproduction are conducted independently. The only contact expected with the original authors is to request their data and code.

Results. We began with an experimental repetition, conducting it in a similar fashion to a conference artifact evaluation [16] (Section 3 of this article). Intuitively, a repetition should simply be a matter of running the code provided by the authors on the original data. Unfortunately, things often do not work out so smoothly. The repetition was only partially successful. We were able to mostly replicate RQ1 based on the artifact provided by the authors. We found 10 languages with a statistically significant association with errors, instead of the 11 reported. For RQ2, we uncovered classification errors that made our results depart from the published ones. In other words, while we could repeat the original, its results were meaningless. Last, RQ3 and RQ4 could not be repeated due to missing code and discrepancies in the data.

For *reanalysis*, we focused on RQ1 and discovered significant methodological flaws (Section 4 of this article). While the original study found that 11 of 17 languages were correlated with a higher or lower number of defective commits, upon cleaning and reanalyzing the data, the number of languages dropped to 7. Investigations of the original statistical modeling revealed technical oversights such as inappropriate handling of multiple hypothesis testing. Finally, we enlisted the help of independent developers to cross-check the original method of labeling defective commits, which led us to estimate a false-positive rate of 36% on buggy commit labels. Combining corrections for all of these aforementioned items, the reanalysis revealed that only 4 of the original 11 languages correlated with abnormal defect rates, and even for those the effect size is exceedingly small.

Figure 2 summarizes our results: Not only is it not possible to establish a causal link between programming language and code quality based on the data at hand, but even their correlation proves questionable. Our analysis is repeatable and available in an artifact hosted at: https://github.com/PRL-PRG/TOPLAS19_Artifact.

Follow up work. While reanalysis was not able to validate the results of the original study, we stopped short of conducting a reproduction as it is unclear what that would yield. In fact, even if we were to obtain clean data and use the proper statistical methods, more research is needed to understand all the various sources of bias that may affect the outcomes. Section 5 lists some challenges that we discovered while doing our repetition. For instance, the ages of the projects

Repetition

RQ1	RQ2	RQ3	RQ4
✓	✗	✗	✗

Reanalysis (RQ1)

	FSE'14	This paper
positive association	C++ Objective-C C PHP Python JavaScript	C++
negative association	TypeScript Clojure Scala Haskell Ruby	Clojure Haskell Ruby

Reanalysis (RQ1)

	FSE'14	This paper
no statistically significant association	C# Go CoffeeScript Java Perl Erlang	C C# Objective-C Go JavaScript CoffeeScript Java Perl PHP Python Erlang Scala TypeScript

Fig. 2. Result summary.

vary across languages (older languages such as C are dominated by mature projects such as Linux), and the data include substantial numbers of commits to test files (how bugs in tests are affected by language characteristics is an interesting question for future research). We believe that there is a need for future research on this topic; we thus conclude our article with some best practice recommendations for future researchers (Section 6).

2 ORIGINAL STUDY AND ITS CONCLUSIONS

2.1 Overview

The FSE paper by Ray et al. [26] aimed to explore associations between languages, paradigms, application domains, and software defects from a real-world ecosystem across multiple years. Its multi-step, mixed-method approach included collecting commit information from GitHub; identifying each commit associated with a bug correction; and using Negative Binomial Regression (NBR) to analyze the prevalence of bugs. The paper claims to answer the following questions.

RQ1. *Are some languages more defect prone than others?*

The paper concluded that "*Some languages have a greater association with defects than others, although the effect is small.*" Results appear in a table that fits an NBR model to the data; it reports coefficient estimates, their standard errors, and ranges of p-values. The authors noted that confounders other than languages explained most of the variation in the number of bug-fixing commits, quantified by analysis of deviance. They reported p-values below .05, .01, and .001 as "statistically significant." Based on these associations, readers may be tempted to conclude that TypeScript, Haskell, Clojure, Ruby, and Scala were less error prone; and C++, Objective-C, C, JavaScript, PHP, and Python were more error prone. Of course, this would be incorrect as association is not causation.

RQ2. *Which language properties relate to defects?*

The study concluded that "*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.*" The impact of nine language categories across four classes was assessed. Since the categories were highly correlated (and thus compromised the stability of the NBR), the paper modeled aggregations of the languages by class. The regression included the same confounders as

in RQ1 and represented language classes. The authors report the coefficients, their standard errors, and ranges of p-values. These results may lead readers to conclude that functional, strongly typed languages induced fewer errors, while procedural, weakly typed, unmanaged languages induced more errors.

RQ3. *Does language defect proneness depend on domain?*

The study used a mix of automatic and manual methods to classify projects into six application domains. After removing outliers, and calculating the Spearman correlation between the order of languages by bug ratio within domains against the order of languages by bug ratio for all domains, it concluded that "*There is no general relationship between domain and language defect proneness.*" The paper states that all domains show significant positive correlation, except the Database domain. From this, readers might conclude that the variation in defect proneness comes from the languages themselves, making domain a less indicative factor.

RQ4. *What's the relation between language & bug category?*

The study concluded that "*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.*" The authors report that 88% of the errors fall under the general Programming category, for which results are similar to RQ1. Memory Errors account for 5% of the bugs, Concurrency for 2%, and Security and other impact errors for 7%. For Memory, languages with manual memory management have more errors. Java stands out; it is the only garbage collected language associated with more memory errors. For Concurrency, inherently single-threaded languages (Python, JavaScript, ...) have fewer errors than languages with concurrency primitives. The causal relation for Memory and Concurrency is understandable, as the classes of errors require particular language features.

2.2 Methods in the Original Study

Below, we summarize the process of data analysis by the original manuscript while splitting it into the following three phases: data acquisition, cleaning, and modeling.

2.2.1 Data Acquisition. For each of the 17 languages with the most projects on GitHub, 50 projects with the highest star rankings were selected. Any project with fewer than 28 commits was filtered out, leaving 729 projects (86%). For each project, commit histories were collected with `git log --no-merges --numstat`. The data were split into rows, such that each row had a unique combination of file name, project name, and commit identifier. Other fields included committer and author name, date of the commit, commit message, and number of lines inserted and deleted. In summary, the original paper states that the input consisted of 729 projects written in 17 languages, accounting for 63 million SLOC created over 1.5 million commits written by 29,000 authors. Of these, 566,000 commits were bug fixes.

2.2.2 Data Cleaning. As any project may be written in multiple languages, each row of the data is labeled by language based on the file's extension (TypeScript is `.ts`, and so on). To rule out small change sets, projects with fewer than 20 commits in any single language are filtered out for that language. Commits are labeled as bug fixes by searching for error-related keywords: *error, bug, fix, issue, mistake, incorrect, fault, defect*, and *flaw* in commit messages. This is similar to a heuristic introduced by Mockus and Votta [19]. Each row of the data is furthermore labeled with four extra attributes. The Paradigm class is either procedural, functional, or scripting. The

Compile class indicates whether a language is statically or dynamically typed. The Type class indicates whether a language admits "type-confusion," i.e., it allows interpreting a memory region populated by a value of one type as another type. A language is strongly typed if it explicitly detects type confusion and reports it as such. The Memory class indicates whether the language requires developers to manage memory by hand.

2.2.3 Statistical Modeling. For RQ1, the manuscript specified an NBR [7], where an observation is a combination of project and language. In other words, a project written in three languages has three observations. For each observation, the regression uses bug-fixing commits as a response variable, and the languages as the independent variables. NBR is an appropriate choice, given the non-negative and discrete nature of the counts of commits. To adjust for differences between the observations, the regression includes the confounders age, number of commits, number of developers, and size (represented by inserted lines in commits), all log-transformed to improve the quality of fit. For the purposes of RQ1, the model for an observation i is as follows:

$$\text{bcommits}_i \sim \mathit{NegativeBinomial}(\mu_i, \theta), \text{ where}$$
$$\text{E}\{\text{bcommits}_i\} = \mu_i$$
$$\text{Var}\{\text{bcommits}_i\} = \mu_i + \mu_i^2/\theta$$
$$log\,\mu_i = \beta_0 + \beta_1 \log(\text{commits})_i + \beta_2 \log(\text{age})_i + \beta_3 \log(\text{size})_i + \beta_4 \log(\text{devs})_i + \sum_{j=1}^{16} \beta_{(4+j)} \text{language}_{ij}$$

The programming languages are coded with *weighted contrasts*. These contrasts are customized in a way to interpret β_0 as the average log-expected number of bugs in the dataset. Therefore, $\beta_5, \ldots, \beta_{20}$ are the deviations of the log-expected number of bug-fixing commits in a language from the average of the log-expected number of bug-fixing commits. Finally, the coefficient β_{21} (corresponding to the last language in alphanumeric order) is derived from the contrasts after the model fit [17]. Coefficients with a statistically significant negative value indicate a lower expected number of bug-fixing commits; coefficients with a significant positive value indicate a higher expected number of bug-fixing commits. The model-based inference of parameters $\beta_5, \ldots, \beta_{21}$ is the main focus of RQ1.

For RQ2, the study fit another NBR, with the same confounder variables, to study the association between language classes and the number of bug-fixing commits. It then uses Analysis of Deviance to quantify the variation attributed to language classes and the confounders. For RQ3, the article calculates the Spearman's correlation coefficient between defectiveness by domain and defectiveness overall, with respect to language, to discuss the association between languages versus that by domain. For RQ4, the study once again uses NBR, with the same confounders, to explore the propensity for bugfixes among the languages with regard to bug types.

3 EXPERIMENTAL REPETITION

Our first objective is to repeat the analyses of the FSE article and to obtain the same results. We requested and received from the original authors an artifact containing 3.45GB of processed data and 696 lines of R code to load the data and perform statistical modeling steps.

3.1 Methods

Ideally, a repetition should be a simple process, where a script generates results and these match the results in the published article. In our case, we only had part of the code needed to generate the expected tables and no code for graphs. We therefore wrote new R scripts to mimic all of the steps, as described in the original manuscript. We found it essential to automate the production of all numbers, tables, and graphs shown in our article as we had to iterate multiple times. The

code for repetition amounts to 1,140 lines of R (file `repetition.Rmd` and `implementation.R` in our artifact).

3.2 Results

The data were provided to us in the form of two CSV files. The first, larger file contained one row per file and commit, and it contained the bug fix labels. The second, smaller file aggregated rows with the same commit and the same language. Upon preliminary inspection, we observed that the files contained information on 729 projects and 1.5 million commits. We found an additional 148 projects that were omitted from the original study without explanation. We choose to ignore those projects as data volume is not an issue here.

Developers vs. Committers. One discrepancy was the 47,000 authors we observed versus the 29,000 reported. This is explained by the fact that, although the FSE article claimed to use *developers* as a control variable, it was in fact counting *committers*: a subset of developers with commit rights. For instance, Linus Torvalds has 73,038 commits, of which he personally authored 11,343, the remaining are due to other members of the project. The rationale for using developers as a control variable is that the same individual may be more or less prone to committing bugs, but this argument does not hold for committers as they aggregate the work of multiple developers. We chose to retain committers for our reproduction but note that this choice should be revisited in follow up work.

Measuring code size. The commits represented 80.7 million lines of code. We could not account for a difference of 17 million SLOC from the reported size. We also remark, but do not act on, the fact that project size, computed in the FSE article as the sum of inserted lines, is not accurate—as it does not take deletions into account. We tried to subtract deleted lines and obtained projects with negative line counts. This is due to the treatments of Git merges. A merge is a commit that combines conflicting changes of two parent commits. Merge commits are not present in our data; only parent commits are used, as they have more meaningful messages. If both parent commits of a merge delete the same lines, then the deletions are double counted. It is unclear what the right metric of size should be.

3.2.1 Are Some Languages More Defect Prone Than Others (RQ1). We were able to qualitatively (although not exactly) repeat the result of RQ1. Table 2(a) has the original results, and (c) has our repetition. Grey cells indicate disagreement with the conclusion of the original work. One disagreement in our repetition is with PHP. The FSE paper reported a p-value <.001, while we observed <.01; per their established threshold of .005, the association of PHP with defects is not statistically significant. The original authors corrected that value in their CACM repetition (shown in Table 2(b)), so this may just be a reporting error. However, the CACM article dropped the significance of JavaScript and TypeScript without explanation. The other difference is in the coefficients for the control variables. Upon inspection of the code, we noticed that the original manuscript used a combination of log and log10 transformations of these variables, while the repetition consistently used log. The author's CACM repetition fixed this problem.

3.2.2 Which Language Properties Relate to Defects (RQ2). As we approached RQ2, we faced an issue with the language categorization used in the FSE paper. The original categorization is reprinted in Table 3. The intuition is that each category should group languages that have "similar" characteristics along some axis of language design.

The first thing to observe is that any such categorization will have some unclear fits. The original authors admitted as much by excluding TypeScript from this table, as it was not obvious whether a gradually typed language is static or dynamic. But there were other odd ducks. Scala is categorized

Table 2. Negative Binomial Regression for Languages (Gray Indicates Disagreement with the Conclusion of the Original Work)

	Original Authors				Repetition	
	(a) FSE [26]		(b) CACM [25]		(c)	
	Coef	P-val	Coef	P-val	Coef	P-val
Intercept	− 1.93	<0.001	− 2.04	<0.001	− 1.8	<0.001
log commits	2.26	<0.001	0.96	<0.001	0.97	<0.001
log age	0.11	<0.01	0.06	<0.001	0.03	0.03
log size	0.05	<0.05	0.04	<0.001	0.02	<0.05
log devs	0.16	<0.001	0.06	<0.001	0.07	<0.001
C	0.15	<0.001	0.11	<0.01	0.16	<0.001
C++	0.23	<0.001	0.18	<0.001	0.22	<0.001
C#	0.03	–	− 0.02	–	0.03	0.602
Objective-C	0.18	<0.001	0.15	<0.01	0.17	0.001
Go	− 0.08	–	− 0.11	–	− 0.11	0.086
Java	− 0.01	–	− 0.06	–	− 0.02	0.61
Coffeescript	− 0.07	–	0.06	–	0.05	0.325
Javascript	0.06	<0.01	0.03	–	0.07	<0.01
Typescript	− 0.43	<0.001	0.15	–	− 0.41	<0.001
Ruby	− 0.15	<0.05	− 0.13	<0.01	− 0.13	<0.05
Php	0.15	<0.001	0.1	<0.05	0.13	0.009
Python	0.1	<0.01	0.08	<0.05	0.1	<0.01
Perl	− 0.15	–	− 0.12	–	− 0.11	0.218
Clojure	− 0.29	<0.001	− 0.3	<0.001	− 0.31	<0.001
Erlang	0	–	− 0.03	–	0	1
Haskell	− 0.23	<0.001	− 0.26	<0.001	− 0.24	<0.001
Scala	− 0.28	<0.001	− 0.24	<0.001	− 0.22	<0.001

Table 3. Language Classes Defined by the FSE Paper

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

as a functional language, yet it allows programs to be written in an imperative manner. We are not aware of any study that shows that the majority of Scala users write functional code. Our experience with Scala is that users freely mix functional and imperative programming. Objective-C is listed as a statically compiled and unmanaged language. However, Objective-C has an object system that is inspired by SmallTalk; its treatment of objects is quite dynamic, and objects are collected by reference counting, so its memory is partially managed. The Type category is the most

Table 4. Negative Binomial Regression for Language Classes

	(a) Original		(b) Repetition		(c) Reclassification	
	Coef	P-val	Coef	P-val	Coef	P-val
Intercept	−2.13	<0.001	−2.14	<0.001	−1.85	<0.001
log age	0.07	<0.001	0.15	<0.001	0.05	0.003
log size	0.05	<0.001	0.05	<0.001	0.01	0.552
log devs	0.07	<0.001	0.15	<0.001	0.07	<0.001
log commits	0.96	<0.001	2.19	<0.001	1	<0.001
Fun Sta Str Man	−0.25	<0.001	−0.25	<0.001	−0.27	<0.001
Pro Sta Str Man	−0.06	<0.05	−0.06	0.039	−0.03	0.24
Pro Sta Wea Unm	0.14	<0.001	0.14	<0.001	0.19	0
Scr Dyn Wea Man	0.04	<0.05	0.04	0.018	0	0.86
Fun Dyn Str Man	−0.17	<0.001	−0.17	<0.001	–	–
Scr Dyn Str Man	0.001	–	0	0.906	–	–
Fun Dyn Wea Man	–	–	–	–	−0.18	<0.001

Language classes are combined procedural (Pro), functional (Fun), scripting (Scr), dynamic (Dyn), static (Sta), strong (Str), weak (Wea), managed (Man), and unmanaged (Unm). Rows marked – have no observation.

counter-intuitive for programming language experts as it expresses whether a language allows value of one type to be interpreted as another, e.g., due to automatic conversion. The CACM paper attempted to clarify this definition with the example of the `ID` type. In Objective-C, an `ID` variable can hold any value. If this is what the authors intend, then Python, Ruby, Clojure, and Erlang would be weak as they have similar generic types.

In our repetition, we modified the categories accordingly and introduced a new category of Functional-Dynamic-Weak-Managed to accommodate Clojure and Erlang. Table 4(c) summarizes the results with the new categorization. The reclassification (using zero-sum contrasts introduced in Section 4.2.1) disagrees on the significance of 2 of 5 categories. We note that we could repeat the results of the original classification, but since that classification is wrong, those results are not meaningful.

3.2.3 Does Language Defect Proneness Depend on Domain (RQ3). We were unable to repeat RQ3, as the artifact did not include code to compute the results. In a repetition, one expects the code to be available. However, the data contained the classification of projects in domains, which allowed us to attempt to recreate part of the analysis described in the paper. While we successfully replicated the initial analysis step, we could not match the removal of outliers described in the FSE paper. Stepping outside of the repetition, we explore an alternative approach to answer the question. Table 5 uses an NBR with domains instead of languages. The results suggest there is no evidence that the application domain is a predictor of bug-fixes as the paper claims. So, while we cannot repeat the result, the conclusion likely holds.

3.2.4 What Is the Relation Between Language and Bug Category (RQ4). We were unable to repeat the results of RQ4, because the artifact did not contain the code that implemented the heatmap or NBR for bug types. Additionally, we found no single column in the data that contained the bug categories reported in the FSE paper. It was further unclear whether the bug types were disjoint: adding together all of the percentages for every bug type mentioned in Table 5 of the FSE study totaled 104%. The input CSV file did contain two columns that, when combined, matched these categories. When we attempted to reconstruct the categories and compared counts of each bug

Table 5. NBR for RQ3

	Coef	p-Val
(Intercept)	−1.94	<0.001
log age	0.05	<0.001
log size	0.03	<0.001
log devs	0.08	<0.001
log commits	0.96	<0.001

	Coef	p-Val
Application	0	1.00
CodeAnalyzer	−0.05	0.93
Database	0.04	1.00
Framework	0.01	1.00
Library	−0.06	0.23
Middleware	0	1.00

type, we found discrepancies with those originally reported. For example, we had 9 times as many Unknown bugs as the original, but we had only less than half the number of Memory bugs. Such discrepancies make repetition invalid.

3.3 Outcome

The repetition was partly successful. RQ1 produced small differences, but qualitatively similar conclusions. RQ2 could be repeated, but we noted issues with language classification; fixing these issues changed the outcome for 2 of 5 categories. RQ3 could not be repeated, as the code was missing and our reverse engineering attempts failed. RQ4 could not be repeated due to irreconcilable differences in the data.

4 REANALYSIS

Our second objective is to carry out a reanalysis of RQ1 of the FSE article. The reanalysis differs from repetition in that it proposes alternative data processing and statistical analyses to address what we identify as methodological weaknesses of the original work.

4.1 Methods: Data Processing

First, we examined more closely the process of data acquisition in the original work. This step was intended as a quality control, and it did not result in changes to the data.

We wrote software to automatically download and check commits of projects against GitHub histories. Out of 729 projects used in the FSE paper, 618 could be downloaded. The other projects may have been deleted or became private. The downloaded projects were matched by name. As the FSE data lacked project owner names, the matches were ambiguous. By checking for matching SHAs, we confidently identified 423 projects as belonging to the study. For each matched project, we compared its entire history of commits to its commits in the FSE dataset, as follows. We identified the most recent commit c occurring in both. Commits chronologically older than c were classified as either *valid* (appearing in the original study), *irrelevant* (not affecting language files), or *missing* (not appearing in the original study).

We found 106K missing commits (i.e., 19.95% of the dataset). Perl stands out with 80% of commits that were missing in the original manuscript (Figure 3 lists the ratio of missing commits per language). Manual inspection of a random sample of the missing commits did not reveal any pattern. We also recorded *invalid* commits (occurring in the study but absent from the GitHub history). Four projects had substantial numbers of invalid commits, likely due to matching errors or a change in commit history (such as with the `git rebase` command).

Next, we applied three data cleaning steps (see below for details; each of these was necessary to compensate for errors in data acquisition of the original study): (1) *Deduplication*, (2) *Removal*

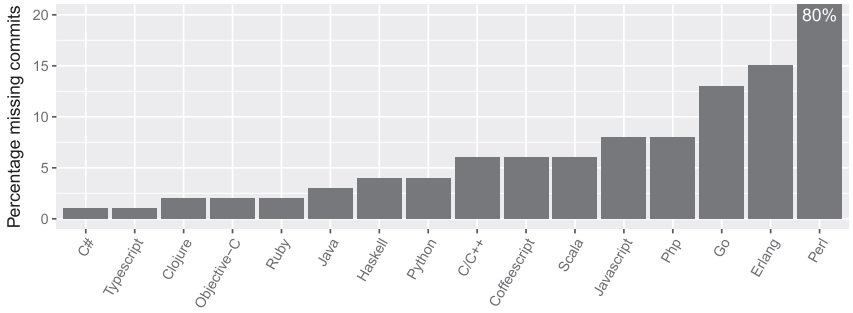


Fig. 3. Percentage of commits identified as missing from the FSE dataset.

of TypeScript, (3) *Accounting for C and C++*. Our implementation consists of 1,323 lines of R code split between files `re-analysis.Rmd` and `implementation.R` in the artifact.

4.1.1 Deduplication. While the input data did not include forks, we checked for project similarities by searching for projects with similar commit identifiers. We found 33 projects that shared one or more commits. Of those, 18 were related to `bitcoin`, a popular project that was frequently copied and modified. The projects with duplicate commits are as follows: litecoin, megacoin, memorycoin, bitcoin, bitcoin-qt-i2p, anoncoin, smallchange, primecoin, terracoin, zetacoin, datacoin, datacoin-hp, freicoin, ppcoin, namecoin, namecoin-qt, namecoinq, ProtoShares, QGIS, Quantum-GIS, incubator-spark, spark, sbt, xsbt, Play20, playframework, ravendb, SignalR, Newtonsoft.Json, Hystrix, RxJava, clojure-scheme, and clojurescript. In total, there were 27,450 duplicated commits, or 1.86% of all commits. We deleted these commits from our dataset to avoid double counting some bugs.

