Characterizing the Roles of Contributors in Open-source Scientific Software Projects

Reed Milewicz
Sandia National Laboratories
Email: rmilewi@sandia.gov

Gustavo Pinto
Federal University of Pará
Email: gpinto@ufpa.br

Paige Rodeghero
Clemson University
Email: prodegh@clemson.edu

***Abstract*—The development of scientific software is, more than ever, critical to the practice of science, and this is accompanied by a trend towards more open and collaborative efforts. Unfortunately, there has been little investigation into who is driving the evolution of such scientific software or how the collaboration happens. In this paper, we address this problem. We present an extensive analysis of seven open-source scientific software projects in order to develop an empirically-informed model of the development process. This analysis was complemented by a survey of 72 scientific software developers. In the majority of the projects, we found senior research staff (e.g. professors) to be responsible for half or more of commits (an average commit share of 72%) and heavily involved in architectural concerns (seniors were more likely to interact with files related to the build system, project meta-data, and developer documentation). Juniors (e.g. graduate students) also contribute substantially — in one studied project, juniors made almost 100% of its commits. Still, graduate students had the longest contribution periods among juniors (with 1.72 years of commit activity compared to 0.98 years for postdocs and 4 months for undergraduates). Moreover, we also found that third-party contributors are scarce, contributing for just one day for the project. The results from this study aim to help scientists to better understand their own projects, communities, and the contributors' behavior, while paving the road for future software engineering research.**

I. Introduction

Computing technologies have had a profound impact on the practice of science: simulation and data-intensive computation are now known as the third and fourth paradigms of science, on equal footing with experimentation and theory [1]. This shift has accelerated the growth of a diverse ecosystem of scientific software projects. The term "scientific software" is an umbrella that covers all aspects of the research pipeline, including codes for simulation and data analysis, dataset management, communication infrastructure, and underlying mathematical libraries [2]. It is software that exists "to support the exploration of a scientific question" [3].

What makes scientific software projects different from traditional software projects? Scientific software operates at the boundaries of human knowledge and tends to be in constant flux as new insights motivate unforeseen changes in requirements [4]. As noted by Segal [5], this pressing need to produce or enable the production of knowledge lends itself to a mindset where "software is valued only insofar as it progresses the science", often in conflict with the need to have reliable, maintainable code. However, Turk and colleagues remarked that, in an era of increasing scale and complexity, "the cyber-infrastructure necessary to address problems in computational science is no longer tractably solved by individuals working in isolation" [6]; broader, more open collaboration necessitates a shift in how the software is developed. From a software engineering research perspective, this motivates important questions about how the software evolves, who develops it, and how quality can emerge from this process.

We focus on the people meeting the demand for scientific software. Such scientific software developers represent a population so far not properly understood, since their characteristics, motivation, and needs to contribute to scientific software projects are intrinsically different than what drives traditional open source contributors. For instance, the actors that play the scientific developer role include students, postdocs, faculty, and staff. Their knowledge, skills, and goals can vary greatly, while also contributing to projects in different ways throughout their tenure. As a consequence, the plethora of existing studies on open source contributors might not help much, since they hardly take into account their roles or the complexity of the domains that scientific software is immersed in.

Much is still unknown about the state-of-the-practice of developing scientific software. For instance, who performs the majority of commit activities? Who fixes bugs? In order to better understand the relationship between these contributors and the software, we first leverage the availability and transparency of social coding websites to inspect data related to source code contributions and contributors. We selected a curated list of seven open-source scientific software projects by searching three different platforms: the Journal of Open Source Software, GitHub, and DOECODE, a platform for publicly funded DOE research codes. For each selected project, we identified the roles played by different contributors by analyzing each projects' documentation, websites, and other readily available sources. We then surveyed representative scientific software developers in order to cross-validate the findings found via the repositories' analysis.

Using quantitative and qualitative data, our study produced a set of findings, some of which confirmed anecdotal accounts

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. SAND2018-9345C.

while others were unexpected. We discuss them in detail in Section IV. In the following, we highlight three of them.

- **Senior researchers tend to be the most active and prolific contributors in terms of commits and file creation.** In four of the seven projects we studied, faculty and staff contributors were responsible for half or more of commits made to the project (with an average commit share of 72%). In five projects, senior members were also responsible for the majority of files created and, by that measure, the resulting project structure. This influence over the overall direction of the software project was also evident in the fact that senior researchers were the most likely to have interacted with files related to the build system, project metadata, and developer documentation.
- **Junior contributors, especially graduate students, are critical drivers of new features as well as supporting activities like test creation**. On average, junior contributors were responsible for 42% of commits across all projects we studied; in one case, juniors were responsible for nearly 100% of all commit activity. The majority of these commits came from graduate students, who had the longest contribution periods among juniors (with 1.72 years of commit activity compared to 0.98 years for postdocs and 4 months for undergraduates). Similar to senior contributors, junior contributors are significantly involved in creating new features, improving existing capabilities, and fixing bugs.
- **An open-source model facilitates external contributions, but the results are mixed.** On one hand, an open-source model makes it easier to attract thirdparty contributors to help grow and maintain the software. However, the software is also made for and by members of a relatively niche and intensely preoccupied community. In the majority of projects we studied, thirdparty contributors tended to be domain expert users who were only active for one day. We also note, however, that these same contributors are more likely to offer defect-correcting commits, which is highly valuable.

II. Background

Scientific software projects are very complicated undertakings that have limited budgets, sometimes a lack of software development expertise, and the inherent complexity of the domain [2].

Senior researchers and staff. As is common in the sciences, a typical software project coalesces around a principal investigator (PI) and one or more co-investigators (Co-Is) who have secured the resources needed for development (e.g. time, money). The reasons for developing software are varied, but include the use-value of the software as a vehicle for research, academic credit, and (in the case of commercial software) revenue [7]. However, it is well-known that the scientist-as-software-developer rarely has the time to maintain the code that they write [8]. Time and energy must be divided between writing papers and grant proposals, reviewing manuscripts, mentoring, and conducting experiments *et cetera*. Hannay et al. found while 84% of interviewees considered developing scientific software important for their own research, the average scientist spent just 30% of their work time on development activities [9].

Junior researchers. In order to meet the labor needs for development, established researchers often rely upon student and post-graduate labor; juniors are seen as young, full of ideas, and (most importantly) inexpensive personnel [10]. According to Heroux 2017, senior members provide a stable presence, determining the scientific questions and the trajectory for the software; they are familiar with the conceptual models and the software design, but they may spend less time writing actual code. Juniors, meanwhile, are transient members with a dual focus on contributing code and producing publications; they undergo a staged process of onboarding, becoming experienced, and departing, and during this time they may make substantial contributions to the software [11].

Third party contributors. For every developer of a critical scientific software package, countless more depend upon it. However, unlike in conventional software development, there are no clear-cut distinctions between users and developers, and, as Turk 2013 argues, trying to force these terms is "actively harmful" to our understanding [6]. Even when they are not directly responsible for a package, it is not uncommon for scientific end-users to write code of their own, such as "glue code" that draws together different software tools into a workflow or custom components that utilize others' software.

The decision to use another's software creates risks because it is not guaranteed that the software will be supported in the future or kept current with the pace of changes in the field [12]. Converting users into contributors is perhaps even more difficult. However, as Bangherth and Heister 2013 explains, scientific software libraries require the support of a broader community of users and contributors in order to survive in the long term [13].

III. Methodology

In this section, we describe our research questions and our approach (Section III-A). For the repository mining portion of our work, we outline our data collection methodology (Section III-B), the corpus we have assembled in order to find the answers to our research questions (Section III-D), and how we distinguish contributors' roles (Section III-E). For our the survey section of this work, we describe our protocol (Section III-F).

A. Research Questions

In this paper, we characterize the habits of scientific software contributors and contributions. Some of our questions are intended to test common wisdom, while others are aimed to probe deeper into the relationships between project contributors. Our research questions are as follows:

RQ1: What is the tenure of different contributors by role on a scientific software project?

RQ2: What is the breakdown of team contributors by role?

RQ3: How much of the development work is done by different contributors by role?

RQ4: What kinds of software maintenance and evolution activities do contributors perform?

RQ5: How do scientific software developers perceive their own software development process?

Our first question **RQ1** is demographic in nature based on aggregated project data, and tests the representativeness of our dataset. **RQ2** enables us to make inferences about the division of labor based on personnel composition. Next, **RQ3** digs into the kinds of responsibilities, such as file ownership and test files creation, that different contributors take up. **RQ4** investigates what kind of maintenance and evolution changes, such as adding new features or fixing bugs, do these contributors contribute to the project. Finally, to provide answers to **RQ5** we surveyed 72 scientific software developers regarding their own contribution behavior.

B. Data Collection Procedure

Figure 1 depicts the steps followed by our data collection procedure.