4.1.2 Removal of TypeScript. In the original dataset, the first commit for TypeScript was recorded on 2003-03-21, several years before the language was created. Upon inspection, we found that the file extension `.ts` is used for XML files containing human language translations. Of 41 projects labeled as TypeScript, only 16 contained TypeScript. This reduced the number of commits from 10,063 to an even smaller 3,782. Unfortunately, the three largest remaining projects (`typescript-node-definitions`, `DefinitelyTyped`, and the deprecated `tsd`) contained only declarations and no code. They accounted for 34.6% of the remaining TypeScript commits. Given the small size of the remaining corpus, we removed it from consideration as it is not clear that we have sufficient data to draw useful conclusions. To understand the origin of the classification error, we checked the tool mentioned in the FSE article, GitHub Linguist.[2] At the time of the original study, that version of Linguist incorrectly classified translation files as TypeScript. This was fixed on December 6, 2014. This may explain why the number of TypeScript projects decreased between the FSE and CACM articles.

4.1.3 Accounting for C++ and C. Further investigation revealed that the input data only included C++ commits to files with the .cpp extension. However, C++ compilers allow many extensions, including .C, .cc, .CPP, .c++, .cp, and .cxx. Moreover, the dataset contained no commits to .h header files. However, these files regularly contain executable code such as inline functions in C and templates in C++. We could not repair this without getting additional data and writing a tool

[2] https://github.com/github/linguist.

	Commits
C	16
C++	7
Python	488
JavaScript	2,907

Fig. 4. V8 commits.

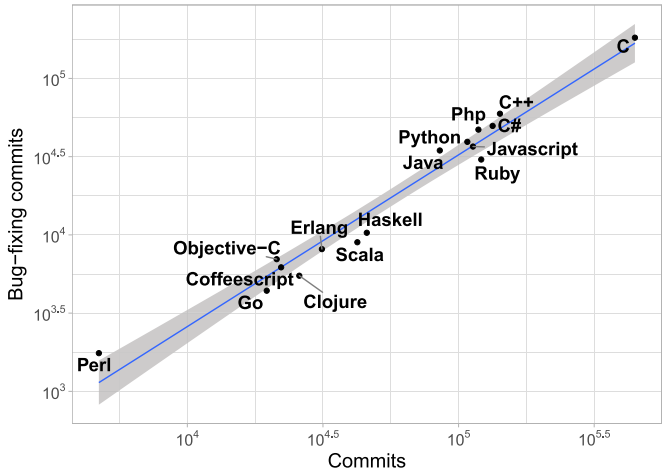


Fig. 5. Commits and bug-fixing commits after cleaning, plotted with a 95% confidence interval.

to label the commits in the same way as the authors did. We checked GitHub Linguist to explain the missing files, but as of 2014, it was able to recognize header files and all C++ extensions.

The only correction we applied was to delete the V8 project. While V8 is written mostly in C++, its commits in the dataset are mostly in JavaScript (Figure 4 gives the number of commits per language in the dataset for the V8 project). Manual inspection revealed that JavaScript commits were regression test cases for errors in the missing C++ code. Including them would artificially increase the number of JavaScript errors. The original authors may have noticed a discrepancy as they removed V8 from RQ3.

At the end of the data cleaning steps, the dataset had 708 projects, 58.2 million lines of code, and 1.4 million commits—of which 517,770 were labeled as bug-fixing commits, written by 46 thousand authors. Overall, our cleaning reduced the corpus by 6.14%. Figure 5 shows the relationship between commits and bug fixes in all of the languages after the cleaning. As one would expect, the number of bug-fixing commits correlated to the number of commits. The figure also shows that the majority of commits in the corpus came from C and C++. Perl is an outlier, because most of its commits were missing from the corpus.

4.1.4 Labeling Accuracy. A key reanalysis question for this case study is as follows: *What is a bug-fixing commit?* With the help of 10 independent developers employed in industry, we compared the manual labels of randomly selected commits to those obtained automatically in the FSE paper. We selected a random subset of 400 commits via the following protocol. First, randomly sample 20 projects. In these projects, randomly sample 10 commits labeled as bug-fixing and 10 commits not labeled as bug-fixing. Enlisting help from 10 independent developers employed in industry, we omitted the commits' bugfix labels and divided them equally among the ten experts.

Each commit was manually given a new binary bugfix label by 3 of the experts, according to their best judgment. Commits with at least 2 bugfix votes were considered to be bug fixes. The review suggested a false-positive rate of 36%; i.e., 36% of the commits that the original study considered as bug-fixing were in fact not. The false-negative rate was 11%. Short of relabeling the entire dataset manually, there was nothing we could do to improve the labeling accuracy. Therefore, we chose an alternative route and took labeling inaccuracy into account as part of the statistical modeling and analysis.

We give five examples of commits that were labeled as bug fixing in the FSE paper but were deemed by developers not to be bug fixes. Each line contains the text of the commit, underlined emphasis is ours and indicates the likely reason the commit was labeled as a bug fix (when apparent), and the URL points to the commit in GitHub:

- `tabs to spaces formatting fixes.`
  https://langstudy.page.link/gM7N
- `better error messages.`
  https://langstudy.page.link/XktS
- `Converted CoreDataRecipes sample to MagicalRecordRecipes sample application.`
  https://langstudy.page.link/iNhr
- `[core] Add NIError.h/m.`
  https://langstudy.page.link/n7Yf
- `Add lazyness to infix operators.`
  https://langstudy.page.link/2qPk

Unanimous mislabelings (when all three developers agreed) constituted 54% of the false positives. To control for random interrater agreement, we compute Cohen's Kappa coefficient. We calculate kappa coefficients for all pairs of raters on the subset of commits they both reviewed. All values were positive with a median of 0.6. Within the false positives, most of the mislabeling arose, because words that were synonymous with or related to bugs (e.g., "fix" and "error") were found within substrings or matched completely out of context. A meta-analysis of the false positives suggests the following six categories:

(1) *Substrings*;
(2) *Non-functional:* meaning-preserving refactoring, e.g., changes to variable names;
(3) *Comments:* changes to comments, formatting, and so on;
(4) *Feature:* feature enhancements;
(5) *Mismatch*: keywords used in an unambiguous non-bug context (e.g., "this is not a bug");
(6) *Hidden features:* new features with unclear commit messages.

The original study clarified that its classification, which involved identifying bugfixes by only searching for error-related keywords came from Reference [19]. However, that work classified modification requests with an iterative, multi-step process, which differentiates between six different types of code changes through multiple keywords. It is possible that this process was planned but not completed in the FSE publication.

It is noteworthy that the above concerned are well known in the software engineering community. Since the Mockus and Votta paper [19], a number of authors have observed that using keywords appearing in commit message is error prone, and that biased error messages can lead to erroneous conclusions [2, 12, 28] (Reference [2] has amongst its authors two of the authors of FSE'14). Yet, keyword based bug-fix detection is still a common practice [3, 6].

4.2 Methods: Statistical Modeling

The reanalysis uncovered several methodological weaknesses in the statistical analyses of the original manuscript.

4.2.1 Zero-sum Contrasts. The original manuscript chose to code the programming languages with weighted contrasts. Such contrasts interpret the coefficients of the Negative Binomial Regression as deviations of the log-expected number of bug-fixing commits in a language from the average of the log-expected number of bug-fixing commits *in the dataset*. Comparison to the dataset average is sensitive to changes in the dataset composition, makes the reference unstable, and compromises the interpretability of the results. This is particularly important when the composition of the dataset is subject to uncertainty, as discussed in Section 4.1 above. A more common choice is to code factors such as programming languages with zero-sum contrasts [17]. This coding interprets the parameters as the deviations of the log-expected number of bug-fixing commits in a language from the average of log-expected number of bug-fixing commits *between the languages*. It is more appropriate for this investigation.

4.2.2 Multiplicity of Hypothesis Testing. A common mistake in data-driven software engineering is to fail to account for multiple hypothesis testing [27]. When simultaneously testing multiple hypotheses, some p-values can fall in the significance range by random chance. This is certainly true for Negative Binomial Regression, when we simultaneously test 16 hypotheses of coefficients associated with 16 programming languages being 0 [17]. Comparing 16 independent p-values to a significance cutoff of, say, 0.05 in absence of the associations implies the family-wise error rate (i.e., the probability of at least one false-positive association) FWER $= 1 - (1 - 0.05)^{16} = 0.56$. The simplest approach to control FWER is the method of Bonferroni, which compares the p-values to the significance cutoff divided by the number of hypotheses. Therefore, with this approach, we viewed the parameters as "statistically significant" only if their p-values were below $0.01/16 = 0.000625$.

The FWER criterion is often viewed as overly conservative. An alternative criterion is the False Discovery Rate (FDR), which allows an average pre-specified proportion of false positives in the list of "statistically significant" tests. For comparison, we also adjusted the p-values to control the FDR using the method of Benjamini and Hochberg [1]. An adjusted p-value cutoff of, say, 0.05 implies an average 5% of false positives in the "statistically significant" list.

As we will show next, for our dataset, both of these techniques agree in that they decrease the number of statistically significant associations between languages and defects by one (Ruby is not significant when we adjust for multiple hypothesis testing).

4.2.3 Statistical Significance versus Practical Significance. The FSE article focused on the statistical significance of the regression coefficients. This is quite narrow, in that the p-values are largely driven by the number of observations in the dataset [11]. Small p-values do not necessarily imply practically important associations [4, 30]. In contrast, *practical significance* can be assessed by examining model-based *prediction intervals* [17], which predict future commits. Prediction intervals are similar to confidence intervals in reflecting model-based uncertainty. They are different from confidence intervals in that they characterize the plausible range of values of the future individual data points (as opposed to their mean). In this case study, we contrasted confidence intervals and prediction intervals derived for individual languages from the Negative Binomial Regression. As above, we used the method of Bonferroni to adjust the confidence levels for the multiplicity of languages.

4.2.4 Accounting for Uncertainty. The FSE analyses assumed that the counts of bug-fixing commits had no error. However, labeling of commits is subject to uncertainty: the heuristic used to

Table 6. Negative Binomial Regression for Languages (Gray Indicates Disagreement with the Conclusion of the Original Work)

	Original Authors		Reanalysis							
	(a) FSE [26]		(b) cleaned data		(c) pV adjusted		(d) zero-sum		(e) bootstrap	
	Coef	P-val	Coef	P-val	FDR	Bonf	Coef	Bonf	Coef	sig.
Intercept	−1.93	<0.001	−1.93	<0.001	–	–	−1.96	–	−1.79	*
log commits	2.26	<0.001	0.94	<0.001	–	–	0.94	–	0.96	*
log age	0.11	<0.01	0.05	<0.01	–	–	0.05	–	0.03	
log size	0.05	<0.05	0.04	<0.05	–	–	0.04	–	0.03	*
log devs	0.16	<0.001	0.09	<0.001	–	–	0.09	–	0.05	*
C	0.15	<0.001	0.11	0.007	0.017	0.118	0.14	0.017	0.08	
C++	0.23	<0.001	0.23	<0.001	<0.01	<0.01	0.26	<0.01	0.16	*
C#	0.03	–	−0.01	0.85	0.85	1	0.02	1	0	
Objective-C	0.18	<0.001	0.14	0.005	0.013	0.079	0.17	0.011	0.1	
Go	−0.08	–	−0.1	0.098	0.157	1	−0.07	1	−0.04	
Java	−0.01	–	−0.06	0.199	0.289	1	−0.03	1	−0.02	
Coffeescript	−0.07	–	0.06	0.261	0.322	1	0.09	1	0.04	
Javascript	0.06	<0.01	0.03	0.219	0.292	1	0.06	0.719	0.03	
Typescript	−0.43	<0.001	–	–	–	–	–	–	–	–
Ruby	−0.15	<0.05	−0.15	<0.05	<0.01	0.017	−0.12	0.134	−0.08	*
Php	0.15	<0.001	0.1	0.039	0.075	0.629	0.13	0.122	0.07	
Python	0.1	<0.01	0.08	0.042	0.075	0.673	0.1	0.109	0.06	
Perl	−0.15	–	−0.08	0.366	0.419	1	−0.05	1	0	
Clojure	−0.29	<0.001	−0.31	<0.001	<0.01	<0.01	−0.28	<0.01	−0.15	*
Erlang	0	–	−0.02	0.687	0.733	1	0.01	1	−0.01	
Haskell	−0.23	<0.001	−0.23	<0.001	<0.01	<0.01	−0.2	<0.01	−0.12	*
Scala	−0.28	<0.001	−0.25	<0.001	<0.01	<0.01	−0.22	<0.01	−0.13	

label commits has many false positives, which must be factored into the results. A relatively simple approach to achieve this relies on parameter estimation by a statistical procedure called the bootstrap [17]. We implemented the bootstrap with the following three steps. First, we sampled with replacement the projects (and their attributes) to create resampled datasets of the same size. Second, the number of bug-fixing commits bcommits_i^* of project i in the resampled dataset was generated as the following random variable:

$$\begin{aligned}\text{bcommits}_i^* \sim\ & \text{Binom}(\text{size} = \text{bcommits}_i,\ \text{prob} = 1 - \text{FP}) \\ & + \text{Binom}(\text{size} = (\text{commits}_i - \text{bcommits}_i),\ \text{prob} = \text{FN})\end{aligned}$$

where FP = 36% and FN = 11% (Section 4.1). Finally, we analyzed the resampled dataset with Negative Binomial Regression. The three steps were repeated 100,000 times to create the histograms of estimates of each regression coefficients. Applying the Bonferroni correction, the parameter was viewed as statistically significant if 0.01/16th and (1-0.01)/16th quantiles of the histograms did not include 0.

4.3 Results

Table 6(b)–(e) summarizes the re-analysis results. The impact of the data cleaning, without multiple hypothesis testing, is illustrated by column (b). Gray cells indicate disagreement with the conclusion of the original work. As can be seen, the p-values for C, Objective-C, JavaScript, TypeScript,

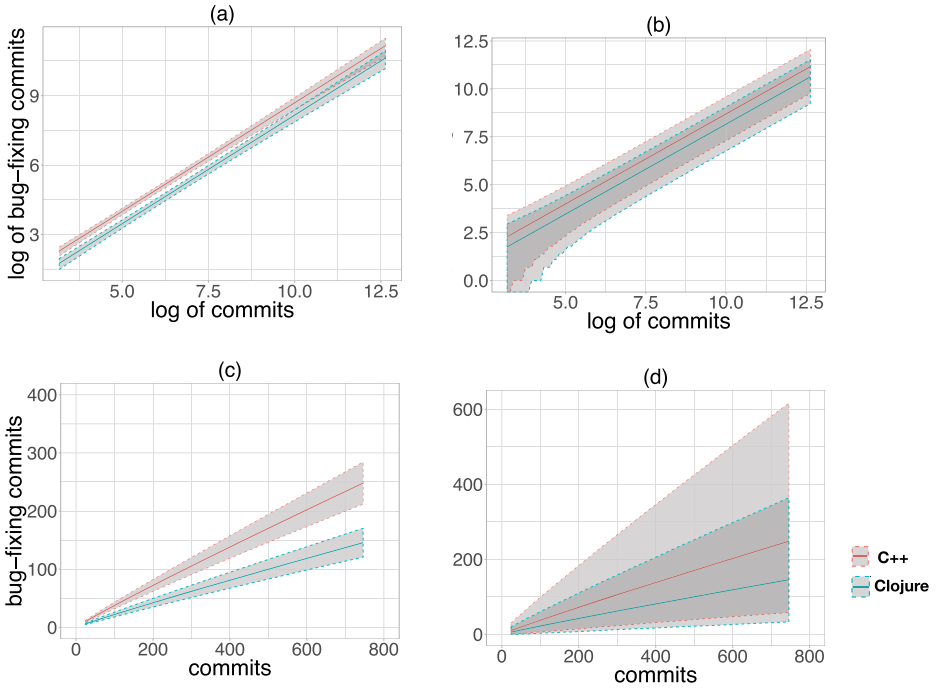


Fig. 6. Predictions of bug-fixing commits as function of commits by models in Table 6(c) and (d) for C++ (most bugs) and Clojure (least bugs). (a) $(1 - 0.01/16\%)$ confidence intervals for expected values on log-log scale. (b) Prediction intervals for a future number of bug-fixing commits, represented by $0.01/16$ and $1 - 0.01/16$ quantiles of the NB distributions with expected values in (a). ((c) and (d)) Translation of the confidence and prediction intervals to the original scale.

PHP, and Python all fall outside of the "significant" range of values, even without the multiplicity adjustment. Thus, 6 of the original 11 claims are discarded at this stage. Column (c) illustrates the impact of correction for multiple hypothesis testing. Controlling the FDR increased the p-values slightly, but did not invalidate additional claims. However, FDR comes at the expense of more potential false-positive associations. Using the Bonferroni adjustment does not change the outcome. In both cases, the p-value for one additional language, Ruby, loses its significance.

Table 6, column (d) illustrates the impact of coding the programming languages in the model with zero-sum contrasts. As can be seen, this did not qualitatively change the conclusions. Table 6(e) summarizes the average estimates of coefficients across the bootstrap repetitions, and their standard errors. It shows that accounting for the additional uncertainty further shrunk the estimates closer to 0. In addition, Scala is now out of the statistically significant set.

Prediction intervals. Even though some of the coefficients may be viewed as statistically significantly different from 0, they may or may not be practically significant. We illustrate this in Figure 6. The panels of the figure plot model-based predictions of the number of bug-fixing commits as function of commits for two extreme cases: C++ (most bugs) commits) and Clojure (least bugs). Age, size, and number of developers were fixed to the median values in the revised dataset. Figure 6(a) plots model-based confidence intervals of the *expected values*, i.e., the estimated average

numbers of bug-fixing commits in the underlying population of commits, on the log-log scale considered by the model. The differences between the averages were consistently small. Figure 6(b) displays the model-based *prediction intervals*, which consider individual observations rather than averages, and characterize the plausible future values of projects' bug-fixing commits. As can be seen, the prediction intervals substantially overlap, indicating that, despite their statistical significance, the practical difference in the future numbers of bug-fixing commits is small. Figure 6(c) and (d) translate the confidence and the intervals on the original scale and make the same point.

4.4 Outcome

The reanalysis failed to validate most of the claims of Reference [26]. As Table 6(d)–(f) shows, the multiple steps of data cleaning and improved statistical modeling invalidated the significance of 7 of 11 languages. Even when the associations are statistically significant, their practical significance is small.

5 FOLLOW UP WORK

We now list several issues that may further endanger the validity of the causal conclusions of the original manuscript. We have not controlled for their impact; we leave that to follow up work.

5.1 Regression Tests

Tests are relatively common in large projects. We discovered that 16.2% of files are tests (801,248 files) by matching file names to the regular expression "`*(Test|test)*`". We sampled 100 of these files randomly and verified that every one indeed contained regression tests. Tests are regularly modified to adapt to changes in API, to include new checks. Their commits may or may not be relevant, as bugs in tests may be very different from bugs in normal code. Furthermore, counting tests could lead to double counting bugs (that is, the bug fix and the test could end up being two commits). Overall, more study is required to understand how to treat tests when analyzing large scale repositories.

5.2 Distribution of Labeling Errors

Given the inaccuracy of automated bug labeling techniques, it is quite possible that a significant portion of the bugs being analyzed are not bugs at all. We have shown how to accommodate for that uncertainty, but our correction assumed a somewhat uniform distribution of labeling errors across languages and projects. Of course, there is no guarantee that labeling errors have a uniform distribution. Error rates may be influenced by practices such as using a template for commits. For instance, if a project used the word `issue` in their commit template, then automated tools would classify all commits from that project as being bugs. To take a concrete example, consider the `DesignPatternsPHP` project: it has 80% false positives, while more structured projects such as `tengine` have only 10% false positives. Often, the indicative factor was as mundane as the wording used in commit messages. The `gocode` project, the project with the most false negatives, at 40%, "closes" its issues instead of "fixing" them. Mitigation would require manual inspection of commit messages and sometimes even of the source code. In our experience, professional programmers can make this determination in, on average, 2 minutes. Unfortunately, this would translate to 23 person-months to label the entire corpus.

5.3 Project Selection

Using GitHub stars to select projects is fraught with perils as the 18 variants of `bitcoin` included in the study attest. Projects should be representative of the language they are written in. The `PHPDesignPatterns` is an educational compendium of code snippets; it is quite likely that is does

represent actual PHP code in the wild. The `DefinitelyTyped` TypeScript project is a popular list of type signatures with no runnable code; it has bugs, but they are mistakes in the types assigned to function arguments and not programming errors. Random sampling of GitHub projects is not an appropriate methodology either. GitHub has large numbers of duplicate and partially duplicated projects [18] and too many throwaway projects for this to yield the intended result. To mitigate this threat, researchers must develop a methodology for selecting projects that represent the population of interest. For relatively small numbers of projects, less than 1,000, as in the FSE paper, it is conceivable to curate them manually. Larger studies will need automated techniques.

5.4 Project Provenance

GitHub public projects tend to be written by volunteers working in open source rather than by programmers working in industry. The work on many of these projects is likely done by individuals (or collections of individuals) rather than by close knit teams. If this is the case, then this may impact the likelihood of any commit being a bug fix. One could imagine commercial software being developed according to more rigorous software engineering standards. To mitigate for this threat, one should add commercial projects to the corpus and check if they have different defect characteristics. If this is not possible, then one should qualify the claims by describing the characteristics of the developer population.

5.5 Application Domain

Some tasks, such as system programming, may be inherently more challenging and error prone than others. Thus, it is likely that the source code of an operating system has different characteristics in terms of errors than that of a game designed to run in a browser. Also, due to non-functional requirements, the developers of an operating system may be constrained in their choice of languages (typically unmanaged system languages). The results reported in the FSE paper suggest that this intuition is wrong. We wonder if the choice of domains and the assignment of projects to domains could be an issue. A closer look may yield interesting observations.

5.6 Uncontrolled Influences

Additional sources of bias and confounding should be appropriately controlled. The bug rate (number of bug-fixing commits divided by total commits) in a project can be influenced by the project's culture, the age of commits, or the individual developers working on it. Consider Figure 7, which shows that project ages are not uniformly distributed: some languages have been in widespread use longer than others. The relation between age and its bug rate is subtle. It needs to be studied, and age should be factored into the selection of projects for inclusion in the study. Figure 8 illustrates the evolution of the bug rate (with the original study's flawed notion of bugs) over time for 12 large projects written in various languages. While the projects have different ages, there are clear trends. Generally, bug rates decrease over time. Thus, older projects may have a smaller ratio of bugs, making the language they are written in appear less error prone. Last, the FSE paper did not control for developers influencing multiple projects. While there are over 45K developers, 10% of these developers are responsible for 50% of the commits. Furthermore, the mean number of projects that a developer commits to is 1.2. This result indicates that projects are not independent. To mitigate those threats, further study is needed to understand the impact of these and other potential biases, and to design experiments that take them into account.

5.7 Relevance to the RQ

The FSE article argues that programming language features are, in part, responsible for bugs. Clearly, this only applies to a certain class of programming errors: those that rely on language

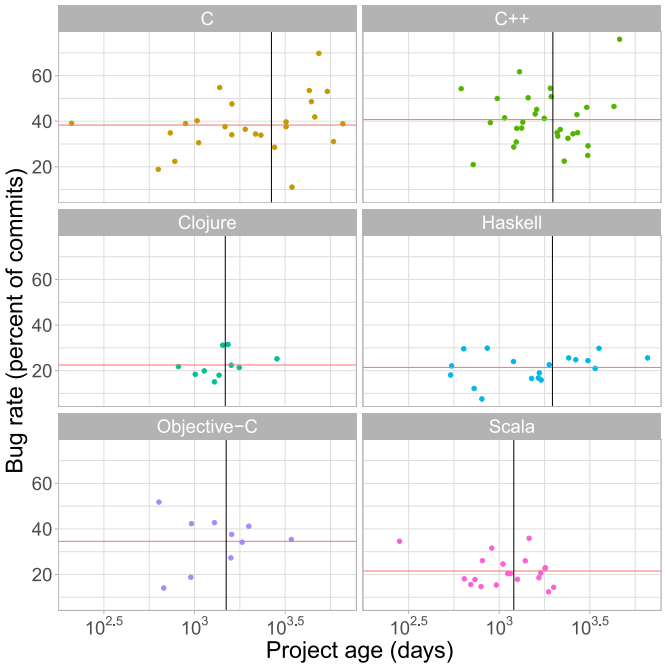


Fig. 7. Bug rate vs. project age. Lines indicate means of project age (x-axis) and bug rate (y-axis).

features. It is unclear if bugs related to application logic or characteristics of the problem domain are always affected by the programming language. For example, setting the wrong TCP port on a network connection is not a language-related bug, and no language feature will prevent that bug,whereas passing an argument of the wrong data type may be if the language has a static type system. It is eminently possible that some significant portion of bugs are in fact not affected by language features. To mitigate this threat, one would need to develop a new classification of bugs that distinguishes between bugs that may be related to the choice of language and those that are not. It is unclear what attributes of a bug would be used for this purpose and quite unlikely that the process could be conducted without manual inspection of the source code.

6 BEST PRACTICES

The lessons from this work mirror the challenges of reproducible data science. While these lessons are not novel, they may be worth repeating.

6.1 Automate, Document, and Share

The first lesson touches upon the process of collecting, managing, and interpreting data. Real-world problems are complex, and produce rich, nuanced, and noisy datasets. Analysis pipelines must be carefully engineered to avoid corruption, errors, and unwarranted interpretations. This turned out to be a major hurdle for the FSE paper. Uncovering these issues on our side was a substantial effort (approximately 5 person-months).

Data science pipelines are often complex: They use multiple languages and perform sophisticated transformations of the data to eliminate invalid inputs and format the data for analysis. For

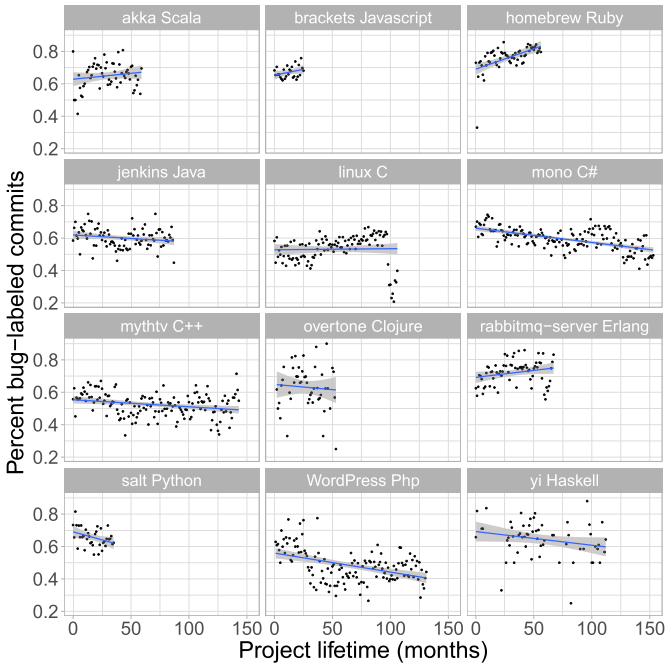


Fig. 8. Monthly avg. bug rate over lifetime. Points are % of bug-labeled commits, aggregated over months.

instance, this article relies on a combination of JavaScript, R, shell, and Makefiles. The R code contains over 130 transformation operations over the input table. Such pipelines can contain subtle errors—one of the downsides of statistical languages is that they almost always yield a value. Publications often do not have the space to fully describe all the statistical steps undertaken. For instance, the FSE paper did not explain the computation of weights for NBR in sufficient detail for reproduction. Access to the code was key to understanding. However, even with the source code, we were not able to repeat the FSE results—the code had suffered from bit rot and did not run correctly on the data at hand. The only way forward is to ensure that all data analysis studies be (a) automated, (b) documented, and (c) shared. Automation is crucial to ensure repetition and that, given a change in the data, all graphs and results can be regenerated. Documentation helps understanding the analysis. A pile of inscrutable code has little value.

6.2 Apply Domain Knowledge

Work in this space requires expertise in a number of disparate areas. Domain knowledge is critical when examining and understanding projects. Domain experts would have immediately taken issue with the misclassifications of V8 and bitcoin. Similarly, the classification of Scala as a purely functional language or of Objective-C as a manually managed language would have been red flags. Finally, given the subtleties of Git, researchers familiar with that system would likely have counseled against simply throwing away merges. We recognize the challenge of developing expertise in all relevant technologies and concepts. At a minimum, domain experts should be enlisted to vet claims.

6.3 Grep Considered Harmful

Simple bug identification techniques are too blunt to provide useful answers. This problem was compounded by the fact that the search for keywords did not look for words and instead captured substrings wholly unrelated to software defects. When the accuracy of classification is as low as 36%, it becomes difficult to argue that results with small effect sizes are meaningful as they may be indistinguishable from noise. If such classification techniques are to be employed, then a careful *post hoc* validation by hand should be conducted by domain experts.

6.4 Sanitize and Validate

Real-world data are messy. Much of the effort in this reproduction was invested in gaining a thorough understanding of the dataset, finding oddities and surprising features in it, and then sanitizing the dataset to only include clean and tidy data [10]. For every flaw that we uncovered in the original study and documented here, we developed many more hypotheses that did not pan out. The process can be thought of as detective work—looking for clues, trying to guess possible culprits, and assembling proof.

6.5 Be Wary of P-values

Our last advice touches upon data modeling and model-based conclusions. Complicated problems require complicated statistical analyses, which in turn may fail for complicated reasons. A narrow focus on statistical significance can undermine results. These issues are well understood by the statistical community, and are summarized in a recent statement of the American Statistical Association [30]. The statement makes points such as "scientific conclusions should not be based only on whether a p-value passes a specific threshold" and "a p-value, or statistical significance, does not measure the importance of a result." The underlying context, such as domain knowledge, data quality, and the intended use of the results, are key for the validity of the results.

7 CONCLUSION

The Ray et al. work aimed to provide evidence for one of the fundamental assumptions in programming language research, which is that language design matters. For decades, paper after paper was published based on this very assumption, but the assumption itself still has not been validated. The attention the FSE and CACM articles received, including our reproduction study, directly follows from the community's desire for answers.

Unfortunately, our work has identified numerous and serious methodological flaws in the FSE study that invalidated its key result. Our intent is not to blame. Statistical analysis of software based on large-scale code repositories is challenging. There are many opportunities for errors to creep in. We spent over 6 months simply to recreate and validate each step of the original paper. Given the importance of the questions being addressed, we believe it was time well spent. Our contribution not only sets the record straight, but more importantly, provides thorough analysis and discussion of the pitfalls associated with statistical analysis of large code bases. Our study should lend support both to authors of similar papers in the future, as well as to reviewers of such work.

After data cleaning and a thorough reanalysis, we have shown that the conclusions of the FSE and CACM papers do not hold. It is not the case that eleven programming languages have statistically significant associations with bugs. An association can be observed for only four languages, and even then, that association is exceedingly small. Moreover, we have identified many uncontrolled sources of potential bias. We emphasize that our results do not stem from a lack of data, but rather from the quality of the data at hand.

Finally, we would like to reiterate the need for automated and reproducible studies. While statistical analysis combined with large data corpora is a powerful tool that may answer even the hardest research questions, the work involved in such studies—and therefore the possibility of errors—is enormous. It is only through careful re-validation of such studies that the broader community may gain trust in these results and get better insight into the problems and solutions associated with such studies.

ACKNOWLEDGMENTS

We thank Baishakhi Ray and Vladimir Filkov for sharing the data and code of their FSE paper; had they not preserved the original files and part of their code, reproduction would have been more challenging. We thank Derek Jones, Shiram Krishnamurthi, Ryan Culppeper, and Artem Pelenitsyn for helpful comments. We thank the members of the PRL lab in Boston and Prague for additional comments and encouragements. We thank the developers who kindly helped us label commit messages.

REFERENCES

[1] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *J. Roy. Stat. Soc. B* 57, 1 (1995). DOI: https://doi.org/10.2307/2346101

[2] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. 2009. Fair and balanced?: Bias in bug-fix datasets. In *Proceedings of the Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*. DOI: https://doi.org/10.1145/1595696.1595716

[3] Casey Casalnuovo, Yagnik Suchak, Baishakhi Ray, and Cindy Rubio-González. 2017. GitcProc: A tool for processing and classifying github commits. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'17)*. DOI: https://doi.org/10.1145/3092703.3098230

[4] David Colquhoun. 2017. The reproducibility of research and the misinterpretation of p-values. *R. Soc. Open Sci.* 4, 171085 (2017). DOI: https://doi.org/10.1098/rsos.171085

[5] Premkumar T. Devanbu. 2018. Research Statement. Retrieved from www.cs.ucdavis.edu/~devanbu/research.pdf.

[6] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*. DOI: https://doi.org/10.1109/ICSE.2013.6606588

[7] J. J. Faraway. 2016. *Extending the Linear Model with R: Generalized Linear, Mixed Effects and Nonparametric Regression Models*. CRC Press.

[8] Dror G. Feitelson. 2015. From repeatability to reproducibility and corroboration. *SIGOPS Oper. Syst. Rev.* 49, 1 (Jan. 2015). DOI: https://doi.org/10.1145/2723872.2723875

[9] Omar S. Gómez, Natalia Juristo Juzgado, and Sira Vegas. 2010. Replications types in experimental disciplines. In *Proceedings of the Symposium on Empirical Software Engineering and Measurement (ESEM'10)*. DOI: https://doi.org/10.1145/1852786.1852790

[10] Garrett Grolemund and Hadley Wickham. 2017. *R for Data Science*. O'Reilly.

[11] Lewis G. Halsey, Douglas Curran-Everett, Sarah L. Vowler, and Gordon B. Drummond. 2015. The fickle p-value generates irreproducible results. *Nat. Methods* 12 (2015). DOI: https://doi.org/10.1038/nmeth.3288

[12] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*. DOI: https://doi.org/10.1109/ICSE.2013.6606585

[13] John Ioannidis. 2005. Why most published research findings are false. *PLoS Med* 2, 8 (2005). DOI: https://doi.org/10.1371/journal.pmed.0020124

[14] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the Conference on Computer and Communications Security (CCS'18)*. DOI: https://doi.org/10.1145/3243734.3243804

[15] Paul Krill. 2014. Functional languages rack up best scores for software quality. *InfoWorld* (Nov. 2014). https://www.infoworld.com/article/2844268/functional-languages-rack-up-best-scores-software-quality.html.

[16] Shriram Krishnamurthi and Jan Vitek. 2015. The real software crisis: Repeatability as a core value. *Commun. ACM* 58, 3 (2015). DOI: https://doi.org/10.1145/2658987

[17] Michael H. Kutner, John Neter, Christopher J. Nachtsheim, and William Li. 2004. *Applied Linear Statistical Models*. McGraw–Hill Education, New York, NY. https://books.google.cz/books?id=XAzYCwAAQBAJ

[18] Crista Lopes, Petr Maj, Pedro Martins, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. Déjà Vu: A map of code duplicates on GitHub. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'17)*. DOI: https://doi.org/10.1145/3133908

[19] Audris Mockus and Lawrence Votta. 2000. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*. DOI: https://doi.org/10.1109/ICSM.2000.883028

[20] Martin Monperrus. 2014. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the International Conference on Software Engineering (ICSE'14)*. DOI: https://doi.org/10.1145/2568225.2568324

[21] Sebastian Nanz and Carlo A. Furia. 2015. A comparative study of programming languages in rosetta code. In *Proceedings of the International Conference on Software Engineering (ICSE'15)*. http://dl.acm.org/citation.cfm?id=2818754.2818848.

[22] Roger Peng. 2011. Reproducible research in computational science. *Science* 334, 1226 (2011). DOI: https://doi.org/10.1126/science.1213847

[23] Dong Qiu, Bixin Li, Earl T. Barr, and Zhendong Su. 2017. Understanding the syntactic rule usage in Java. *J. Syst. Softw.* 123 (Jan. 2017), 160–172. DOI: https://doi.org/10.1016/j.jss.2016.10.017

[24] B. Ray and D. Posnett. 2016. A large ecosystem study to understand the effect of programming languages on code quality. In *Perspectives on Data Science for Software Engineering*. Morgan Kaufmann. DOI: https://doi.org/10.1016/B978-0-12-804206-9.00023-4

[25] Baishakhi Ray, Daryl Posnett, Premkumar T. Devanbu, and Vladimir Filkov. 2017. A large-scale study of programming languages and code quality in GitHub. *Commun. ACM* 60, 10 (2017). DOI: https://doi.org/10.1145/3126905

[26] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar T. Devanbu. 2014. A large scale study of programming languages and code quality in GitHub. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE'14)*. DOI: https://doi.org/10.1145/2635868.2635922

[27] Rolando P. Reyes, Oscar Dieste, Efraín R. Fonseca, and Natalia Juristo. 2018. Statistical errors in software engineering experiments: A preliminary literature review. In *Proceedings of the International Conference on Software Engineering (ICSE'18)*. DOI: https://doi.org/10.1145/3180155.3180161

[28] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying linux bug fixing patches. In *Proceedings of the International Conference on Software Engineering (ICSE'12)*. DOI: https://doi.org/10.1109/ICSE.2012.6227176

[29] Jan Vitek and Tomas Kalibera. 2011. Repeatability, reproducibility, and rigor in systems research. In *Proceedings of the International Conference on Embedded Software (EMSOFT'11)*. 33–38. DOI: https://doi.org/10.1145/2038642.2038650

[30] Ronald L. Wasserstein and Nicole A. Lazar. 2016. The ASA's statement on p-values: Context, process, and purpose. *Am. Stat.* 70, 2 (2016). DOI: https://doi.org/10.1080/00031305.2016.1154108

[31] Jie Zhang, Feng Li, Dan Hao, Meng Wang, and Lu Zhang. 2018. How does bug-handling effort differ among different programming languages? *CoRR* abs/1801.01025 (2018). http://arxiv.org/abs/1801.01025.

Received December 2018; revised May 2019; accepted June 2019

4.3 Paper 3 - CodeDJ: Reproducible Queries over Large-Scale Software Repositories

Petr Maj, Konrad Siek, Alexander Kovalenko, Jan Vitek.

In: 35th European Conference on Object-Oriented Programming (ECOOP 2021). Article No. 7; pp. 7:1–7:24

4.3.1 Author's Contributions

I was responsible for the paper's main idea of implementing CodeDJ, a tool for precise, scalable and reproducible project selection from large software repositories. I was responsible for the overall design of CodeDJ that consists of two components, Parasite and Djanco. I designed and implemented Parasite, which is an incremental GitHub scrapper and reproducible source code warehouse. I performed the reproduction analysis and helped with the query language interface design and calculation of attributes for Djanco. I helped with the artifact preparation.

Konrad Siek implemented the second component, the Djanco query engine and prepared the artifact. Alexander Kovalenko was responsible for the graphs in the paper.

I presented the paper at ECOOP 2021.

4.3.2 Citations

1. Arteca, E. & Turcotte, A. Npm-filter: Automating the mining of dynamic information from npm packages. *Proceedings - 2022 Mining Software Repositories Conference, MSR 2022*. pp. 304-308 (2022)

CodeDJ: Reproducible Queries over Large-Scale Software Repositories

Petr Maj[1]
Czech Technical University in Prague

Konrad Siek[1]
Czech Technical University in Prague

Alexander Kovalenko
Czech Technical University in Prague

Jan Vitek
Czech Technical University in Prague and
Northeastern University

Abstract

Analyzing massive code bases is a staple of modern software engineering research – a welcome side-effect of the advent of large-scale software repositories such as GitHub. Selecting which projects one should analyze is a labor-intensive process, and a process that can lead to biased results if the selection is not representative of the population of interest. One issue faced by researchers is that the interface exposed by software repositories only allows the most basic of queries. CodeDJ is an infrastructure for querying repositories composed of a persistent datastore, constantly updated with data acquired from GitHub, and an in-memory database with a Rust query interface. CodeDJ supports reproducibility, historical queries are answered deterministically using past states of the datastore; thus researchers can reproduce published results. To illustrate the benefits of CodeDJ, we identify biases in the data of a published study and, by repeating the analysis with new data, we demonstrate that the study's conclusions were sensitive to the choice of projects.

2012 ACM Subject Classification Software and its engineering → Ultra-large-scale systems;

Keywords and phrases Software, Mining Code Repositories, Source Code Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.7

1 Introduction

With over 190 million public projects, GitHub is our largest source of empirical data about how software is developed. It is a treasure trove that must be mined if we want to distill insights from its contents. Manual inspection is limited to small-scale case studies; even automated analysis tools struggle with the sheer amount of data available. The software engineering community has taken up this challenge, researchers examine increasingly larger numbers of projects in order to test hypotheses and derive knowledge about the software development process. Examples of such studies include investigations of testing practices [12], changes to licensing over time [18], popularity trends [4] and configuration settings [17]. These works use samples of GitHub ranging from 15K to 100K projects filtered to exclude projects considered as lacking in size, popularity, originality or importance.

For any scientific study of software, selecting the projects that make up the input of that study is fraught with risks. Any given choice can introduce unwanted and sometimes undetected bias. This bias may, in turn, taint the conclusions of the work. Much like the task of polling voters before an election, choosing a subset of a larger population must be

[1] These authors contributed equally.

35th European Conference on Object-Oriented Programming (ECOOP 2021).
Editors: Manu Sridharan and Anders Møller; Article No. 7; pp. 7:1–7:24

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

done carefully. In polls, the goal is to ensure appropriate representation of likely voters. The chosen subset excludes citizens who are either not eligible or unlikely to vote, and balances the various population groups. At the same time, for reasons of cost and practicality, the size of this subset is kept as small as possible. Even when pollsters are careful, the accuracy of predictions varies. In software engineering, we often look for some properties of "real" code — where our definition of the term is sensitive to context and research goals. One may exclude course assignments because the errors made by beginners are not relevant to deployed software; on the other hand, if our goal is to shine a light on acquisition of programming skills, then that kind of code may be exactly what is needed. Picking the right set of inputs is thus the first challenge any researcher in the field must address.

With software, Nagappan et al. warned us that more is not always better [14]. Their observations hold now more so than back in 2013 as anyone can create a GitHub repository at no cost and house almost anything there. Manual inspection found that 37% of hosted projects are not used for software development [11]. Thus, the quality of data gathered from software repositories should always be questioned. A stark illustration why skepticism is in order comes from the finding that ten common source corpora have up to 68% of bit-for-bit identical file duplicates [1]. Furthermore, the same paper showed that clones impacted the accuracy of results obtained with these corpora. We argue that more is worse: as the number of projects to scrutinize grows, it becomes harder to check whether their data is clean, consistent and well-formed. Consider the case of text files accidentally misidentified as code [15], an error that went unnoticed for three years and was "fixed" by partially invalidating the original paper's conclusions [2]. As a result of this state of affairs, researchers spend significant effort collecting and curating meaningful suites of open source projects. Unfortunately, manual curation cannot track the constantly changing software landscape.

In this paper, we aim to address a seemingly simple yet eminently practical question, *how does one find software projects in large-scale software repositories?* The assumption underlying our work, our hypothesis, is that it is possible to select thousands of projects from millions by formulating queries on attributes found in the projects' metadata and on easily computed properties of their source code. To be concrete about the kinds of queries we envision, consider looking for the one hundred most popular projects predominantly written in Java, developed in the five years before the introduction of Lambdas by at least two developers with five years of experience. Furthermore, let's ensure that the selected projects have no more than 5% duplicate files between each other. While the search interface provided by software repositories may allow to query for projects by language, there is no way to compute this query automatically without retrieving all projects.

This paper reports on the status of CodeDJ, an infrastructure for querying large-scale software repositories. In its current incarnation our system is geared towards processing data from any git-based software repository. For our experiments, we specifically target GitHub. The three main engineering challenges we contend with are the sheer size of the data source, the constant updates to its data, and the narrow, rate-limited, interface for accessing projects. In addition, a key design requirement is reproducibility; not only should queries execute deterministically, but the infrastructure should be able to replay a historical query with identical results. Thus, researchers may take any query from the literature, even years after it was originally run and its output was used in a publication, and match its results. Furthermore, researchers should be able to modify a historical query and run it based on the information available at any point in the past.

To address these challenges and requirements, CodeDJ is architected in two distinct subsystems. Interaction with the data source is mediated by Parasite, a time-indexed datastore that automatically and continuously queries GitHub for data about projects. Parasite is responsible for data acquisition and keeping that data up-to-date over time. Every datum is logically time-stamped to enable reproducibility. To ensure that CodeDJ can scale, Parasite can be split up into multiple distinct substores based on the projects' main language. The second subsystem, an in-memory database named Djanco, handles user-written queries. For each query, Djanco determines the portion of the datastore that is required, loads the data, and executes the query. Queries evaluate with project metadata in memory while source code remains on disk. The query syntax is based on data frame manipulation interfaces popular in data science, such as dplyr [19], and is expressed in Rust. We claim the following contributions:

- The design of CodeDJ, a scalable infrastructure for querying large-scale software repositories that supports reproducibility and continuously updated data sources.
- A prototype implementation of Parasite and Djanco written in Rust that shows scalability to millions of projects.
- A dataset consisting of 3.6 million software projects written in 17 languages obtained from GitHub.
- A case study illustrating that the choice of projects can invalidate the conclusion of a research project.

Equally important is what we don't do. We do not provide guidance on how to use our infrastructure. The determination of what is the *right* input for a given analysis is problem specific and the choice remains something individual researchers must grapple with. We have not shown scalability of our infrastructure to the whole of GitHub, we are comfortable with datastores of up to 10 million projects. A larger size may require more work. We do not support interactive queries, our infrastructure was designed with the understanding that queries can take hours to run. We did not focus on optimizing query evaluation by, e.g. parallelizing their execution. Lastly, we do not index any artifacts other than code. Adding images, configuration files and documentation is possible but was not considered one of our targets.