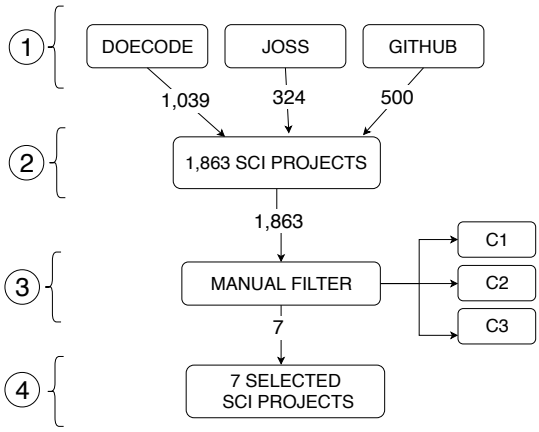

Fig. 1: Steps of the data collection procedure.

The first step (1) is aimed to find representative projects. We relied upon three data sources. First, we consulted DOECODE[1], a platform for publicly funded DOE research codes. Next, we searched the Journal of Open Source Software (JOSS)[2], a database of open source research software [14], which requires all entries be publicly available. Finally, we did searches by topic on GitHub to find repositories with relevant tags (e.g., `computational-neuroscience`, `bioinformatics`). This yielded roughly 1,039 repositories from DOECODE, 324 from JOSS, and another 500 from GitHub. These numbers corresponds all projects in these platforms, except for GitHub, in which we stopped searching when we found 500 projects. This resulted in an initial set of 1,863 open source scientific software projects which we chose to take into consideration (step (2)).

[1]https://www.osti.gov/doecode/

[2]https://joss.theoj.org/

From this list, we manually analyzed these repositories over several days (step (3)), filtering the results according to the following criteria:

C1) **Projects should have a contributor list.** The repository must link to a detailed contributor list or research team page that identifies the roles played by different contributors to the project. We use this data to later distinguish contributors' roles (Section III-E).

C2) **Projects should be active.** The project must be at least a year old, and the repository must have more than 500 commits. For example, a large number of projects on DOECODE were developed internally and then later released to the public. Thus, the GitHub repository is a shallow copy of the most recent version with no commit history.

C3) **Projects should be collaborative.** There must be at least three contributors which can be positively identified, and at least one these must be considered a "junior" contributor. Many research projects on GitHub are small codes developed by individual researchers in isolation without any significant collaborations with others. Others are collaborative projects between senior staff at different institutions.

After applying these filters, we ended up with a curated list of seven scientific software projects (step (4)).

C. Characterizing the population

We believe that the 1,863 projects in our population of repositories we is a representative sample of scientific software projects that can be found in the wild. However, many of these projects are unlikely to provide useful information for our purposes, such as short-lived or single-user research codes, snapshots of codes released for publications, untouched clones of decades-old legacy projects, or mirrors of private repositories lacking history information. As shown on Figure 2, filtering for the number of commits and contributors eliminates roughly 8 out 10 of the repositories; the remainder are most likely to be active, collaborative, and (most importantly) to have a rich history on GitHub that we can pull apart. The median project within this group projects has 12 contributors and 1770 commits spread out over 2.93 years.

We can examine a sample of 5000 of the contributors to these repositories, using their number of commits and the number of days spanned by those commits as a proxy for involvement. 29% of these contributors make only one commit, and 40% are active for no more than a day. Among contributors more active than these, the average individual has an active tenure of 1.37 years, and during this time they make 116 commits (6% of the commit activity of the median project). If we zoom in on any one of these contributors, we can observe their activities, but what interests us most is how their identity relates to those activities. This is a more challenging problem to solve, and analysis is much less scalable. Instead, in this work we take a deep dive into a handful of projects whose members we can identify, as a case study into the composition of these teams.

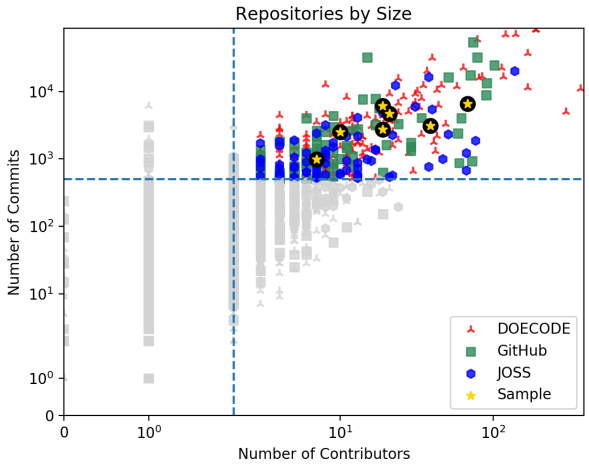

Fig. 2: A symmetric log plot of the number of contributors and commits of repositories considered in this work, with those falling beneath the thresholds colored in gray. After these first filtering steps, 359 repositories remain (19% of the original total), comprising the work of around 10000 contributors.