Availability. CodeDJ is an open source infrastructure. Readers interested in repeatability, will find our reproduction package at:

```
https://github.com/PRL-PRG/codedj-ecoop-artifact
```

The source code of Parasite and Djanco are on GitHub at:

```
https://github.com/PRL-PRG/codedj-parasite
https://github.com/PRL-PRG/djanco
```

As our datastore is too large to easily share, Sec. 3.3.4 discusses how external users can run queries on our servers. Another alternative is for users to set up their own CodeDJ instance and gather their own data to execute queries. Our reproduction package contains a complete walk-through of the set up procedure. Of course, users must publish their dataset to enable reproducibility.

2 Related Work

Table 1 gives a high-level comparison with eight systems with aims similar to ours. The first column (*Active*) indicates if the system is actively maintained. Some research projects have fallen into disrepair and their web pages are unreachable. The second column (*Updated*) indicates if continuous updates are supported. Given the rate of addition to GitHub, most systems struggle to keep up. The third column (*Reproducible*) indicates if results are reproducible. Reproducibility is only relevant when the data is updated, systems built on a single static snapshot trivially support reproducibility. The fourth column (*Consistent*) indicates that the data is consistent. Inconsistencies arise when some earlier data (such as parent commits) are missed. The fifth column (*Queries*) describes the nature of the query interface exposed to users. Some systems have a simple filtering mechanism for a fixed set of attributes, such as the language of the project, others have their own query language. In our case, we express queries in Rust. The sixth column (*Sources*) indicates where the data comes from. Mostly this is GitHub, but the Apache Software Foundation and various other sources have also been used in the past. The seventh column (*Size*) is an estimate of how many projects are available. Finally the last column (*Contents*) indicates if source code can be queried. Most systems only include metadata about projects due to the size of the code.

	Active	Updated	Reproducible	Consistent	Queries	Sources	Size	Contents
Stress [8]	–	–	Y	Y	Filter	Apache	211	–
Flossmetrics [9]	–	–	Y	Y	Filter	Many	2.8**K**	–
Orion [3]	–	–	Y	Y	Own	Many	185**K**	Y
Boa [7]	Y	–	Y	Y	Own	Java	380**K**	Y
Black Duck	Y	Y	–	Y	Filter	Many	680**K**	–
Sourcerer [16]	–	–	Y	Y	Filter	GitHub	4.5**M**	–
GHTorrent [10]	Y	Y	–	Y	SQL	GitHub	157**M**	–
GitHub	Y	Y	–	–	Filter	GitHub	190**M**	Y
CodeDJ	Y	Y	Y	Y	Rust	GitHub	3.6**M**	Y

Table 1 Systems comparison

Stress: This system aims to help choose projects in a reproducible manner [8]. Its corpus consists of 211 projects which can be filtered on 100 pre-computed attributes such as bug tickets or lifetime. The corpus can be sorted and sampled randomly. Queries can be exported so they can be repeated later. Source code is not available for querying. Stress is inactive. CodeDJ scales to larger corpora and allows to specify richer queries. In terms of reproducibility, we support updates to the corpus.

Flossmetrics: This work analyzed 2800 open source projects and computed statistics about various aspects of their development process, such as number of commits and developers [9]. Information from additional sources such as project mailing lists and issue trackers was included. Queries could be formulated on metrics such as COCOMO effort, core team members, evolution and dynamics of bugs. Filtering based on these criteria was supported. The project is inactive and it did not support updates.

Orion: This system aimed to enable retrieving projects using complex search queries linking different artifacts of software development, such as source code, version control metadata, bug tracker tickets, developer activities and interactions extracted from the hosting platform [3]. The project is no longer maintained, it scaled to about 185K projects. CodeDJ is designed to scale to larger corpora and offers a more flexible query interface.

Boa: This system focuses on semantics queries over Java programs [7]. A corpus of 380K Java projects can be queried using a dedicated query language that supports automatic parallelization and pluggable mining functions. Source code can be queried in sophisticated ways as Boa is able to parse and analyze Java. A larger corpus of 7.5M projects can be queried on project summaries. Boa provides reproducibility by ensuring its queries are deterministic with respect to the dataset's version, which are created and archived infrequently (i.e. 2013, 2015, 2019, 2020). CodeDJ differs from Boa in that it is language agnostic and geared towards project selection, as opposed to project analysis. Furthermore, CodeDJ provides full reproducibility in the presence of a continuously evolving dataset.

Black Duck Open Hub: A public directory of open source software[2] that offers search services for discovering, evaluating, tracking, and comparing projects. It analyzes both the code's history and ongoing updates to provide reports about the composition and activity of code bases. CodeDJ allows researchers to write their own queries and supports reproducibility.

SourcererCC: The aim of this project is to detect code clones [16]. The tool scales to large datasets and can detect near-identical code at various granularities. It has been used to analyze cloning across large corpora of Java, JavaScript, Python, C and C++ projects on GitHub [13]. It can be used by researchers to detect duplication in their samples which is a source of bias. The project's web page appears to be inactive.

GHTorrent: This database of metadata about GitHub projects offers an SQL interface for queries [10]. It monitors GitHub events to constantly update the available data. The limitation of the approach is that GitHub's events do not have all commit details and file contents, thus these are not stored by GHTorrent. In our experience, the database is not always consistent, this may be due to missed events. We have attempted to upload queries through the public SQL interface but the queries timed out.

GitHub: This service provides two ways to query metadata and contents. A REST API can be used for requesting information about projects and listing them, its search queries provide filtering capabilities across a small set of fixed attributes. A web API provides extended filtering options such as searching within repositories written in a particular language. These interfaces are rate-limited and thus return partial results. The results are non-deterministic and non-reproducible as projects may be added and deleted at any time. CodeDJ provides a view of a subset of GitHub on which we support reproducibility and our queries are richer and deterministic.

We would be remiss if we failed to mention the Software Heritage Archive which aims to preserve all publicly available source code; currently upwards of 9.5B source files, 2B commits and 150M projects [6]. It only allows retrieval of single objects. The authors point to the fragility of current arrangements and the dynamic nature of source code repositories makes it difficult to reproduce studies that use them. We have encountered this ourselves: we see projects deleted from GitHub, changing names, or visibility. In the future, CodeDJ can be extended to query the heritage corpus as well as other repositories.

[2] `https://www.openhub.net`

3 An Infrastructure for Querying Large-Scale Repositories

The goal of CodeDJ is to allow researchers to formulate queries that evaluate attributes of projects hosted on GitHub and return data about projects matching a specified predicate.

3.1 Design considerations and system architecture

The design of CodeDJ flows from four high-level principles that we motivate next:

- **Consistent, eventually:** The sheer size and churn in data sources such as GitHub means that obtaining a snapshot of the whole data source is not practical. But, it is often the case that a slightly out-of-date view is sufficient for most investigations. We choose to refresh entire projects atomically at irregular intervals. Thus, any individual project is consistent, but for any group of projects, the lower bound on their refresh times is the last consistent time point (`git` histories can be destructively updated, allowing for post factum inconsistencies, we ignore these).
- **Code-centric, language agnostic:** We aim to support queries on project metadata and file contents written in any programming language. To reduce space requirements, the only source artifacts we store is code, deduplication is used to remove redundancy, and metadata is trimmed where possible.
- **Flexible query interface:** Popular data science tools such as dplyr [19] or Spark [20] offer a mix operations inspired by database query languages extended with general purpose capabilities. Inspired by these, we propose an interface expressed in Rust as a library with operations for selecting, grouping, filtering and sampling data. The benefits of our approach over, say, SQL, is that queries are type-safe and benefit from the full generality of the Rust language.
- **Reproducible by design:** The importance of reproducibility cannot be overstated [5], consider [15] which recorded the names of the most starred projects seven years ago, without author names it is not possible uniquely to identify projects, and even with their full names, reconstructing a historical star count is not possible. CodeDJ is designed so it is possible to run any query with the information that the datastore had at an arbitrary point in the past. For this purpose the datastore is time-indexed, strictly append-only.

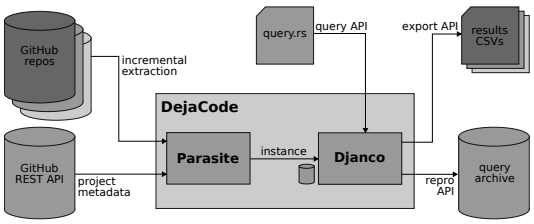


Figure 1 System overview

Fig. 1 overviews the architecture of CodeDJ. The system is structured around two components, Parasite, a datastore that tracks GitHub, and Djanco, an in-memory database with a Rust interface. Parasite is set up to continuously extract information from GitHub using its REST API for some data and cloning project repositories for other data. The information obtained

from the data source is deduplicated and stored in a dedicated format on disk. At irregular intervals projects are refreshed, and the new information is appended to the datastore. When an end-user query is submitted for execution, it comes as a Rust function calling the Djanco query API, a database instance is created for that query. The database will load the data needed for query execution from Parasite. The output of a query is some results, usually as a text file and a record of that query in a reproducibility archive.

The remainder of this section describes our implementation, the design of the query interface and our support for reproducibility.

3.2 The Parasite datastore

Parasite is a dedicated, perpetually running application whose task is to synchronize its on-disk representation with GitHub. This task is complicated by these four constraints:

- **Scalable:** We expect to grow to hundreds of millions of projects, the disk format must be space efficient and its in memory format must be compact and fast to access.
- **Peaceful co-existence:** We must abide by GitHub's terms of service. Parasite must be economical in both the number requests to the GitHub API calls and raw `git` operations.
- **Time-indexed:** Every datum in the store must be associated with its acquisition date, this feature must have a minimal overhead so as not to increase our footprint.
- **Robust:** Backups are not possible due to limited resources, the datastore must thus be resilient to corruption.

Our description focuses on three aspects, the data acquisition process, the data storage format and the interface exposed to Djanco. We also explain how we meet the above constraints.

3.2.1 Acquisition

While, in theory, the GitHub API is sufficient to fulfill all our needs, the fact that GitHub defends itself against denial of service attacks limiting users to 5,000 requests per hour causes a practical problem. As every commit requires one request, the interface is too restrictive to collect data within a reasonable amount of time. Therefore, instead of relying on the API alone, Parasite combines a number of interfaces:

- **Git:** we use the `git clone` command to retrieve source code files and commit histories from repositories;
- **GitHub:** we use the REST API for project metadata (stars, watchers, issues, etc.), information that cannot be obtained through `git` alone;
- **GHTorrent:** instead of querying GitHub for projects directly, we seeded Parasite with the URLs of projects obtained from GHTorrent.[3]

Parasite continuously downloads data from its data sources on a per-project basis. The projects known to Parasite are maintained in a priority queue. Projects are visited in inverse order of last access time. Thus, given any group of projects, the lower bound on the time they were last visited determines the last point when Parasite had a consistent view of those projects modulo destructive `git` history rewrites.

When a project is visited, the download procedure begins. First, the project's metadata is retrieved via a call to the REST API. This yields a JSON file with metadata and sundry

[3] While GHTorrent has over 100M URLs, they are not all valid. Out of 5.5M URLs we visited, only 3.6M were usable, the remaining are either duplicates, have been deleted, or have become private.

information. The metadata is stripped of non-essential information (such as URLs for various REST API requests) and stored. The project's current and last known URLs are compared to detect renaming and the new URL is recorded if a change occurred. Next, the project's heads are checked against the heads in the datastore. Each head corresponds to a branch in `git`. If any of the heads changed, the project is cloned and data about new commits and the contents of changed files are extracted and stored. We clone projects because using the REST API to get new commits is slow and rate limited. We clone repeatedly at each visit, caching projects is not feasible due to space limitations (in the future, we plan to cache the most active projects to reduce the amount of data unnecessarily transferred via full clones).

Once a local copy of a project exists, we determine which *substore* that project belongs to and append new commits and files to it. Substores are partitions of the dataset that Parasite uses to organize its disk structures around. Projects are matched to a single substore by properties such as size (e.g. a substore for small projects) or dominant language (e.g. a substore of Python projects).

When processing a chain of commits, a simple optimization is achieved by observing that if we find a commit that is already in the datastore, then all of its parent commits must also already be present. The final step is to record the time of the visit, and move to next project in the queue. Any error during the processing, terminates the visit and the project is flagged as potentially invalid.

Parasite is written in Rust using `libgit2`. It has been parallelized at project-level granularity and scales up to 32 threads. With more threads, the bottleneck shifts from local repository analysis to network bandwidth and ultimately to the GitHub rate limit. When adding projects, Parasite processes 244 projects per thread per hour. As GitHub limits are attached to users (identified by tokens), Parasite supports rotating multiple tokens which allow us to sustain a download rate of 7821 projects per hour using 32 threads. Since Parasite is still in accretion mode, we cannot report on the update rate alone, but we expect it to be limited by GitHub to a rate of 120K active project updates per day per token.

	Records	Size	Ratio
Users	4.8M	200M	<0.01%
Projects	3.6M	4.9G	0.2%
Commits	167M	88G	3.2%
Paths	848M	80G	2.9%
Files	463M	2603G	93.7%

Table 2 Current dataset composition

Parasite has visited 3.6M projects composed from all non-fork C++ and Python projects available in GHTorrent and a random subset of 50K projects in 17 popular languages. In total, the datastore has 3.6M projects and occupies 2.8TB on disk. Table 2 shows that the majority of the datastore is taken by source code.

3.2.2 Storage

The storage format of Parasite is designed to ensure a low disk footprint, to scale to hundreds of millions of projects. The store is append-only to allow reverting to historic states and to simplify recovery from data corruption. Parasite can be thought of as storing *records*. Records of same kind are backed by a single *record file*. Records compose together to form *entities*. The following entities are stored by Parasite:

- **Projects:** A project is identified by unique `git` clone URL, it has a set of *heads* (one per branch) and other information from GitHub metadata.
- **Commits:** A commit is identified by its SHA hash, it has a message, changes, parents, an author, a committer, and a time.
- **Paths:** A path is identified by the hash of its string value.
- **Users:** A user is identified by their email.
- **Snapshots:** A snapshot of a file containing source code is identified by its hash.

Records are the smallest unit of information in the datastore, the only way to update an entity is to add a new record. The decomposition of entities to records has been designed along the lines of what information can be updated in isolation. Entities are assigned unique numeric *identifiers* based on their contents. One of the key internal data structures in Parasite are the multiple *mappings* from entity hashes to identifiers. These mappings are used for deduplication.

Deduplication is crucial as up to 94% of files can be duplicates [13]. Mappings are costly as they must be kept in memory. For our corpus, the deduplication mappings for all entities require 89GB. While not a concern at this time, as our dataset grows, mappings will become a bottleneck. To decrease their size, we split Parasite into *substores*. Each substore manages a disjoint partition of the projects. We perform deduplication only within substores. This means that mappings are smaller at the price of some duplication across substores. Our implementation assigns projects to substores based on their size and dominant language; small projects (less than 10 commits) are kept distinct from projects written in targeted languages. A drawback of this design is that identifiers are not unique, if multiple substores must be accessed, extra care must be taken when merging their contents. On the other hand, this compartmentalization has immediate benefits: In terms of robustness, different substores can be stored in different locations and a loss of one does not impact the others. In terms of performance, queries can trivially skip reading irrelevant substores. We measured the duplication across substores at only 5.1%.

As source code (snapshots) dominate the datastore, Parasite internally splits snapshots by language, storing each language separately. This improves reading times for queries that filter by language.

Parasite avoids storing information that is expensive to update and that can be computed readily. For instance, the relation between commits and their project is not stored; it can be recovered from project heads and commit parents. To further reduce footprint, larger records are compressed. For snapshots, the compression ratio is 70%.

To quickly find the latest records for a particular entity, Parasite computes indices, which are stored in dedicated *index files* that provide, for each entity, the location of the latest version of its constituent records. These index files are updated in place as new records are added which exposes them to the risk of being inconsistent. If this occurs, they can always be recomputed from scratch. As of this writing, all indices in the datastore comprised 0.6% of our disk footprint.

To ensure that it is possible to associate a time with every datum on disk, Parasite introduces the notion of a *savepoint*. Since the store is append-only, time-indexing in Parasite boils down to simply associating a time to the current position of each substore. For consistency, savepoints can only be created between visits of projects. They are thus both a mechanism for reproducibility and robustness. Any query can be re-executed at any savepoint and will see the same information. The datastore can be rolled back to a savepoint in case of data corruption.

3.2.3 Interfaces

Parasite has two interfaces, one for data acquisition and another for reading data.

For monitoring purposes data acquisition exposes a detailed breakdown of running tasks, their progress and the usage of GitHub resources. Parasite has both an interactive text-based interface and a command-line interface for automation via scripts. These interfaces allow to create savepoints, verify integrity of the datastore and repair data corruption by reverting to previous savepoints. Parasite monitors available memory to keep as many mappings in memory as it can. Most of the datastore management can be done without reloading any mappings; the initial load takes 26 minutes.

The read interface allows to access records. Iterators are created relative to a savepoint and return records in the order they were added up to that savepoint. Many records are never superseded, for these iterator return values can be used as such. For records that can be overridden with newer values, iterators return updates in reverse chronological order. For projects, Parasite assembles their information; this takes some time as URLs, heads, update status, substore, and metadata must be loaded first, assembly discards all but the most recent versions. Iterators are geared towards sequential access to all elements, but the index files kept by Parasite can be used for random access as well.

3.3 The Djanco database

The Djanco database acts as an intermediary between Parasite and the end-user. It provides a robust query engine that manages loading and pre-processing data and a domain-specific language to express queries easily and concisely. Finally, it supports replaying historical queries. Djanco is designed under the following simplifying assumptions:

- **Single-user:** Djanco is used by a single user for a single query at a time; any parallelism is internal and transparent.
- **Determinism:** Queries are fully replayable on the basis of parameters explicitly provided by the end-users such as random seeds, timestamps, and data source.
- **Read-only:** Queries cannot update the datastore, changes are limited to local objects and are not persisted.
- **Fixed-schema:** Djanco only contains data and metadata pertaining to GitHub.

The need for Djanco comes from the structure of Parasite. The datastore is designed to allow continuous updates and to decrease footprint. This complicates answering research questions. For instance, Parasite elides the relation from a project to its commits. A simple question such as how many commits there are in a project requires recomputing that relation by looking up one the of project's branches and its most recent commit. From that commit, one can follow the parent commits and recursively enumerate them all. Then, repeat for all branches. The database layer computes relations such as these and caches data persistently to speed up queries.

The rationale for a dedicated database rather than an off-the-shelf one are threefold. First, and most arguably, our experience using MySQL on a related project suggested that scalability to large data size (2.8TB and growing) can lead to significant execution overheads. Secondly, we can leverage the assumptions above to implement a domain-specific database as many features of traditional databases (transactions, locks, a general schema) are superfluous. Instead, we implement a solution specialized to our schema that lazily loads selected data from the datastore. Finally, some of our queries are difficult to express in the relational model. Queries can become lengthy and involve multiple joins, nesting and views, which makes them difficult to debug and maintain.

3.3.1 Instances

A Djanco database *instance* is logically created for each end-user query. Each instance is irrevocably tied to a specific slice of the datastore. This slice is defined by two parameters: the substores that indicate which projects to load, and a timestamp indicating a savepoint to be checked out from each substore. If multiple datastores are used, the database joins and deduplicates them.

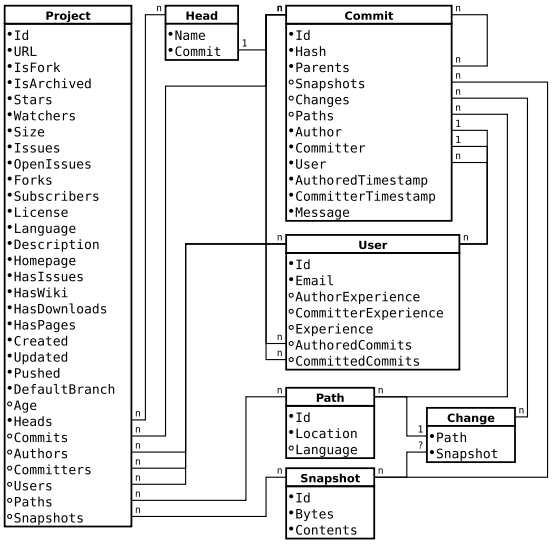


Figure 2 Djanco schema (computed attributes marked ∘)

The Djanco schema is shown in Fig. 2, it defines five different entities: projects, commits, paths, users and snapshots. Each with their own attributes and convenience methods. Even though Djanco derives its schema from Parasite, there is not a one-to-one correspondence between them. While Parasite tends towards generality and frugality, Djanco instead tends towards expressivity and convenience. For instance, Parasite stores project metadata in JSON, while Djanco parses the format, extracts useful information at sensible types. The basic information about projects is their ID and URL. The metadata includes:

- the language as determined by GitHub;
- the numbers of stars, watchers, subscribers, issues, and forks;
- dates for creation, most recent update, and most recent push;
- the license, description, and homepage URL;
- which web services are active: issues, wiki, downloads, pages;
- size in bytes;
- name of the default branch (e.g. “master” or “main”);
- whether the project is archived or a fork.

Djanco provides a method to calculate the age of a project as the span of the time between its first and most recent commit. Finally, it provides methods to retrieve relations between a

project and other entities: heads, commits, users, authors, committers, paths, and snapshots. Except for heads, all the relations need to be computed.

Commits have IDs, hashes, messages, as well as timestamps at which they were authored and pushed. Each commit is associated with users, having an author and a committer. A commit also has a list of changes: a change is a modification to a file represented by a path in the repository and the contents of the file after the change. Finally, commits reference a list of zero or more parent commits in the commit tree.

Users have IDs and emails. In addition, experience is computed for authors and committers as the timespan between their first and last commit. Users also have a method to acquire the list of commits they authored or committed.

Paths represent file system locations within the project (e.g. `"src/main.c"`). They are identified by a synthetic ID and contain a string representing the path. A method to guess the language of a file from its extension is provided. Snapshots are the stream of bytes that are contents of a file at some point in time. For instance, if a file is edited during a commit, the contents of that file before and after the edit are two separate snapshots.

3.3.2 Queries

Queries can be expressed either through a low-level interface or via a DSL. The former accesses the schema directly with Rust iterators and methods. The DSL is a more compact way to implement common queries.

The first step for all queries is to construct a database instance. Since an instance wraps around a specific view of the datastore, constructing it requires specifying a path, a savepoint and substores. The following snippet constructs an instance for small projects available on December 1st, 2016:

```
let db = Djanco::new(PATH, timestamp!(December 2016), substore!(SmallProjects))?;
```

Alternatively, an instance for C, C++, and Python programs is constructed like this:

```
let db = Djanco::new(PATH, timestamp!(December 2016), substores!(C, C++, Python))?;
```

Parameters can be skipped; an instance from all substores at their most recent savepoint is constructed thus (values of defaults are recorded for reproducibility):

```
let db = Djanco::from(PATH)?;
```

Iterators offer access to entities. The snapshot iterator is lazy, the others eagerly load information from the datastore. Iterators are entry points to queries; they return objects that conform to the schema of Fig. 2. This snippet extracts a vector of all languages occurring in projects:

```
let all_languages = db.projects()
  .map(|project| project.language())
  .unique()
  .collect()::<Vec<Language>>;
```

While iterators suffice for just about any query, most queries can be expressed more concisely in our DSL. The DSL uses a pipeline paradigm, where an initial data structure is transformed by a series of methods (aka verbs) that do part of the processing in each step. We provide the following verbs: `group`, `filter`, `sort_by`, `sample`, and `map_into`. We also provide access to any attribute in the schema. In addition, objects and their attributes are composable into complex statements expressing comparisons (e.g. `AtLeast`, `AtMost`, `Matches`, `Contains`), basic statistical functions (`Count`, `Max`, `Median`), sampling methods (`Top`, `Random`), and many others. The code below showcases a few of these:

```
let selection = db.projects()
  .group_by(project::Language)
  .filter_by(AtLeast(Count(project::Users), 5))
  .sort_by(project::Stars)
  .sample(Top(50));
```

Projects are grouped according to their language, then filtered so that only projects that have at least 5 users are kept, these are sorted by the number of stars in each project and, finally, a sample of top 50 projects is returned.

A useful feature is the ability to deduplicate projects while sampling them according to specific criteria. For example, in the following snippet projects will not be added to the result set unless 90% of their commits are unique with respect to any other project already within the result set:

```
selection.sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)))
```

The final step of a query is to output its results; here we show results written to a CSV file:

```
selection.into_csv(OUTPUT_PATH)?;
```

Each object serializes verbosely, including all information about itself. If only specific information is required, an appropriate format may be imposed by using the map verb to translate an object into its attributes. Here each project is translated into its ID and URL:

```
selection
  .map_into(Select!(project::Id, project::URL))
  .into_csv(OUTPUT_PATH)?;
```

We also provide a function that outputs all information related to a project, including commits, users, paths and snapshots. This creates multiple CSV files.

```
selection.dump_all_info_to(OUTPUT_DIR_PATH)?;
```

Crucially, end-users can do their own use-case–specific formatting by resorting to Rust:

```
selection.for_each(|project| println!("{}: {}", project.url(), project.has_wiki()))
```

Further details about our query facilities can be found in the Djanco GitHub repository.

A friend in need We had an opportunity to test our system when posed a question that was difficult to answer with GitHub's REST API. The query had to retrieve popular C++ repositories that use custom allocators. Finding out whether a project is using a custom allocator requires checking if it imports a library called `memory_resource`. Therefore, we

```
let wanted: HashMap<SnapshotId> = db
 .snapshots()
 .filter(|snapshot|
   snapshot.contains(
    "#include <memory_resource>"))
 .map(|snapshot| snapshot.id())
 .collect();

let projects = db.projects()
 .filter(|project| {
  project.snapshots()
   .map_or(false, |snapshots| {
    snapshots.iter()
     .map(|snapshot| snapshot.id())
     .any(|snapshot_id| {
      wanted.contains(snapshot_id)
     })
   })
 .sorted_by_key(|project|
  project.star_count());
```

```
let wanted: HashSet<SnapshotId> = db
 .snapshots()
 .filter_by(
   Contains(snapshot::Contents,
     "#include <memory_resource>"))
 .map_into(snapshot::Id)
 .collect();

let projects = db.projects()
 .filter_by(
   AnyIn(project::SnapshotIds, wanted))
 .sort_by(project::Stars);
```

Figure 3 Emery query

grep through source code for the string `"#include␣<memory_resource>"`. In a second step, we iterate over projects and find those, which contain one of the selected snapshots. At that point, we order them by popularity and retrieve some number of the most popular projects. For comparison we wrote the query in pure Rust and then in the DSL. Both implementations are in Fig. 3. As expected the DSL is more compact and more readable. We ran the query on a store with 3M projects and 429M snapshots. The first part of the query found 1724 snapshots in 12 hours. The second part of the query retrieved 1197 projects and their metadata in 24 hours. Then, an additional 6 hours was spent on preparing the project metadata for CSV export.

3.3.3 Data management

Djanco transparently manages the loading and pre-processing of data from the datastore. This involves two mechanisms: lazy loading and caching. Given the size of the data, loading it all into memory is not desirable. Most queries are interested with a small slice of the data, usually filtering out most projects and neglecting most attributes. Therefore, Djanco uses lazy loading to tailor the in-memory data to the needs of each specific query. Snapshots (source code files) are bulky and cannot be split into independent attributes. Only a single snapshot is held in memory at once. The database retrieves them from the datastore only when needed either by scanning the store sequentially or by using the datastore's ability to seek and access a specific snapshot. For the other objects (projects, commits, paths, and users), their attributes are loaded independently on request. Attributes are cached in the database as they can be needed several times.

Memory usage is not the only concern while loading data from the store. From our experiences in querying GitHub, we find that many similar queries are executed on the same datastore view, especially when a query is being developed. Loading attributes from the datastore can be costly, especially in places where the Djanco schema requires the values to be calculated, e.g. for mappings between entities. Therefore, we found it beneficial to avoid recalculating some attributes across queries by implementing on-disk attribute caching, thus improving performance of similar or repeated queries.

For each attribute occuring in a query, the database creates an in-memory map, mapping an entity ID to that entity's value for a given attribute. After an attribute has been loaded, the caching extension serializes it onto disk using the CBOR serialization format. The on-disk cache structure preserves information about which datastore, savepoint, and substore a particular attribute map was read from. Subsequent queries requesting this attribute for this particular datastore view then prefer loading data from the cache rather than the datastore. This process is transparent to the end-user, and can be turned off to save disk space.

	extracting from store	writing to cache	reading from cache	size on disk	cached?
commit::Parents+commit::Users	1h 21m 28s	35m 16s	7m 25s	2.3GB	Y
user::Experience	1h 10m 19s	1s	1s	5.7MB	Y
user::CommitterExperience	1h 9m 52s	1s	1s	5.6MB	Y
user::AuthoredCommits	1h 8m 47s	1m 1s	39s	213MB	Y
project::Commits	1h 8m 33s	5m 29s	3m 25s	1.1GB	Y
commit::Changes	52m 29s	2h 53m 53s	1h 21m 28s	20GB	N
commit::CommitterTimestamp	41m 49s	1m 55s	1m 21s	418MB	Y
commit::Message	41m 24s	3m 20s	1h 38m 3s	6GB	N

Table 3 Caching performance

However, while the cache uses up disk space, reading an attribute from CBOR is potentially orders of magnitude faster than loading it from the store. On the other hand, when loading from the store is simple and the data is difficult to serialize (e.g. it consists of large string vectors) caching is not indicated. We have benchmarked and pre-tuned the database to cache only when it is clearly advantageous. Table 3 shows the performance impact of caching while extracting selected attributes on a dataset containing 130K projects and 44M commits. The table lists a few representative attributes in the first column. Columns two and three present what happens when the attribute is requested for the first time: how long it takes to extract it from the datastore and how long it takes to subsequently serialize it onto disk. The fourth and fifth columns show the impact of caching: how long it takes to read the argument from cache (e.g. when the query is re-executed or when another query requires the same attribute from the same datastore view) and how much disk space has to be devoted to the CBOR file. The final column shows our decision whether to cache this attribute or not.

3.3.4 Availability

While users can download their own datasets and run queries on them locally, doing so requires time and computational resources. Therefore, we also provide a procedure for running queries on our hardware using our incrementally updated dataset. A durable, publically available resource also fosters reproducibility.

The submission procedure plugs into the standard Rust toolkit. Queries are submitted as cargo crates. These crates include functions marked as individual queries via annotations which also specify the savepoint and subsets that the specific query expects. For convenience, we provide a template for query crates that works with the `cargo generate` command.[4] We also provide an accompanying `cargo djanco` command[5] which generates an execution harness around query functions. The harness is a small standalone Rust program that sets up the datastore and runs each query according to the specifications found in their annotations. The harness includes a commandline interface through which it can be executed with a specific dataset paths, output directory, and other parameters. We generate the harness for executing the query on our server, but it can be used to test queries locally as well.

As of this writing queries are scheduled manually by the authors. Users should contact us by email with a link to the repository. The query will undergo a manual inspection and will be executed on our hardware and dataset using the same generated harness as above. After the query is executed, a snapshot of the crate is created and stored it in the query archive. The snapshot contains the complete source code of all the queries, logs, the exact generated harness used for execution, and the results of all the queries — files generated to the designated output directory. Any result file exceeding 50MB is ignored (if a query produces large files we contact the user to advise on compaction or to negotiate different means of delivery).

In the future, we will extend our infrastructure to include a web API that will allow users to execute queries themselves. These queries will be expressed in a limited query language (to obviate security risks) and the volume of results will be limited. Queries and results will also be archived and accessible publicly with a receipt. Another extension we foresee is to extend the existing mechanism to allow automatic query execution. This would resemble our current process but it would remove the need for a manual check and emailing the authors

[4] `https://github.com/PRL-PRG/djanco-query-template#template`
[5] `https://github.com/PRL-PRG/cargo-djanco`

as submission could be automated. This option is contingent on our ability to create a static checker for incoming crates and sufficiently isolating them during execution.

Finally, storing user emails has privacy issues. we are considering whether it is appropriate to expose emails for external queries. If retaining emails becomes problematic, we may have to obfuscate the emails and replace them with numeric identifiers.

3.3.5 Reproducibility

To further support reproducibility, above and beyond the ability to deterministically run historical queries, every query executed by Djanco is stored in a public query archive. The query archive is a git repository hosted on GitHub.[6] Each query is hosted in a separate branch in the repository. We expect queries to undergo revisions. Each revision and execution results from that revision are archived as separate commits in a single branch. This produces a development history of the query.

Each query execution produces a *receipt* — a hash representing a specific commit in the archive repository representing the execution. The hash can be used to share queries (exactly as executed) and their results (exactly as produced). It can be used to retrieve the cargo crate and to re-execute the code (e.g. on a different dataset). Code re-execution is helped by the fact that queries are deterministic and the snapshot of the crate contains a list of all depedencies, a timestamp, a list of all subsets and all random seeds. The receipt for the queries in this paper is `da6ae7dd50565e84efbeac990f5788f383939014`.[7]

4 A Case Study: Of Bugs and Languages

The work's motivation is the claim that the *selection of inputs matters in empirical studies of software and that CodeDJ can assist researchers in that process.* We illustrate these points with a case study. We start from prior work, and show that input selection impacts scientific claims, and that CodeDJ allows rapid exploration of the input space.

The starting point is a Foundation of Software Engineering (FSE) paper published in 2014 [15].[8] One contribution of that work is to establish that some programming languages have a greater association with defects than others (RQ1 in [15]). Their methodology can be summarized as follows. For 17 popular languages, select 50 projects hosted on GitHub that have at least 28 commits. For each commit touching a file that contains code in one of the target languages, label the commit as bug-fixing if its message contains a bug-related keyword. Fit a Negative Binomial Regression (NBR) against the labeled data and obtain, for each language, a coefficient and a p-value. The coefficient indicates the strength of the association (positive means more bugs), and the p-value tells us about statistical significance (less than .05 means the coefficient is significant). The FSE paper concluded that TypeScript, Clojure, Haskell, Ruby and Scala were associated with *fewer* bugs, while C, C++, Objective-C, JavaScript, PHP and Python were associated with *more* bugs. The remaining languages did not have statistically significant coefficients.[9]

[6] `https://github.com/PRL-PRG/codedj-query-archive`
[7] `https://github.com/PRL-PRG/codedj-query-archive/tree/da6ae7dd50565e84efbeac990f5788f383939014`
[8] A revised version of the work appeared in the Communications of the ACM in 2017 with some issues fixed, notably the removal of TypeScript from the analyzed languages.
[9] These results were questioned, but the issues raised in [2] are orthogonal to the selection of inputs.

4.1 Corpus

For this experiment we created a datastore using stratified sampling of data available on GHTorrent. We started with 11,000 projects with at least 28 commits written in each of the 17 languages. For each language, we added 6,000 projects randomly selected from GitHub (including smaller projects). In total, our dataset had 172K projects with 28 or more commits and 230K projects in total. Only 3.8K large Erlang projects were available. The dataset has 47M unique commits (and 66M commits in total, suggesting a commit-duplication of 30%, high given forks were excluded). The datastore occupies 51GB on disk. Our goal was to have enough variety to represent the richness of GitHub. Unlike the FSE paper, which was written in 2013, our corpus goes all the way to 2020.

4.2 Random input selection

Our first experiment explores the distribution of possible analysis outcomes. For this, we repeatedly pick a random subset of 50 projects of each of the 17 languages and fit them with NBR. Fig. 4 shows the distribution of the coefficients obtained by 1000 such random selections compared to the results obtained in [15] (shown as a tick to the right of the distribution). Positive values indicate a higher association of the language with defects.

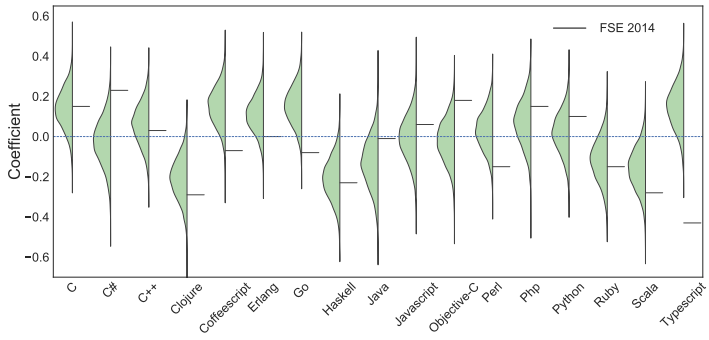


Figure 4 Random subsets

The spread of each distribution is a measure of the sensitivity of the analysis to its inputs. Intuitively, consider the distribution of coefficients for Objective-C, it is roughly centered around 0. This means, that a random input is about equally likely to say that the language has a positive association with defects as a negative one. One could argue that picking close to the median of the distribution could give a representative answer. As we can see the FSE paper often picks subsets that are outliers; see the cases of CoffeeScript, Go, Perl, Scala and most strikingly TypeScript.

Discussion: As most distributions straddle the axis, random selection is likely to result in noisy conclusions. But, GitHub is noisy itself — for instance there is much code duplication, and the are many low quality projects. A random selection is not the appropriate choice for making conclusions about software developed by professionals. One could choose to mitigate selection bias by increasing the size of the sample; CodeDJ can be used to generate multiple random inputs, if the inputs agree, then our confidence in the results increases.

4.3 Observing change over time

As we have more data than was available in 2013, we can use CodeDJ to select inputs at various times. Here we create eight datasets, each containing data up to one of the years between 2013 and 2020. For simplicity, we only plot the distribution of coefficients for TypeScript. The original paper's coefficient was −.43 (shown as a red line). The graph clearly shows that the value was an outlier. The association with bugs shifted over time, increasing to a relatively stable position from 2016.

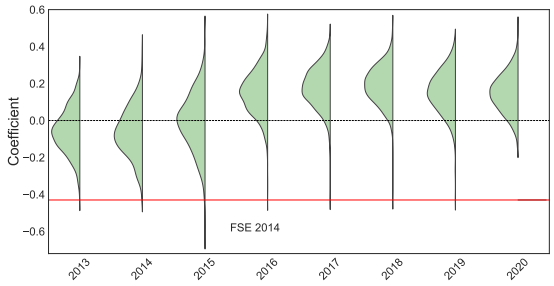


Figure 5 TypeScript over time

While it is reasonable to expect variations from year to year, TypeScript experienced a rather large shift over a short period. The language was released in 2012, so there were few projects on GitHub in 2013. Furthermore, a number of human language translation files were misidentified as TypeScript; these files did not have bugs, biasing the result. The rising popularity of TypeScript quickly caused real code to crowd out the translation files, and the association with bugs settled to around 0.2.

Discussion: Using CodeDJ to prepare inputs at different time points can help researchers spot trends in the data. For some properties of interest one expects changes over time, for others changes may be an indication of bias that needs to be controlled for. For instance, one would expect the association with bugs of an established, popular, language to be stable.

4.4 Introducing domain knowledge

Choosing any subset of a larger population introduces bias, but this may be intentional, reflecting domain knowledge about the relative importance of observations. For instance, small projects with few commits may be less interesting as they correlate with student projects. These projects have fewer descriptive commit messages and their defects reflect beginner mistakes. It stands to reason to exclude such projects from consideration. Justifying the choice of any particular selection criterion is beyond the scope of our work. CodeDJ allows researchers to explore the impact of various subsets. Our next experiment looks at 6 different criteria for selecting projects and compares them to the original paper's criterion. The Djanco code for those queries is in Fig. 8 in the appendix.

- **Stars:** Pick projects with most stars. *Rationale:* starred projects are popular and thus likely to be well written and maintained. [**Used in FSE 2014**]
- **Touched Files:** compute #files changed by commits, pick projects that changed the most files. *Rationale:* indicative of projects where commits represent larger units of work.

- **Experienced Author:** experienced developers are those on GitHub for at least two years; pick a sample of projects with at least one experienced contributor. *Rationale:* less likely to be throw-away projects.
- **50% Experienced:** projects with two or more developers, half of which experienced. *Rationale:* focus on larger teams.
- **Message Size:** Compute size in bytes of commit messages; pick projects with the largest size. *Rationale:* empty or trivial commit messages indicate uninteresting projects.
- **Number of Commits:** Compute the number of commits; pick projects with the most commits. *Rationale:* larger projects are more mature.
- **Issues:** Pick projects with the most issues. *Rationale:* issues indicate a more structured development process.

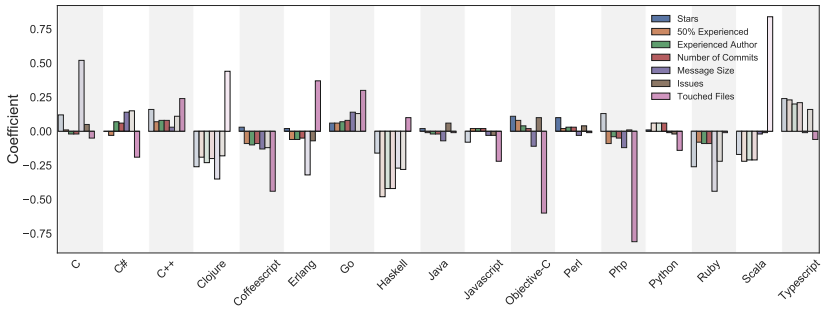


Figure 6 Domain knowledge

Fig. 6 shows, for each language, the value of the coefficients (higher means more bugs); the queries returned 50 projects in each of the 17 target languages: Coefficients that are not statistically significant are shown in faded colors. If the input set did not matter for the model, one could expect the different queries to give roughly the same coefficients with the same significance. This is not the case. If we focus on how many languages have statistically significant coefficients: The touched files query is highly predictive, 14 of the languages are significant, but the coefficients are frequently opposite from those of other queries. Specifically, C is associated with slightly fewer bugs, so are C#, CoffeeScript, Java, JavaScript, Objective-C, Perl, PHP, Python, Ruby and TypeScript. On the other hand C++, Erlang, Go and Haskell are associated with more defects. This is striking as it goes against expectations. The stars query is the least informative. It only gives 7 statistically significant coefficients with remarkably low values.

Discussion: While some queries yield broadly similar conclusions, this is not the case for all. We stress the importance of understanding the selection criteria and its impact, as statistical significance should not be confused with validity. To help, CodeDJ provides distributions of various measures in the data, Fig. 7 visualizes the distribution of project sizes (left) and project age (right) for the entire dataset and for the various queries.
Looking at these distributions makes it clear that the queries return quite different projects. The experienced author and number of commits are remarkably similar and return projects that meet our expectations. The issues distribution is similar, which should raise red flags given that it frequently disagrees. The stars query returns many smaller projects. Finally,

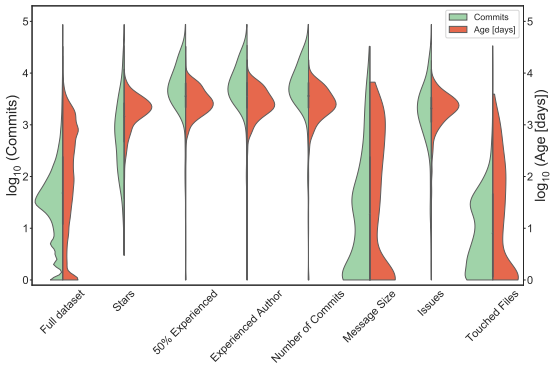


Figure 7 Project Size and Age Distributions

message sizes and touched files show distributions opposite to those expected. They favor degenerate young projects with few commits that are either verbose, or disproportionately large (touching over 100K files). This is reflected in the input sizes, ranging from 8M rows for the experienced author query to mere 79K rows of the touched files query. It is likely that these queries are "wrong" in the sense they do not return the population of interest. The figure also suggest that stars is a bad choice.

5 Conclusions

Finding projects on GitHub is akin to looking for the proverbial needle in a haystack. While having a wealth of data at our fingertips is an undeniable asset to empirical software engineering research, the sheer size of the code being hosted is a challenge to any data processing pipeline. Selecting manageable subsets of available projects can introduce subtle, but significant biases that, in turn, can influence or even invalidate the conclusion of the analysis being conducted. Our case study illustrates this problem — we demonstrate that by choosing various, apparently sensible, subsets of the data at hand, we can significantly change the observed association between programming languages and software defects.

This paper introduces CodeDJ, an infrastructure designed to support the reproducible specification of selection criteria for projects hosted on large-scale software repositories. Our implementation is geared towards GitHub. As GitHub is a living system undergoing constant change, ensuring reproducibility requires extra work. The same project downloaded today and last month may contain different code, different commit histories, or the project may disappear entirely. Our infrastructure mitigates this problem by building on a time-indexed, append-only datastore. Queries are expressed in a front-end database that can access a view of the data at a specific point in the history of the datastore.

For future work, three directions stand out: Expanding the datastore, improving the query evaluation performance, and extending accessibility of the our dataset. The dataset provided contains only a fraction of the data we expect to eventually need. As the data grows in volume, our downloading, storage, and processing capabilities will be put to the test and adjusted accordingly to ensure they scale up. We will explore how to ensure backwards compatibility and determinism of queries in the face of changes to the implementation, and

to the data format (e.g. adding new information, such as issues, or new file kinds). In terms of performance, our implementation does not try any optimizations of the query evaluation. We intend to parallelize queries and explore ideas from the database community regarding query compilation strategies. Finally, we plan on extending our infrastructure. We will create a web API and a limited query language to make our dataset more generally accessible. We will also investigate an infrastructure for automatic security checking and execution scheduling for query crates which would allow for their automated submission.

Acknowledgments. This work is supported by the Czech Ministry of Education, Youth and Sports from the Czech Operational Programme Research, Development, and Education, under grant agreement No.CZ.02.1.01/0.0/0.0/15_003/0000421 and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 695412).

References

1 Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2019. doi:10.1145/3359591.3359735.

2 Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. On the impact of programming languages on code quality: A reproduction study. *ACM Trans. Program. Lang. Syst.*, 41(4):21:1–21:24, 2019. doi:10.1145/3340571.

3 T. F. Bissyande, F. Thung, D. Lo, L. Jiang, and L. Reveillere. Orion: A software project search engine with integrated diverse software artifacts. In *International Conference on Engineering of Complex Computer Systems*, 2013. doi:10.1109/ICECCS.2013.42.