D. Studied projects

Our data collection process yielded a total of seven projects which met our criteria. The descriptions of these projects can be found on Table I.

Taken together, we argue that our sample of scientific software projects is relevant due to their diversity: 1) they are written in up to four different programming languages (although mainly written in C++, Python, and Scala), with an average of 110.13 kLOC; 2) they span very different domains; 3) they have an average of 5.2 years of historical records; and 4) they are written by scientists that do not necessarily have a computer science or (in particular) a software engineering background. When looking at the number of stars, one might argue that our selected scientific software projects have low popularity. However, as a recent work reported, the median number of stars of R packages published on GitHub is 2 [20].

E. Establishing identities of contributors

Our step was to establish the identities of contributors. We followed the strategy used by Sheltzer and Smith [21] and first scraped data from laboratory websites and project documentation. This was followed up by searching departmental directories and performing web searches in order to disambiguate contributors where necessary.

The final step in this process is to code each individual in our dataset. This amounts to reviewing the assembled information about each individual and assigning a label to them. For the purposes of this study, we sorted subjects into the following categories: undergrad (i.e., undergraduates students), gradstudent (i.e., master's and doctoral students), postdoc (i.e., postdoctoral researchers), staff (i.e., investigators and support staff), thirdparty (i.e., external collaborators), and unknown (i.e., which the identity could not be established). In cases where we had to rely on incomplete GitHub profile data (e.g. unlisted thirdparty contributors), we attempted to extract names from handles (e.g. `johnsmith79` → `John Smith`) and cross-referenced those names with web searches for similarly named researchers in the relevant field; where we could not be reasonably convinced that the identities matched, the contributor was left as unknown. To ease understanding, we further group these contributors as juniors (i.e., gradstudent, undergrad, and postdoc), seniors (i.e., staff), and thirdparty.

F. Complementary Survey of Developers

The majority of the findings of this work come from a quantitative analysis of repositories. To triangulate our findings and better understand the perceptions of scientific software developers, we additionally performed a complementary qualitative analysis of scientific software development teams. We sought to capture, in their own words, (1) who contributes to their projects, (2) how they prepare for contributors to join or leave, and (3) what roles different people play in the development of the software.

To do this, we designed an online survey. For each participant, we presented three multiple choice questions on their background and experience in developing scientific software, followed by five open-ended questions addressing their team composition and their division of labor. Participation in the survey was voluntary and responses were anonymous. For the open-ended questions, we coded the answers and organized them into categories following the guidelines on open coding procedures [22] (cf. [23]).

To identify the target population, we reached out to development teams whose projects had been accepted to the Journal of Open Source Software (JOSS); we used the JOSS Github repository, which is used to track submissions, to collect information on points-of-contact. From the accepted submissions to JOSS, we identified 273 scientific software developers that either owned or made the majority of contributions to a GitHub project, and recruited them by email. 17 emails were not sent due to mailing errors, and 11 emails were returned due to out of office automatic replies. Over the period of two weeks, we received 72 answers (a response rate of approximately 30%).

IV. Analysis

In this section we provide answers to each research question.

RQ1: What is the tenure of different contributors by role on a project?

For **RQ1** we want to know what the expected contribution period is for different contributors based on their project role. While students may spend years with a research team and staff for decades, only a limited portion of that time will be spent on software development activities. Knowing how much labor is available to a team helps staff to understand issues related to task allocation or job rotation. To gather this information, we apply the Kaplan-Meier procedure [24] to perform a survival analysis of participants from all projects organized by role.

TABLE I: List of studied projects. Age is present in years. kLOC is calculated using the `cloc` utility, encompassing blank, comments, and code lines. PL means Programming Language.