4 Hudson Borges, André C. Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of GitHub repositories. *CoRR*, 2016. URL: http://arxiv.org/abs/1606.04984.

5 Andy Cockburn, Pierre Dragicevic, Lonni Besanc on, and Carl Gutwin. Threats of a replication crisis in empirical computer science. *Communications of the ACM*, 2020. doi:10.1145/3360311.

6 Roberto Di Cosmo and Stefano Zacchiroli. Software Heritage: Why and How to Preserve Software Source Code. *International Conference on Digital Preservation*, 2017. URL: https://hal.archives-ouvertes.fr/hal-01590958.

7 Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *International Conference on Software Engineering (ICSE)*, 2013. URL: http://dl.acm.org/citation.cfm?id=2486788.2486844.

8 Davide Falessi, Wyatt Smith, and Alexander Serebrenik. Stress: A semi-automated, fully replicable approach for project selection. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017. doi:10.1109/ESEM.2017.22.

9 Jesus M. Gonzalez-Barahona, Gregorio Robles, and Santiago Dueñas. Collecting data about FLOSS development: The FLOSSMetrics experience. In *International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS)*, 2010. doi:10.1145/1833272.1833278.

10 Georgios Gousios and Diomidis Spinellis. GHTorrent: GitHub's data from a firehose. In Michael W. Godfrey and Jim Whitehead, editors, *Working Conference on Mining Software Repositories (MSR)*, 2012. doi:10.1109/MSR.2012.6224294.

11 Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining GitHub. In *Working Conference on Mining Software Repositories (MSR)*, 2014. doi:10.1145/2597073.2597074.

12 P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. Adoption of software testing in open source projects–a preliminary study on 50,000 projects. In *European Conference on Software Maintenance and Reengineering*, 2013. doi:10.1109/CSMR.2013.48.

13 Crista Lopes, Petr Maj, Pedro Martins, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjà Vu: A map of code duplicates on GitHub. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017. doi:10.1145/3133908.

14 Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Diversity in software engineering research. In *Foundations of Software Engineering (FSE)*, 2013. doi:10.1145/2491411.2491415.

15 Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *International Symposium on Foundations of Software Engineering (FSE)*, 2014. doi:10.1145/2635868.2635922.

16 Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: scaling code clone detection to big-code. In *International Conference on Software Engineering (ICSE)*, 2016. doi:10.1145/2884781.2884877.

17 Gerald Schermann, Sali Zumberi, and Jürgen Cito. Structured information on state and evolution of dockerfiles on github. In *International Conference on Mining Software Repositories (MSR)*, 2018. doi:10.1145/3196398.3196456.

18 Christopher Vendome, Gabriele Bavota, Massimiliano Di Penta, Mario Linares-Vásquez, Daniel German, and Denys Poshyvanyk. License usage and changes: a large-scale study on GitHub. *Empirical Software Engineering*, 2016. doi:10.1007/s10664-016-9438-4.

19 Hadley Wickham, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain Franc ois, Garrett Grolemund, Alex Hayes, Lionel Henry, Jim Hester, Max Kuhn, Thomas Lin Pedersen, Evan Miller, Stephan Milton Bache, Kirill Müller, Jeroen Ooms, David Robinson, Dana Paige Seidel, Vitalie Spinu, Kohske Takahashi, Davis Vaughan, Claus Wilke, Kara Woo, and Hiroaki Yutani. Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686, 2019. doi:10.21105/joss.01686.

20 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010. doi:10.5555/1863103.1863113.

A Analysis with GitHub toolkits

Can users do without CodeDJ? Consider the case study queries: stars and touched files.

GitHub exposes a REST API that can return any object and its metadata. The API is limited. It allows filtering by language and sorting by stars, but not by touched files. Furthermore it only returns 1000 results. Therefore, we can't get directly the 17K projects of the case study. While repositories can be obtained by numeric IDs, given the rarity of some of languages such as Erlang means that a random sample would, in the worst case, end up sampling every project on GitHub.

Repository URLs can be retrieved with the `/repositories` query. Assuming 150M repositories, 1.5M queries are needed to find them all. The rate limit is 5K queries/user/hr, so this takes 12 days. We also need language and number of commits to perform stratified sampling. Getting languages is another 12 days. This can be done by getting a list of contributors and summing up their contributions. This only requires one query per repository, so another 12 days. Stratified sampling thus requires approximately a month.

The GitHub data is in JSON, which is not easy to query. One can convert it into a more useful format, such as a relational database. From there, one can retrieve top 50 most-starred projects in each language within that dataset with a query like:

```
select id from (
  select id, row_number() over(partition by language order by stars desc) as place
  from projects
) ranks
where place <= 50;
```

The second use case query requires ordering projects by average number of changes per commit. This requires information about all commits. The REST API can list commits, but not changes. To get those, the detailed metadata of each commit is need. This requires one query per commit. With 66M commits, that is 550 days. Deduplicating commits before retrieval shaves this down to 391 days. Having retrieved the data, one can select projects:

```
select id from (
  select id, row_number() over(partition by lang order by avg_touched desc) as place
    from (
      select id, language as lang, avg(touched) as avg_touched
      from project_commits
      join (
        select commit_id, count(path_id) as touched
        from commit_changes
        group by commit_id
      ) touched on project_commits.commit_id = touched.commit_id
      join projects on projects.id = project_commits.project_id
      group by project_id, language
    ) projects
) ranks
where place <= 50;
```

The query is complex. An alternative is to update the data with precomputed attributes.

As the reader may have gathered using GitHub is impractical. An alternative is to use multiple sources of information. Project URLSs, stars and commit counts can be obtained from GHTorrent, commits can be obtained by cloning repositories and analyzing their logs locally. However, these sources have their own shortcomings. GHTorrent does not contain all information, and it can be out of date. For instance, we found commit and star counts off by orders of magnitude. Cloning repositories requires significant bandwidth. In addition, care must be taken with large projects as they can take weeks to analyze if approached naïvely. Gathering data never goes smoothly. The code will likely run for weeks even if massively parallel and then fail on some unexpected corner case. If one then continuously and incrementally update the obtained dataset, then one has essentially reinvented CodeDJ.

B Domain queries

Fig. 8 gives the queries used to inject domain knowledge in the analysis discussed in Sec. 4.

Stars:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .sort_by(project::Stars)
  .sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)));
```

Touched Files:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .sort_by(Median(FromEach(project::Commits, Count(commit::Paths))))
  .sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)));
```

Experienced Author:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .filter_by(AtLeast(Count(FromEachIf(project::Users,
                                      AtLeast(user::Experience,
                                              Duration::from_years(2)))), 1))
  .sort_by(Count(project::Commits))
  .sample(Distinct(Random(50, Seed(42)), MinRatio(project::Commits, 0.9)));
```

50% Experienced:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .filter_by(AtLeast(Count(project::Users), 2))
  .filter_by(AtLeast(Ratio(FromEachIf(project::Users,
                                      AtLeast(user::Experience,
                                              Duration::from_years(2))),
                           project::Users),
                     Fraction::new(1,2)))
  .sample(Distinct(Random(50, Seed(42)), MinRatio(project::Commits, 0.9)));
```

Message Size:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .sort_by(Mean(FromEach(project::Commits, commit::MessageLength)))
  .sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)));
```

Number of Commits:

```
Djanco::from(PATH).projects()
  .group_by(project::Language)
  .sort_by(Count(project::Commits))
  .sample(Distinct(Top(50), MinRatio(project::Commits, 0.9)));
```

Figure 8 Domain queries

4.4 Paper 4 - The Fault in Our Stars: How to Design Reproducible Large-scale Code Analysis Experiments

Petr Maj, Stefanie Muroya, Konrad Siek, Jan Vitek

Submitted to 37th European Conference of Object-Oriented Programming (ECOOP 2023).

4.4.1 Author's Contributions

I was responsible for the paper's idea of providing new methodology for precise project selection, designed with the aim to replace the convenience sampling by project popularity. I have designed the methodology, performed the acquisition and curation of all datasets, dataset analysis, stars analysis, most extensions of the Djanco query engine and one of the dataset experiments (called "What constitutes a software"). I was responsible for the artifact and have supervised junior members of the team.

Konrad Siek performed literature review and reviewed the selection criteria used in existing papers. He also performed one dataset experiment. Stefanie Muroya performed two dataset experiments.

The Fault in Our Stars
How to Design Reproducible Large-scale Code Analysis Experiments

Petr Maj
Czech Technical University
Stefanie Muroya
Czech Technical University
Konrad Siek
Czech Technical University
Jan Vitek
Czech Technical University

Abstract

Software engineering benefits from the insights gleaned from large-scale software repositories as they offer an unmatched window into the software development process. Their sheer size holds the promise of broadly applicable results. At the same time, that very size presents scalability challenges. The traditional answer to such challenges is to limit studies to representative samples and generalize observations to the entire population. The contribution of this paper is both modest and, we believe, important. We advocate in favor of a standardized experimental design methodology for experiments over large-scale repositories. In particular, we steer researchers away from using extrinsic attributes such as stars, and emphasize careful delineation of the population of interest backed up by random sampling of inputs.

2012 ACM Subject Classification Software and its engineering → Ultra-large-scale systems;

Keywords and phrases software, mining code repositories, source code analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.23

1 Introduction

And so it begins

> *We count the number of stars associated with each repository. The number of stars relate to how many people are interested in that project. Thus, we assume that stars indicate the popularity of a project. We select the top 50 projects in each language...*

Sentences like these appear in the methodology sections of our papers. They are often all there is to be found in terms of experimental design. This paper aims to convince readers of the dangers that this state of affairs presents for generalizability and reproducibility of our results and to suggest some simple improvements.

Large-scale code repositories such as GitHub are a boon to the software engineering community as they give us a large body of software along with metadata written in many languages with various degrees of care and expertise. The number of artifacts for each of the major language ecosystems ranges in the millions. With a little patience and enough storage, a researcher can acquire thousands of projects for their latest research effort. Unfortunately, obtaining an entire ecosystem is difficult, and analyzing it may be prohibitive – in hardware resources and researcher effort.

Empirical software engineering studies are experiments performed on a corpus of software to validate some hypothesis. The value of any given experiment does not lie in what we

37th European Conference on Object Oriented Programming (ECOOP 2023).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

learn about the projects that were analyzed, but rather in what they teach us about the larger population. There is little value in, say, learning that 10 particular Java projects adopted a new language feature if we cannot generalize that result to a broader portion of the ecosystem. Yet, few papers articulate their claims of generality and it is often not even clear how researchers selected the software artifacts they studied.

Table 1 has a meta-study of three editions of *Mining Software Repositories* (2019, 47 papers; 2020, 45 papers; 2021, 48 papers). Out of 140 papers, 46 do not have an experimental component that involves software, 24 analyze very small curated datasets, and 29 use the entire available populations. This leaves 41 papers which analyze code obtained from a larger population: 5 are not reproducible, lacking information about how their dataset was constructed or using proprietary data, 21 use GitHub stars to filter projects, 10 use combinations of attribute thresholds and only 5 use random sampling. In summary, out of 41 large-scale code analysis papers, 51% rely on stars.

papers	projects	classification	description
46	–	incompatible	No experiments
24	–	curated	Small curated datasets
29	–	everything	Entire population
5	1–35K	unknown	Unknown or proprietary
21	5–2M	stars	Filter projects using stars
10	7–290K	other	Other filter for projects
5	6–51K	random	Filter and sample randomly

Table 1 Experimental design summary (MSR 2019–2021)

Why do GitHub stars play such a central role in our experimental methodology? We want to think it is neither malice nor sloth, but rather expectations and pragmatics. Community standards are set by the papers we publish. The literature codifies expectations for authors of the next batch of papers. These expectations slowly evolve in response to reviewer attitudes. So we use stars because our peers do, but the pragmatics are just as important. GitHub does not provide an index of its projects, nor does it allow to query over intrinsic attributes of code. Finding inputs is thus hobbled by limitations of our tools. One wants to find projects of interest while avoiding the duplicates that litter most language ecosystems and weeding out obviously uninteresting projects. Absent any other tools, stars play a double role. First, they are an index of projects, one that can be queried from the GitHub interface. Second, there is an expectation that they correlate with some notion of quality. But, not only do stars not accomplish that, they introduce reproduction barriers into project selection. So what to do?

We propose a methodology for designing reproducible experiments with the explicit goal of improving the generalizability of our results. The methodology is in line with evolving community standards [21] but specific to large-scale code analysis, and we emphasize the needed for proper tooling to ensure reproducibility. Our approach takes the form of the following protocol:

1. **Population Hypothesis:** A brief description of the population of interest, what the research should generalize to, which may be a narrow slice such as “programs written by students learning JavaScript as their first language” or a broader one such as “commercial code”.

2. **Frame Oracle:** A procedure for deciding if a project belongs to the population. Ideally, an algorithm efficiently computed over intrinsic attributes of a project. An oracle could, e.g., return GitHub projects with one JavaScript file which were created by a user with no previous commits.
3. **Sampling Strategy:** A strategy for selecting a subset of the values of the population. Ideally, specified algorithmically. An example is random sampling without replacement from a known seed.
4. **Validity:** An argument about the oracle's and sampling strategy's validity as means to obtain representative samples from the population. A discussion of attempts to validate result quality, such as manual inspection of a sample to check if JavaScript code was actually written by beginners.
5. **Reproduction Artifacts:** The artifact should allow to reproduce exactly the reported results as well as to change either the input or the experiment.

Reproducibility has nuances. We specifically do not talk about the experiment itself – others have been there before us. Instead our emphasis is on inputs and support for the following three use cases: *Repetitions* which run the reproduction artifact to obtain bit-for-bit equal results (or as close as feasible). This is the most stringent use case and often requires a reproduction artifact that bundles code and inputs. *Reanalysis* alters either the method or its input, it requires an executable artifact and a method for acquiring new inputs. Finally, *reproductions* are independent implementations that require the paper to have an unambiguous description of all experimental details.

Design experiments that support reproducibility can be greatly simplified with appropriate tooling. Our work builds on the open source CodeDJ infrastructure.[1] Our contributions, briefly, are:

1. **A dataset** of 2Mio+ Java, Python and JavaScript projects. Modified CodeDJ to compute 36 intrinsic project attributes. (Sec. 4)
2. **A characterization of stars** as a means to select inputs for code analysis experiments. (Sec. 4)
3. **A methodology** that can be readily adopted by researchers to improve reproducibility of their work. (Sec. 5)
4. **A reproduction** of four papers that highlights challenges to generality. (Sec. 3 and 6)

Our work is open source and reproducible.[2]

2 Related Work

We review relevant advice, warnings and the state of tooling.

Community Standards. A strong push towards reproducibility is underway, efforts such as the standards framework of [21] include a section on experimental design and specifically sampling. These ideas are further explored by [1] who argue that software engineering faces a generalizability crisis. They carried out a meta-analysis of 120 papers in all areas of the field and report that purposive and convenience sampling are widely used. Such sampling

[1] `https://codedj-prg.github.io`

[2] A polished artifact will be submitted for evaluation should our paper be accepted. For now, we share an anonymized code bundle. We have several terabytes of data, the Program Chairs kindly agree to act as intermediary should reviewers require access: `https://github.com/unknown-john/ICSE21-Anonymized`

techniques rarely lead to representative samples, and – without a careful study of potential sources of bias – can lead to conclusions that do not generalize. They explain this state of affairs by a fundamental challenge in the field, the lack of appropriate sampling frames to access elements of the population of interest. Earlier work by [17] already attempted to address this problem by defining the notion of sample coverage to assess the quality of the data used as input to an experiment. Even closer to our paper is the study by [4] which reported that out of 93 large corpus papers, 63 papers failed to provide replication datasets. Most papers did not use random samples and omitted mentions of limitations.

Mining Repositories. GitHub is extremely popular data source. Warnings about perils go back to the work of [9] who highlighted "noise" among hosted projects. In particular they point out that tiny and inactive projects dominate the platform. [11] pour oil on that fire by showing that up 95% of the code in some language ecosystems were copies. Stars are known to be widely used as a mean to find signal in that sea of noise. But what do they mean? [2] surveyed users and found that the most common reasons for starring a project are to show appreciation (e.g. *I starred this repository because it looks nice.*) and bookmark it (e.g. *I starred it because I wanted to try using it later.*). They also warn against promotional campaigns in social media driving up ratings. Popularity of projects was studied by [7] who suggest that while most users believe stars are the best metric to determine popularity of a project, other attributes such as branches, open issues and contributors are better predictors. Expending on that result, [16] propose to use random forest to create a classifier for *engineered projects*, which they define as projects that leverage sound software engineering principles. Their classifier outperforms stars. [20] further improved classification with an approach based on time-series clustering.

Tools for Miners. A number of infrastructures have been developed to assist researchers in the field. The most ubiquitous is GHTorrent [6], a continuously updated database of metadata about public projects that is a valuable building block for other tools. Boa is complementary as it lets users write sophisticated queries over source code [5]. CodeDJ is a newer infrastructure that supports queries over both meta-data and file contents [13]. Unlike Boa it is language agnostic. [12] and [15] address performance issues of querying at scale. Of these, only CodeDJ ensures reproducible queries.

3 State of Practice

How do people design experiments for large-scale code studies? This section give examples from the meta-study of Table 1. We emphasize to the reader, it is not our goal to criticize individual authors, but it is helpful to establish a baseline to improve community best practices. Following our proposed methodology, for each paper, we give a brief summary of the scientific claim followed by an account of the paper's stated population hypothesis, a description of the frame oracle, sampling strategy, validation and reproduction artifacts. We conclude with some observations.

3.1 MSR 2020: What is Software

"Software" has an intuitive definition, namely code, but there is more. [19] classifies the content of repositories in categories such as code, data and documentation. They then observe that software is more than just code. Documentation is an integral constituent of software, and software without data is often correlated with libraries, and finally that software without code is rare, but exists.

Population Hypothesis: The paper answers the question *"what are the constituents of software and how are they distributed?"* The authors claim that existing definitions of the term are non-descriptive, inconclusive and even contradictory. Implicitly the population is all inclusive.

Frame Oracle: Any software project hosted on GitHub.

Sampling Strategy: Convenience sampling; the authors chose popular repositories and further clarify that *"by popularity we mean the starred criteria with which GitHub users express liking similar to likes in social networks."*

Most-starred projects in 25 languages were acquired by executing one query by language, saying that *"without language qualifier, the API returns only 1,020 repositories in total, which we decided is not enough for our study."*

Validity: No discussion of relevant issues.

Reproducibility Artifacts: A listing of files and repositories is provided with the code of the classifier. Figures and numbers produced by a notebook are also included. The contents of the repositories analyzed are not preserved.

3.2 MSR 2020: Method Chaining

In an object-oriented language, a *method chain* occurs when the result of a method invocation is the receiver of a subsequent method call. In Java, method chaining manifests as a sequence of calls connected by dots. [18] analyze trends in usage of method chains and conclude that their use increased over a period of eight years.

Population Hypothesis: Java projects developed *"by real-world programmers."* The authors state that they *"did not apply any filter to the collected repositories. This supports the generalizability of our results."* The authors consider generalization beyond Java, saying *"our results are more likely to be applied to a language that does not provide such a construct (e.g. PHP and JavaScript). The empirical study of this hypothesis is future work."* The construct in question is support for DSLs.

Frame Oracle: Implicitly defined as all Java projects hosted on GitHub.

Sampling Strategy: The authors use convenience sampling, taking 2,814 projects that appeared at least once in the list of the 1K most-starred projects between November and December 2019. Projects were deduplicated and filtered for syntactically invalid files.

Validity: –

Reproducibility Artifacts: Project metadata and computed chain lengths are published.[3] Communication with the authors reveals that their complete reproduction package is currently not available.

3.3 MSR 2019: Style analyzer

Each software project seems to develop its own formatting conventions. [14] demonstrate that an unsupervised learning algorithm can automate project-specific code formatting. They reproduce the style of a project with a high degree of precision on a dataset of repositories with one base and one head commit specified for each.

[3] https://zenodo.org/record/3697939#.YSYcZ9OA63I

Population Hypothesis: From statements made about the outcome of the experiment, we surmise that the population is that of *"real projects."* From tools, mechanics and the sample prepared for the experiment, we suppose the authors aimed for developed projects. The population is implicitly limited to JavaScript as the tool includes a parser for that language.

Frame Oracle: The oracle is implicitly all JavaScript project hosted on GitHub.

Sampling Strategy: Convenience sampling: 19 JavaScript projects with high numbers of stars are picked. Date of selection is not provided.

Validity: Authors manually inspected projects in the selection.

Reproducibility Artifacts: A GitHub repository containing the tool and a file with project URLs along with their head and base commits is provided.[4] Contents of repositories are not included. Documentation, run scripts and configuration information are patchy.

3.4 MSR 2020: Code Smells

Code smells are programming idioms that are often correlated with correctness or maintenance issues. [8] contrast code smells in projects related to deep learning and general purpose software projects. Their scientific claim is that for large and small projects there is a statistical difference in the occurrence of code smells, whereas medium sized projects are indistinguishable.

Population Hypothesis: The authors are interested in two populations: On one hand projects that implement or use deep learning algorithms, and general purpose software on the other. For pragmatic reasons, they focus on the Python ecosystem as it is widely used for machine learning.

Frame Oracle: Projects must be in Python and hosted on GitHub. Keyword search is used for machine learning frameworks and technologies such Tensorflow and Keras, discarding tutorials. Furthermore, the authors *"also carefully select popular and mature DL projects from them by employing maturity and popularity metrics (e.g., issue count, commit count, contributor count, fork count, stars)."*

Sampling Strategy: A staged strategy was employed to sample both populations. The authors relied on judgment sampling to manually select 59 deep learning projects. For general purpose projects, they used a top-starred list of 106 Python projects from [3] and randomly sampled 59 projects. Projects were further clustered into small (≤ 4000), medium, and large (≥ 15000).

Validity: –

Reproducibility Artifacts: A listing of the 59 deep learning projects is provided.[5]

3.5 Reproducibility issues

While these four research projects were done with care, none can be fully reproduced. Reproducibility failures have many reasons, most of which are common to several of the papers we have reviewed:

- *Missing descriptions:* Failure to specify either one of: population hypothesis, frame oracle or sampling strategy. Reproduction is fraught with perils and an apple-to-apple

[4] `https://github.com/src-d/style-analyzer`
[5] `https://github.com/Hadhemii/DLCodeSmells/blob/master/data/dlRepos.csv`

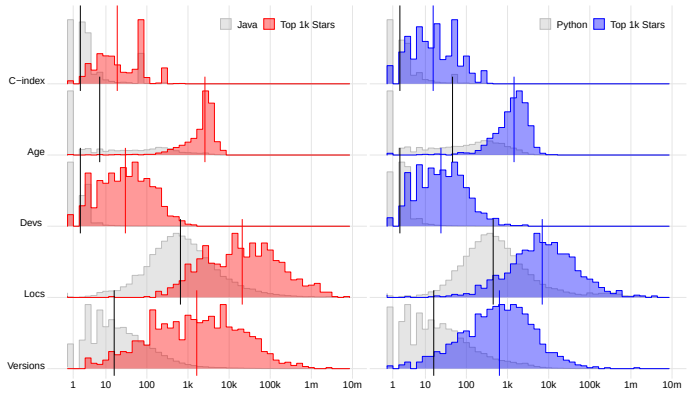


Figure 1 Comparing datasets

comparisons between papers is difficult. This affects [19, 18, 14, 8] as their descriptions are open to interpretation.

- *Missing projects:* Even with a list of URLs, the corresponding projects may vanish at any time (e.g., deleted or made private). Reproductions are partial at best, we have seen a project disappear while being downloaded. This affects [19, 18, 14, 8].
- *Fading stars:* Stars are volatile. [18] observed close to 3,000 projects in the top 1K during a period of two months. Without a history of star attribution and a timestamp, reconstructing the star listings is not possible. This affects [8].
- *Shifting contents:* The contents of a project change with new commits. To reconstruct the data, ids of the last observed commit must be specified. Even that is not foolproof as Git histories can be updated destructively. This affects [19, 18, 8].
- *Language attribution:* Projects contain code in many languages. For reproduction attribution must specified. While dlegating to, e.g. GitHub, is reasonable, one should be aware that GitHub has changed their attribution algorithm several times. Double counting a project is sometimes valid. This affects [19, 18, 14].
- *Deterministic replay:* Non-determinism must be limited. Random sampling seeds should be specified. This affects [14].

4 Mapping the GitHub Landscape

The meta-study of Table 1 highlights the dominant position of GitHub as a data source in large-scale code analysis studies. We claimed that convenience sampling using stars as a proxy for various other characteristics of "real-world" software is flawed. While this may sound plausible to some readers, it should be backed up with data. Given the size of GitHub, this section uses sampling to answer the following questions: *Are starred projects a representative sample of all projects?* and *Are starred projects a representative sample of developed projects?* where what it means for a project to be developed is purposefully left open.

Since the later parts of this paper require Java, Python and JavaScript, we acquire samples of these three ecosystems. We use CodeDJ to do this. CodeDJ is an open source

project that can be forked and used to create a dedicated project database. CodeDJ ensures reproducibility of queries over the database and generates reproduction receipts for all queries used in this paper.

We used random sampling over the entire GHTorrent dataset to select which projects to acquire in each of the languages of interest. The number of downloaded projects is somewhat arbitrary as it is based on available hardware during the acquisition phase which began on April 1st, 2021. The datastore has 1,111,950 Java projects, 216,602 Python projects and 1,259,856 JavaScript projects (include source code). To give an idea of the scale, our Java dataset accounts for 20% of all non-forked GitHub Java projects. For simplicity, we down sampled further, randomly selecting 1Mio Java and JavaScript projects, and 200K Python projects.

4.1 Attributes

With CodeDJ, it is easy to write queries that compute project attributes. For this paper, we calculate 36 attributes for each project. From these, we select five attributes that highlight the differences between projects:

- **Age:** The age of a project is the number of day separating the first commit and the most recent commit. This correlates with the maturity of a project.
- **Devs:** The count of unique developer handles in the git logs; includes both the author of a code change and the committer of that change. Devs approximates the size of a team, of course some individuals may have more than one handle.
- **Locs:** The total number of lines in files that are recognized as code, in any language, and appear in the head of the default branch. Locs measures the active code in the project.
- **Versions:** A version is implicitly created for each commit touching a file, be that for insertion, deletion or update. This counts versions in the entire project's history including branches. Versions measure the activity in a project.
- **C-index:** A developer handle has a c-index of n if that developer was party to at least n commits to n projects (i.e. n^2 commits). The c-index of a project is the highest such number across developers. This measures developer expertise.

While we make no claims that these five attributes suffice to describe a software project, we have found them to be an effective summary of many interesting dimensions.

4.2 Stars v. All

What do these attributes tell us about the overall population and about starred projects? Fig. 1 is a histogram of attributes, the x-axis is log scaled and the y-axis is normalized for maximum height. In grey (background) is the whole population, red (Java) and blue (Python) are used for the 1K most starred projects.

The whole population is similar across languages. Most projects are young, with 49%/34% (J/P) of projects less than a week old, and with median ages of 7/46 days. Many projects are the work of a single developer, medians are 2/2 Devs. Most project are small, medians are 655/448 Locs. Median versions are 16/16. Finally, the C-index is low, with medians of 2/2.

Unsurprisingly, Fig 1 confirms that starred project, and in particular the top 1K, have a very different make up than the overall population of GitHub projects. Visually it is obvious that every distribution is shifted towards older and larger projects with more developers and these being more experienced. While there are slight differences between languages, the overall picture is consistent. The greatest shift is in project ages with medians of

2,581/1,440 days, i.e. many years old projects. C-Indices also increase, with medians of 19/15.5, suggesting that active developers tend to contribute to popular repositories. While many of them are team efforts, a significant portion has few contributors. Manual inspection revealed that any starred projects have been inactive for years. Project cannot "loose" stars, so if project gets to the top there is a chance it will stay there long past its useful lifetime.

We can now answer the first question by the negative. Starred projects do not yield a representative sample of the overall population. Now, this is not necessarily a bad thing, as folklore suggests that most of GitHub is uninteresting.

4.3 Stars v. Developed

Many researchers yearn for *engineered* [16, 20] or *developed* projects – informally, taken to mean projects that have been created with some care – alas there is little agreement on a precise definition. Slightly easier, perhaps, is to settle on what we don't want, the projects that are clearly of little value for any reasonable research question. Moreover, one could hope that the complement of uninteresting projects are the projects we want to analyze. Let us define a project that has less than 100 lines of code, fewer than 7 days old, and fewer than 10 commits as *uninteresting*. When this definition is used to filter projects, this rather low bar manages to eliminate 71% of Java and 55% of Python projects.

It would be handy if stars were a proxy for filtering out such uninteresting projects. Fig. 2 overlays the whole population (grey), the result of removing uninteresting projects (black) and the top 1K starred projects (red for Java, blue for Python). Sadly, developed projects do not align with stars. In terms of lines of code, developed projects have roughly the same distribution as the whole population but biased towards larger projects. Stars push the distribution much further. As for ages, our criteria filters out a large number of short lived projects, but stars skew significantly older.

Manual inspection of the starred project highlights their main issue – stars are extrinsic properties without a direct connection to any attributes of a project, and unlike attributes stars grow monotonically. Thus their meaning is unclear. Users award them for various reasons including humor and shock value. Some projects earned many stars because of a joke not fit for a research paper,[6] another has invalid code and a documentation daring users to star junk.[7] While these remarks might seem off-topic, they illustrate that stars do not correlate with quality or usefulness of repositories.

To further illustrate the limitation of stars as a filter, we take, for each attribute, the 20 lowest scoring Java and Python top starred projects. Table 2 has our manual classification. Arguably none of these projects is particularly useful: externals lack histories, widgets are small and biased by their application domain, babies are too small to yield much insights, and the others only have code snippets.

Fig. 2 answers our second question, developed projects are broadly similar in terms of distribution of attribute values as the whole population. For all attributes starred projects trend towards higher values. To summarize what we learned about stars, they capture extrinsic characteristics of GitHub projects and are at best indirect and noisy proxies for a robust frame oracle.

[6] `https://github.com/dickrnn/dickrnn.github.io`
[7] `https://github.com/gaopu/java`

4.4 How to select projects?

What to use for project selection if not stars? We argue that selection must be based on intrinsic features – measurable attributes of a project's contents or origin. While one may use machine learning [16, 20] to build classifiers, we propose to leverage the discriminative power of our five attributes as a frame oracle.

Fig. 3 is the cumulative density function of the various attributes for Java (Python is similar). The interpretation of each line is what percentage of the dataset is filtered for a particular attribute value. So for instance, if one were to use 10 days as a cutoff, then 52% of the Java set would be filtered out. Half of Java projects have been around for less than 10 days! What also stands out is that 82% of projects do not have any stars. A 10 star cutoff one filters out 98% of all projects. The discontinuity of C-Index at 65 is worrisome. After investigation, we found a single GitHub'developer' with such a high index, it turns out that it is a bot doing automated updates.

Project selection can be performed by a combination of attributes with cutoffs. We do not argue for a particular formula; researchers must make their own choices in this respect.

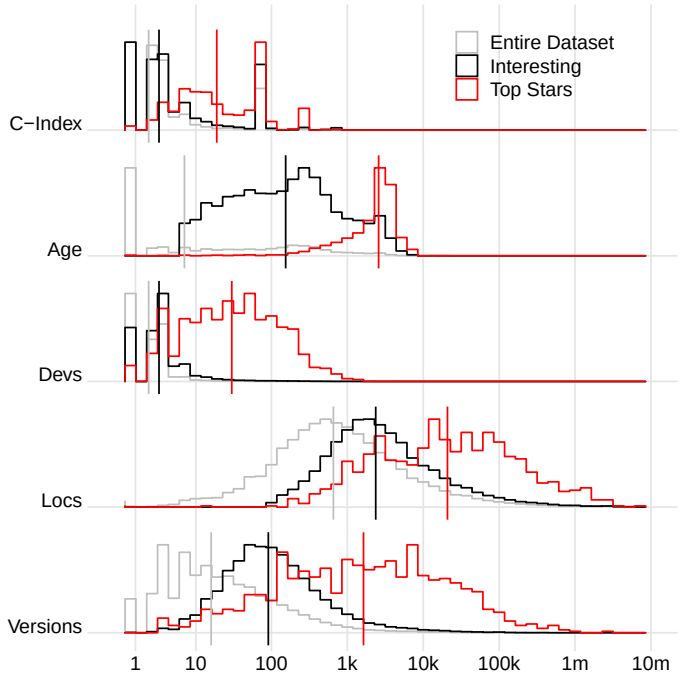


Figure 2 Comparing developed and starred projects

Category	Java	Python
Externals	9%	5%
Very few commits, likely from another repository, occasionally synchronized.		
Widgets	43%	0%
Tiny projects with little activity that implement popular UI widgets or plugins.		
Docs	4%	15%
Interview questions, code snippets, course materials, card games, knitting patterns.		
Tutorials	17%	9%
Educational materials, tutorials and example applications.		
Babies	16%	32%
Valid but extremely small projects with little activity.		
Artifacts	0%	21%
Artifacts for (mostly ML) research papers. Likely developed elsewhere.		
Deprecates	1%	5%
Deprecated projects, no code on the main branch.		

Table 2 Categorizing 200 starred projects

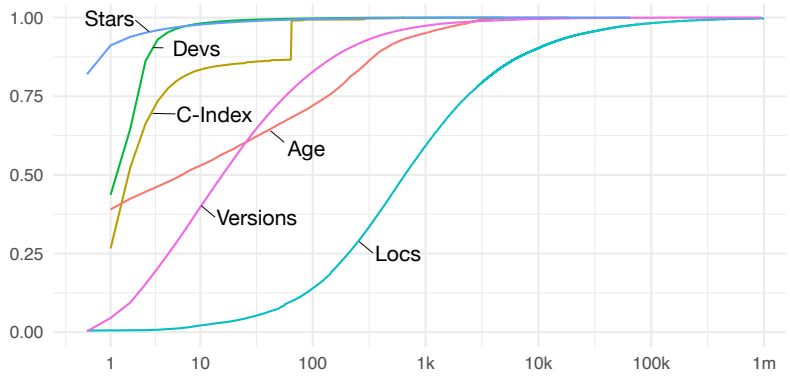


Figure 3 Cumulative Density Functions

4.5 Validity

Working with the dataset, we noticed an oddity around project ages. Experience with GitHub trained us to expect the unexpected. Our investigation started with a plot of creation dates. Fig. 4 shows the log scaled counts of new projects over time. While there is a steady progression in the count of projects created each year, we see a significant drop in 2015 and a plateau until 2019. We reviewed our pipeline to no avail. We use GHTorrent to acquire all available URLs. Then, we randomly sample projects from that list. We validated both acquisition and sampling. This leaves with two hypotheses. First is a consistent flaw in the CodeDJ downloader causing some projects to fail to download. 17% URLs obtained from GHTorrent point to dead projects, but there is no apparent bias. Second some projects could

be missing in GHTorrent. Again, we have not been able to eliminate this possibility. Another issue showed up on inspection, JavaScript project ages are significantly higher than those of other languages. We found that GitHub timestamps are frequently inconsistent, but why JavaScript be more affected? Until an explanation can be found, we removed JavaScript from the overall comparison and use JavaScript projects in the reproduction with extreme care.

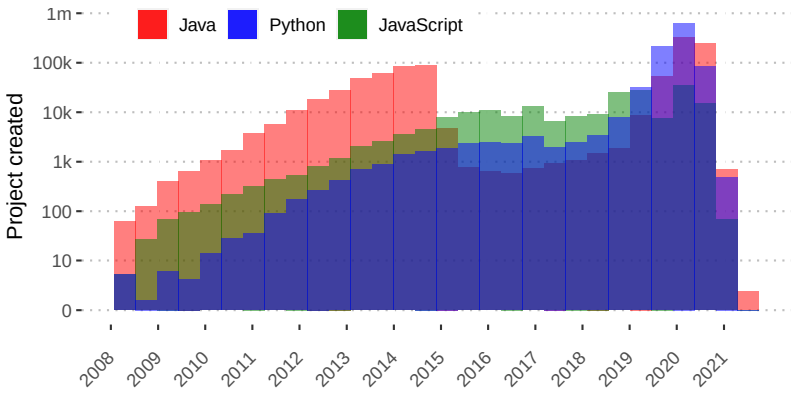


Figure 4 Creation date

5 Reproducible Experiment Design

This paper proposes that researchers conducting experiments over large-scale software repositories follow a specific experimental design methodology to ensure their work can be reproduced and increase chances their results generalize as expected. While the mechanics of reproducibility of the actual experiment itself vary, the setup of the experiment is a common problem. The proposed methodology has five steps, we encourage researchers to document each of these steps explicitly.

5.1 Population Hypothesis

Formulating a population hypothesis lets researchers stake a claim about the applicability of their work. This represent the population to which the result of an experiment should generalize to. The statement of that hypothesis can be brief and appeal to intuition, the other parts of the description flesh out the details. Ideally we would like our results to be as broadly applicable as possible, but pragmatically designing experiments that back up overly broad claims is difficult. Some populations of interest are difficult to sample, for instance "commercial software" is a relatively simple and unambiguous description but one that we typically can't sample from as most of the software is not in the public domain. Other populations can be difficult to identify. Imagine a study of the challenges linked to retraining imperative programmers to use functional idioms. Finding code written by such developers can be done manually but is difficult to automate. It is often easier to describe a population by intrinsic features of projects such as the language used to write the code or some estimate of the size of the project.

5.2 Frame Oracle

A frame oracle is a, possibly noisy, deterministic algorithm for deciding if a project belongs to the population of interest. The oracle is our best approximation of the population of interest. An executable and reproducible oracle allows to compare different papers with the same selection. The description of the oracle should specify the data source along with any information required to acquire projects. The procedure for evaluating a project should be clear and based on intrinsic attributes. A paper should at least have a short description of the oracle, full details should be given in the reproduction artifact.

5.3 Sampling Strategy

The literature has an abundant advice on sampling (see e.g. [10]). Briefly, a sampling strategy picks the type of sampling (probabilistic or non-probabilistic) and describe the high-level steps used to obtain a sample. The sampling implementation is expected to be found in the reproduction artifact. Many works use purposive or convenience sampling as it is simpler, cheaper and less time consuming. A better alternative is some form of probabilistic sampling as it is more likely to yield a representative sample. Probabilistic sampling can be staged if the structure of the population is more complex. The simplest approach is random sampling where each element has the same chance of being picked. We often have to resort to stratified sampling when the population is divided in subgroups of different sizes. Typically we sample without replacement as we do not want to pick the same project multiple times.

5.4 Validity

The validity section should argue, when there are reasons for doubt, why using the frame oracle and the sample strategy results in representative samples of the population. This section should also address potential sources of bias and attempts by the authors to control for them. This section should address any foreseen challenges to reproducibility and offer means to mitigate them.

5.5 Reproducibility Artifacts

The last components of our approach is to link the paper to a reproduction artifact that contains code and data to support experimental repeatability and reanalysis.

5.6 Tool support

Section 3 has listed specific issues with reproducibility. Roughly, there are two kinds of issues. The first is related to authors not being precise in their description of some of the steps outlined above. We believe that following the methodology as a template in the text of a paper and providing a reproduction artifact will greatly help. The second category of issues are more pragmatic, it is difficult to repeat the analysis of a paper because some aspect of the data used is not available. We suggest that research infrastructures should support the task by explicitly supporting experimental reproducibility. An example of such an infrastructure is CodeDJ which is both a continuously updated datastore and a database that can be queried by a DSL written in Rust. We have adopted that infrastructure for our work and illustrate how it helps with reproducibility.

The implementation of a frame oracle and the sampling strategy can be combined into a single expression. Fig. 5 shows a query which starts by filtering out projects with fewer

than 80% JavaScript code, then it uses pre-computed attributes Locs, Age and Devs to filter further. The last stage of filtering involves computing an attribute on the fly, here we sum up the commits in the project. Random sampling is implemented by calling the ş function.

```
database.projects()
  .filter(|project| {
      project.language_composition()
             .map_or(false, |languages| {
                languages.into_iter().any(
                  |(language, proportion)| {
                    language == Language::JavaScript
                        && proportion >= 80
          })
      })
  })
  .filter_by(AtLeast(Locs, 5000))
  .filter_by(AtLeast(Age,
                     Duration::from_months(12)))
  .filter_by(AtLeast(Devs, 2))
  .filter_by(AtLeast(Count(Commits), 100))
  .sample(Random(30, SEED)))
```

Figure 5 Project selection with CodeDJ

The architecture of CodeDJ is split between a persistent datastore in which every data item is timestamped with an insertion data, and an ephemeral database used to service queries. A reproducible query is a Rust crate archived in a git repository associated to the datastore. Running the query produces a *receipt* which is the hash of a commit automatically added to the archive repository. The receipt can be used to share the query (exactly as executed) and its results (exactly as produced). It can be used to retrieve the Rust crate and re-execute the code. Code re-execution is helped by the fact that queries are deterministic and the crate contains a list of all dependencies, a timestamp, and all random seeds. When a historical query is executed CodeDJ access the exact state of the datastore at the time the query was run. Since CodeDJ stores the contents of files, entire experiments can be fully reproduced.

6 Reproductions

We demonstrate the value of the methodology with examples. Results suggest that additional experiments are needed to validate some of the claims made in the reproduced works.

6.1 Reproducing: What is Software

This reproduction aims to validate two simple findings of [19]: (C1) sowftare is diverse, only 4% of repositories do not contain code, data and documentation; (2) documentation is an integral constituent of software, only 2% of repositories do not contain documentation. Our methodology is to start by a reproduction that attempts to follow the paper. Then we investigate if the results generalize to the intended population.

Population Hypothesis: The entire universe of software projects.

Frame Oracle: To understand the impact of project selection we consider three oracles. O1 accepts any software project hosted on GitHub. O2 is subset of O1 with uninteresting projects removed (as defined above). O3 uses a stronger filter, removing projects with fewer

than 500 commits, 180 days, or 10K Locs. We use GitHub language attribution to select a project's language.

Sampling Strategy: We report on four samples. S0 is a convenience sample of starred projects from O1 following [19]. S1, S2 and S3 are random samples without replacement from O1, O2 and O3 respectively, stratified by language. Projects with duplicate contents are removed.

Validity: Our reproduction differs in the number of languages (3 v. 25) and by categorizing files based on the file path alone. We tested stability of our results with multiple samples of varying sizes and manually inspected the produced labels.

Reproduction Artifact: Our artifact has CodeDJ receipt for this query.

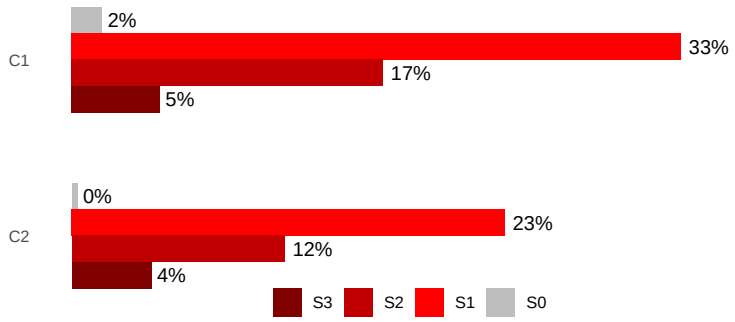


Figure 6 Content of software projects

Results

Fig. 6 shows the results of reproduction for claims C1 and C2. Compare the percentages between S0 (original) and S1 (target population). Statistical analysis is not required to see that the difference is significant. The samples S2 and S3 are there to illustrate the impact of slightly more developed populations, but even these are still quite different. Would the results agree if we included more languages? The three languages we downloaded account for most of GitHub, it is conceivable that other languages could affect results, but that would just push the generalizability issue somewhere else as the claims would become language-specific.

6.2 Reproduction: Method Chaining

[18] claim that 50% of projects in 2018 had method chains longer than 7 while in 2010 that number was 42%, and more generally they observed longer chains at all lengths. They state that "chains of length 8 are unlikely to be composed by programmers who tend to avoid method chaining, this result is another supportive evidence for the widespread use of method chaining.'

We intend to reproduce the authors methodology, and then compare to various samples that might represent that the authors expresed as their population of interest in their paper.

Population Hypothesis: The universe of real-world Java programs.

Frame Oracle: We accept any Java project hosted on GitHub and delegate to GitHub for language attribution.

Sampling Strategy: Stratified sampling to randomly select projects with commits in 2010 and 2018.

Validity: To reproduce the original results, we performed stratified sampling to get top starred projects active in the target years. The authors used a different sample of top stars. The original paper had different sample sizes for each year, but those are not specified. We fix the sample size to 250. The authors could not locate the code of their chain detector, so we use our own implementation.

Reproduction Artifact: The selection of input is represented by a CodeDJ receipt in the artifact.

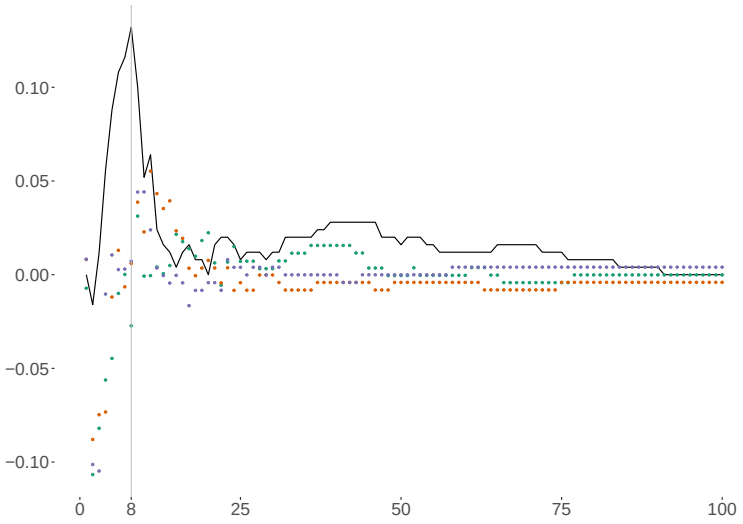


Figure 7 Difference in chain lengths

Results

Fig. 7 shows the difference in proportion of projects at various chain lengths. The solid line uses stars, colors represent different random samples. For instance, if we pick chains of length 8, the number used by [18], the difference is a 13% increase in the number of projects between 2011 and 2018. The differences for our random samples are -2%, 0.6% and 0.7%. This particular population does not seem to show the effect expected by the authors. We surmise that some notion of developed project may show more favorable results, but without more guidance in the population hypothesis it is hard to guess which to pick.

6.3 Reproduction: Style Analyzer

[14] build model of the style of a repository and apply this model on a held-out part of that repository to produce corrections. Their experiment uses 19 top-starred JavaScript project to gauge the precision with which the tool flags formatting discrepancies and the relationship between this precision and the size of the project. They report a precision of 94% (average, weighed by project size) and better overall performance for large projects and projects with better style guidelines.

We investigate how different project samples impact these conclusions. For the reproduction we attempted to create samples to fulfill our intuition about the original's paper intentions as best we understand them from the conclusions they draw.

Population Hypothesis: A developed JavaScript projects.

Frame Oracle: Our oracle picks JavaScript projects such they contain at 80% JavaScript code (files), Loc ≥ 5000, Age $\geq 12 * 31$ and Devs ≥ 2.

Sampling Strategy: We randomly select 10 sets of 30 projects. We select more projects than the original sample to account for errors in processing. After processing is finished, we randomly select 19 out of the pool of successfully processed projects in each selection.

Validity: Given the complexity of the tools configuration and the fact that it is missing from the artifact, we used a default configuration provided by the tool. This produces an average increase in project size by 46% per project (up to a maximum of 154%) and causes precision to diverge by 2.2% on average, and up to 7.9%.

The tool failed to process 4 projects: `freecodecamp` and `atom` due to errors in unicode processing, `express` due to a programming bug, and `30-seconds-of-code` probably due to bad file identification. Three of the missing projects were located close to the median in terms of precision, prediction rate, and project size in the original paper, while `axios` was in the lower quartile for sample count. We reproduced the study using 19-project samples.

Style analyzer analyzes each project at two points in its history specified by a base commit and a head commit. The base commit is a point in the past which the tool checks out to learns the project's formatting style. The head commit is a more recent point used to evaluate the model and calculate precision. The original paper provides head and base commits for each project in their experiment, but does not specify the method of selecting these commits. We pick the current head of the default branch as the head commit. For base commit we pick one that lies at an offset equal to 10% of the number of all commits in the default branch from the head commit. This retrieves different commits than the original paper, which causes a 3.1% median change in precision (up to 17%—`telescope`) and a median project size increase of 76%, and up to 311% (`reveal.js`).

Reproduction Artifact: Datasets, receipts from submitted Djanco queries, style analyzer's reports and scripts for the entire experimental pipeline are included in the artifact.

Results

We recreate a plot of the effect of the number of items in the training set on precision from the original paper in Fig. 8. The training set consists of snippets created around tokens/AST nodes relevant to formatting (whitespace, indentation, quotes, zero-length gaps). We plot the selection from the original paper along three selections from our interesting project frames. In addition, we plot the distributions of precision in each selection in Fig. 9. We compare the precision scores in each sample with the selection used in the original paper using a Mann-Whitney U test to show which samples performed statistically differently from the

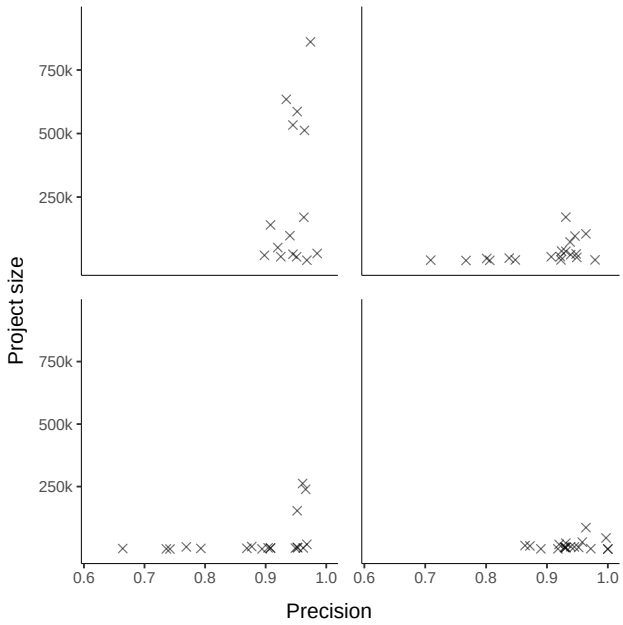


Figure 8 Relationship between label groups and precision

original. The scatter plots show a different grouping of results from the original paper. The groupings in the scatter plot visibly differ between selections. The distribution comparison shows that our selections generate significantly smaller training sets in all cases and yield lower precision. In addition, 6 out of the 10 interesting project selections produced significantly lower precision, with the remainder producing a statistically equivalent distribution.

Overall, we see our selections yielding precision between 0.91 and 0.95 (the paper sets a precision of 0.95 as a benchmark for success). We also do not see a clear relationship between the number of label groups and precision, such as the one the authors note in the original paper.

6.4 Reproduction: Code Smells

We seek to validate the claim of [8] that for large and small projects there is a statistical difference in the occurrence of code smells, whereas medium sized projects are indistinguishable.

Population Hypothesis: Mature GitHub projects in all application domains including machine learning written in Python.

Frame Oracle: Projects with C-Index ≥ 5, or Age ≥ 180, or Locs ≥ 10000, or Versions ≥ 100. We delegate to GitHub for language attribution.

Sampling Strategy: The deep learning projects were provided by the authors. Out of 59

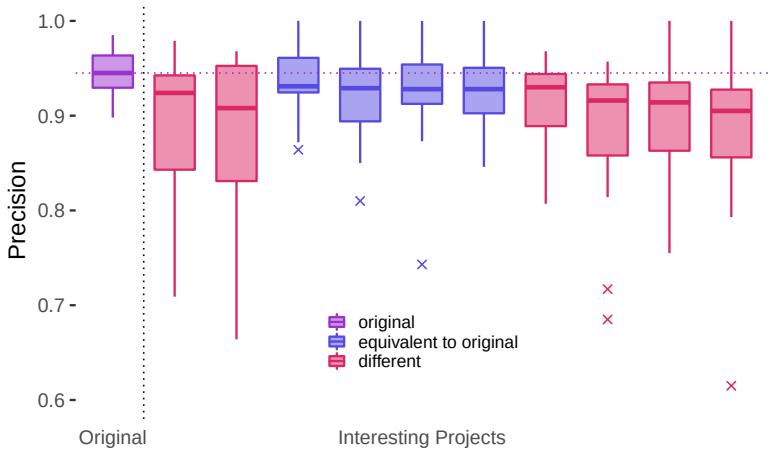


Figure 9 Comparing label group count and precision

projects, 57 were still accessible on August 2nd 2021. At download time there were 6 small, 13 medium, and 38 large deep learning projects. For the reproduction of the original results, we used a staged strategy, first convenience sampling the top starred Python projects and amongst those used stratified sampling to select 57 projects with a similar distribution of sizes. To generalize the results we used quota sampling to match the size distribution.

Validity: Our reproduction uses the Locs reported by CodeDJ. The date the authors downloaded the repositories is unknown. We use the content of the main branch of each repository as of April 1st, 2020. The authors say "*each of repositories is pre-processed and prepared for code smell detection*", however details are missing. We used the default thresholds of their tool.

Reproduction Artifact: A CodeDJ receipt is included in our reproduction package along with code to run the experiment.

Results

Fig. 10 contrasts the distribution of smell for deep learning projects, top stars, and three random samples. Computing the p-values with the non-parametric Mann-Whitney Wilcoxon shows that one of the samples disagrees on small projects (i.e. the difference is not statistically significant) and two of the samples disagree on the large projects (again the difference is not significant). Generalizability of the results is thus questionable.

7 Conclusions

Sometimes doing it wrong is so much easier than the alternative, that we convince ourselves that the wrong is right enough.

Our paper is unusual. While it purports to contain a call to arms for better experimental practices, it is just as much a record of our own journey to that goal. What reads as

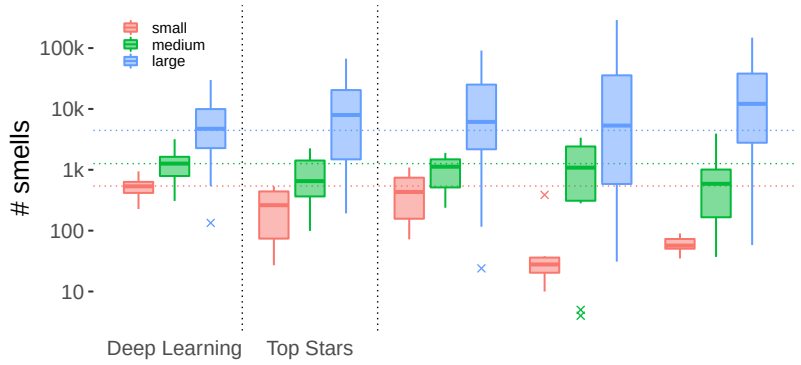


Figure 10 Comparing Smells

criticism was just as likely written in self-reflection. So, what can a researcher in the field take away from this paper? There are three ideas we would like to leave you with regarding *generalizability*, *reproducibility* and *tooling*.

Generalizability. The value of an experiment often lies as much in what it generalizes to, as in the experiment's outcome. We found that many researchers rely on GitHub stars to pick representative samples of software projects, yet starred projects tend to be larger in most dimensions than typical ones, also that they are more likely to be inactive, and that their ranking is not a measure of intrinsic qualities of the code. Hopefully, this paper is the last nail in that coffin. More generally, we advocate for the use of probabilistic sampling over populations defined by intrinsic attributes of software, and also for clear and standardized documentation of experimental design.

Reproducibility. The value of a scientific experiment also lies in our ability to reproduce it. Carrying out reproducible experiments over large-scale software repositories is hard. Especially when aiming to support the three reproduction modalities: repetition, as practiced in artifact evaluation, where an artifact is re-executed to obtain identical results; reanalysis, where the artifact or its input are modified; and independent reproduction, where the entire experiment is re-implemented from scratch. The first modality requires faithful replay and is best served if all data used is included with the artifact. The second, requires support for automatically acquiring new representative samples. The third needs an unambiguous description of all experimental steps. We advocate for reproductions artifacts that supports the first two modes, and a detailed description of the experiment for the last.

Tooling. Generalizability and reproducibility, while worthy goals, represent much work, and they are work that is orthogonal to the scientific goals of researchers. The only reasonable answer is to provide tooling that automates acquisition of representative samples and generation of reproduction artifacts. In this paper, we used `CodeDJ` and found it helpful as it let us specify queries over attributes of the code for many projects, while also supporting experimental repetition and reanalysis through historical queries. It has its limitations, we found execution times to be somewhat long and doubt it will scale to the whole of GitHub.

Our vision for a bright and shiny future is one where the community agrees on standard

tools and techniques for this kind of experiment, tools which automate the acquisition and packaging of input datasets and the re-execution of entire experiments.

References

1. S Baltes and P Ralph. Sampling in software engineering research. *CoRR*, 2020. URL: https://arxiv.org/abs/2002.07764.
2. H Borges and M Tulio Valente. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 2018. doi:10.1016/j.jss.2018.09.016.
3. Z Chen et al. Understanding metric-based detectable smells in python software. *Information and Software Technology*, 2018. doi:10.1016/j.infsof.2017.09.011.
4. V Cosentino, J Izquierdo, and J Cabot. Findings from GitHub: Methods, datasets and limitations. In *Mining Software Repositories (MSR)*, 2016. doi:10.1145/2901739.2901776.
5. R Dyer, H Nguyen, H Rajan, and T Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Int. Conf. on Software Engineering (ICSE)*, 2013. doi:10.5555/2486788.2486844.
6. G Gousios and D Spinellis. GHTorrent: GitHub's data from a firehose. In *Mining Software Repositories (MSR)*, 2012. doi:10.1109/MSR.2012.6224294.
7. J Han et al. Characterization and prediction of popular projects on GitHub. In *Computer Software and Applications Conf. (COMPSAC)*, 2019. doi:10.1109/COMPSAC.2019.00013.
8. H Jebnoun et al. The scent of deep learning code. In *Mining Software Repositories (MSR)*, 2020. doi:10.1145/3379597.3387479.
9. E Kalliamvakou et al. The promises and perils of mining GitHub. In *Mining Software Repositories (MSR)*, 2014. doi:10.1145/2597073.2597074.
10. S Lohr. *Sampling: Design and Analysis*. 2010.
11. C Lopes et al. Déjà Vu: A map of code duplicates on GitHub. *Proc. ACM Program. Lang.*, (OOPSLA), 2017. doi:10.1145/3133908.
12. Y Ma et al. World of code: enabling a resarch workflow for mining and analyzing the universe of open source vcs data. *Empirical Softw. Eng.*, 2021. doi:10.1007/s10664-020-09905-9.
13. P Maj et al. CodeDJ: Reproducible queries over large-scale software repositories. In *European Conf. on Object-Oriented Programming (ECOOP)*, 2021. doi:10.1145/2658987.
14. V Markovtsev et al. Style-analyzer: fixing code style inconsistencies with interpretable unsupervised algorithms. In *Mining Software Repositories (MSR)*, 2019. doi:10.1109/MSR.2019.00073.
15. T Mattis, P Rein, and R Hirschfeld. Three trillion lines: Infrastructure for mining github in the classroom. In *Conf. on Art, Science & Eng. of Programming <Programming>*, 2020. doi:10.1145/3397537.3397551.
16. N Munaiah et al. Curating github for engineered software projects. *Empirical Software Engineering*, 2017. doi:10.1007/s10664-017-9512-6.
17. M Nagappan, T Zimmermann, and C Bird. Diversity in software engineering research. In *Foundations of Software Engineering (FSE)*, 2013. doi:10.1145/2491411.2491415.
18. T Nakamaru et al. An empirical study of method chaining in Java. In *Mining Software Repositories (MSR)*, 2020. doi:10.1145/3379597.3387441.
19. R Pfeiffer. What constitutes software? In *Mining Software Repositories (MSR)*, 2020. doi:10.1145/3379597.3387442.
20. P Pickerill et al. Phantom: curating github for engineered software projects using time-series clustering. *CoRR*, 2019. URL: http://arxiv.org/abs/1904.11164.
21. P Ralph et al. Empirical standards for software engineering research. *CoRR*, 2020. URL: https://arxiv.org/abs/2010.03525.

Chapter 5

Conclusions

The wealth of data available in large software repositories offers unprecedented opportunities for analyzing source code and software development patterns. Due to the sheer size and noise present in those repositories, careful project selection and filtering is a crucial step for each such analysis. The contributions of this thesis lie in improving our ability to select software projects precisely and reproducibly at scale. We introduce CodeDJ, an infrastructure for the maintenance and querying of a local mirror of software projects that provides precision in the project selection criteria, scalability in the number of software projects, and reproducibility in the presence of frequent updates to both the stored projects and data types kept over time, including historical accuracy which allows querying an updated database in the future as if the query happened on the dataset as it looks now.

The details of these contributions can be found in the four research papers that form the bulk of the thesis, namely:

1. *A Map of Code Duplicates on GitHub* [A.3] analyses source code clones present in GitHub projects. It verifies the existence of one of the most common biases and shows its scale. Our findings signify the necessity for dedicated project selection and filtering steps in big code analyses.

2. *On the Impact of Programming Languages on Code Quality* [A.2] is a reproduction study focusing on the data filtering, reproducibility, and statistical interpretation of large corpora analyses. The paper shows the problems pointed out by this thesis are present in contemporary research and that they affect our results.

3. *Reproducible Queries over Large-Scale Software Repositories* [A.1] introduces the infrastructure that forms the statement of this thesis: a scalable, precise, deterministic, up-to-date and reproducible project selection pipeline.

4. *How to Design Reproducible Large-scale Code Analysis Experiments* [A.4] then devises and argues for an explicit and rigorous project filtering step and demonstrates how it can be done with the tool presented in the previous paper.

5.1 Future Work

Although the tool and dataset as provided are immediately usable, further improvements are planned along the following main lines:

5.1.1 Increasing the dataset size

The actual GitHub scrapper processes projects based on their main programming language and new languages are added as needed. Aside from new projects, the dataset should also grow in the types of data it stores, such as discussions, continuous integration results, releases, etc. As this information comes at the cost of extra GitHub API requests, which is a scarce resource, these too, will be added on a needed basis.

We have already demonstrated scalability of the solution to millions of projects, but given the enormous size of GitHub, it may be that extra effort will be necessary in the future to maintain scalability.

Finally, software repositories other than GitHub can be added as sources to CodeDJ in the future. This would require adapting the sources to the `git` based terminology of CodeDJ for source code contributions, and either mapping the metadata to the items known and already downloaded from GitHub, or simply adding new categories.

5.1.2 Improving querying capabilities

CodeDJ enforces a split between the datastore management and updates and the querying itself. It provides only a simple programmatic API that can be used for parallel random access and linear scanning of the stored information. Djanco, the proof of concept querying engine provided in the paper [A.1] is very expressive and user friendly, but as it does not support parallelism, has limits in terms of scalability. To remain competitive with increasing Parasite dataset sizes, Djanco needs to be optimized. Addition of other, perhaps less expressive, but simpler and faster querying front-ends to complement Djanco would be a welcome improvement.

5.1.3 Finding more uses

Having access to a large database of software projects and their attributes is useful not only for project selection and sampling. The author of this thesis is currently working on an automated detection of implicit clones, a tool that for any given GitHub project uses the Parasite database to scan its contents for duplication and then reports any possible problems, such as outdated copy, etc. More uses are possible and should be explored.

Bibliography

[1] T. F. Bissyande, F. Thung, D. Lo, L. Jiang, and L. Reveillere. Orion: A software project search engine with integrated diverse software artifacts. In *International Conference on Engineering of Complex Computer Systems*, 2013.

[2] Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In *iPRES 2017: 14th International Conference on Digital Preservation*, Kyoto, Japan, 2017.

[3] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, oct 1972.

[4] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *International Conference on Software Engineering (ICSE)*, 2013.

[5] Davide Falessi, Wyatt Smith, and Alexander Serebrenik. Stress: A semi-automated, fully replicable approach for project selection. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017.

[6] Jesus M. Gonzalez-Barahona, Gregorio Robles, and Santiago Dueñas. Collecting data about FLOSS development: The FLOSSMetrics experience. In *International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS)*, 2010.

[7] Statista inc. Programming languages that are associated with the highest salaries worldwide in 2022. `https://www.statista.com/statistics/1127190/programming-languages-associated-highest-salaries-worldwide/`.

[8] Github LLC. The 2021 state of the octoverse. `https://octoverse.github.com/static/octoverse-report-2021.pdf`, 2021.

[9] Github LLC. The 2022 state of the octoverse. `https://octoverse.github.com/2022/how-companies-invest-in-open-source`, 2022.

[10] Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, Breno Bernard Nicolau de FranC ca, Carlo Alberto Furia, Greg Gay, Nicolas Gold, Daniel Graziotin, Pinjia

He, Rashina Hoda, Natalia Juristo, Barbara Kitchenham, Valentina Lenarduzzi, Jorge Martínez, Jorge Melegati, Daniel Mendez, Tim Menzies, Jefferson Molleri, Dietmar Pfahl, Romain Robbes, Daniel Russo, Nyyti Saarimäki, Federica Sarro, Davide Taibi, Janet Siegmund, Diomidis Spinellis, Miroslaw Staron, Klaas Stol, Margaret-Anne Storey, Davide Taibi, Damian Tamburri, Marco Torchiano, Christoph Treude, Burak Turhan, Xiaofeng Wang, and Sira Vegas. Empirical standards for software engineering research, 2020.

[11] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A large-scale study of programming languages and code quality in github. *Commun. ACM*, 60(10):91–100, sep 2017.

[12] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 155–165, New York, NY, USA, 2014. Association for Computing Machinery.

[13] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, 1975.

[14] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: scaling code clone detection to big-code. In *International Conference on Software Engineering (ICSE)*, 2016.

Reviewed Publications of the Author Relevant to the Thesis

[A.1] P. Maj, K. Siek, A. Kovalenko, J. Vitek. CodeDJ: Reproducible Queries over Large-Scale Software Repositories. In *35th European Conference on Object-Oriented Programming (ECOOP)*, 2021.

[A.2] E.D. Berger, C. Hollenbeck, P. Maj, O. Vitek, J. Vitek. On the impact of programming languages on code quality: a reproduction study. *ACM Transactions on Programming Languages and Systems (TOPLAS), 41*(4), 1-24, 2019.

[A.3] C. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, J. Vitek. DéjàVu: A Map of Code Duplicates on GitHub. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2017.

Submitted Publications of the Author Relevant to the Thesis

[A.4] P. Maj, S. Muroya, K. Siek, J. Vitek. The Fault in Our Stars: How to Design Reproducible Large-scale Code Analysis Experiments. Submitted to *37th European Conference on Object-Oriented Programming (ECOOP)*, 2023.

Remaining Publications of the Author Relevant to the Thesis

[A.5] E.D. Berger, P. Maj, O. Vitek, J. Vitek. SE/CACM Rebuttal [2]: Correcting A Large-Scale Study of Programming Languages and Code Quality in GitHub. arXiv preprint arXiv:1911.11894, 2019.

[A.6] P. Maj, C. Hollenbeck, S. Hussain, J. Vitek. Analyzing Duplication in JavaScript. In *BenchWork*, 2018.

[A.7] P. Maj, F. Gauthier, C. Hollenbeck, S. Hussain, J. Vitek, C. Cifuentes. Building a node.js Benchmark: Initial Steps. In *BenchWork*, 2018.

[A.8] P. Maj. Analyzing Large Code Repositories. Ph.D. Minimum Thesis, Faculty of Information Technology, Prague, Czech Republic, 2018.

Remaining Publications of the Author

[A.9] J. Sliacky, P. Maj. Lambdulus: teaching lambda calculus practically In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E, pages 57-65*, 2019.

[A.10] T. Kalibera, P. Maj, F. Morandat, J. Vitek. A Fast Abstract Syntax Tree Interpreter for R. In *Conference on Virtual Execution Environments (VEE)*, 2014.

[A.11] P. Maj, T. Kalibera, J. Vitek. TestR: R language test driven specification. In *The R User Conference, useR!*, 2013.

[A.12] T. Kalibera, J. Hagelberg, P. Maj,F. Pizlo, B. Titzer, J. Vitek. A family of real-time Java benchmarks. In *Concurrency and Computation: Practice and Experience*, 2011.

[A.13] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, J. Vitek. High-level programming of embedded hard real-time devices. In *Proceedings of the 5th European conference on Computer Systems, pages 69-82*, 2010.

[A.14] F. Pizlo, L. Ziarek, P. Maj, A.L. Hosking, E. Blanton, J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. *ACM Sigplan Notices, 45*(6), pages 146-159, 2010.