Project/GitHub	Contributors Identified (%)	kLOC	Commits	Stars	PL	Age	Description
Chaste (Chaste/Chaste) [15]	97%	371,4k	4,7k	22	C++	8	Tissue and cell level electrophysiology, discrete tissue modeling, and soft tissue modeling
Khmer (dib-lab/khmer) [16]	90%	145,1k	6.6k	528	Python	7	Nucleotide k-mer counting, filtering, and graph traversal
PyGBe (barbagroup/pygbe) [17]	100%	12,4k	0.9k	28	Python	6	Biomolecular electrostatics and nanoparticle plasmonics
LBANN (LLNL/lbann) [18]	99%	66,8k	3,5k	40	C++	4	Artificial neural network toolkit
Hail (hail-is/hail)	98%	72,9k	3,1k	357	Scala	2	Genomic analysis
Genn (genn-team/genn) [19]	96%	37,4k	1,8k	77	C++	6	Neuron and synapse modeling
openMOC (mit-crpg/openMOC)	90%	21,6k	2,6k	50	C++	4	Nuclear reactor physics

In the medical domain, survival analysis measures the fraction of patients who remain alive for a certain amount of time after treatment. In our work, survival refers to how long it takes for a contributor to become inactive. For the purposes of this analysis, we consider a contributor inactive if they have not made a commit within the last 180 days, following the example of Lin et al. [25]; this is more strict than is done in other works (cf. [26], one commit per year counts as active). We do this because contributors may start or cease their commits in the middle of their tenure (e.g., a junior pivoting towards finishing a thesis). After tuning with different knobs, we found that six months was a reasonable limit. Moreover, recent work has also experimented with different thresholds (e.g., 30, 90, 180 days), and results suggest the same trends over the experiments [25]. We used the `gitstats` utility[3] to collect information on the length of each subject's participation in their respective projects.

The results of the analysis can be seen on Figure 3. This figure shows a series of declining horizontal steps which approaches the true survival function for that population. The x-axis represents the survival duration, while and the y-axis indicates the probability that a contributor can survive (i.e., keep actively contributing to the project). The median survival time per group is 4.06 years for staff, 1.72 years for gradstudents, 0.98 years for postdocs, 4 months for undergrads, and just a day for thirdparty contributors.

A careful, contextualized reading of this data proves informative. With regards to juniors, aside from staff, we see that gradstudents are the most likely to stay around. That being said, a typical PhD student (as almost all of the graduate students in our dataset are) takes 5 years to graduate, and this means that the median gradstudent only spends 34.52% of their tenure contributing to a project. Meanwhile, postdocs spend even less time than gradstudents as contributors, even though the typical term limit of postdocs today is also 5 years [27]; that being said, postdocs are brought on-board with prior knowledge that they can immediately apply to the software project. Finally, undergraduate students in our dataset only spend one semester out of their 4 year education participating in developing scientific software. Taken all together, the median junior in our projects spends only 24.84% of their time in their position doing software development work.

[3]https://GitHub.com/hoxu/gitstats

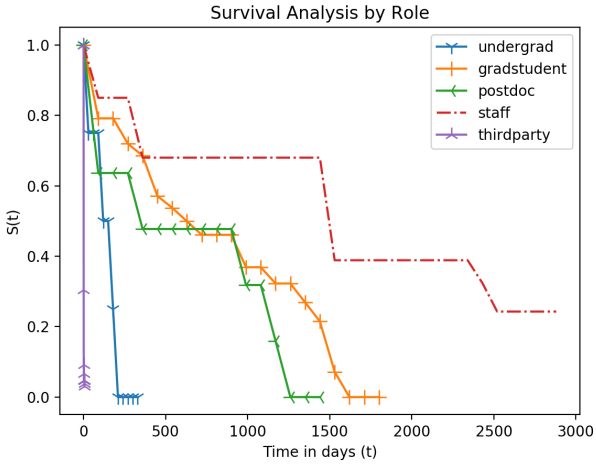

Fig. 3: A Kaplan-Meier survival plot of contributors to projects in our dataset, grouped according to role. $S(t)$ indicates the number of individuals in the population who are still actively contributing t days after starting.

Meanwhile, seniors provide the most stable presence, with a median survival time of 4 years. We note that this is affected by right-censoring because the average age of our projects is 5 years. However, our evidence suggests that senior members who do contribute code may not do so indefinitely. Once the software reaches a point of maturity, they may hand off the work to juniors. In other cases, they may leave virtually all the work to juniors. Lastly, we found that in the projects we studied, thirdparty contributors tend to remain at the periphery and do not engage with a project for any significant length of time; as we observed, thirdparty contributors stay, on average, 1 day.

RQ2: What is the breakdown of team contributors by role?

For **RQ2**, we are interested in the breakdown of contributors to each project by role. How many people contribute to projects overall? How can we characterize them? A key

distinction that we are making is that this is not the same thing as the number of contributors listed as team members on the webpage of the project or research group. For example, a graduate student may be a user of the software but not a contributor, and in the case of projects like Chaste, we can have multiple staff members responsible for design work and guidance of students but who have no commits to their name.

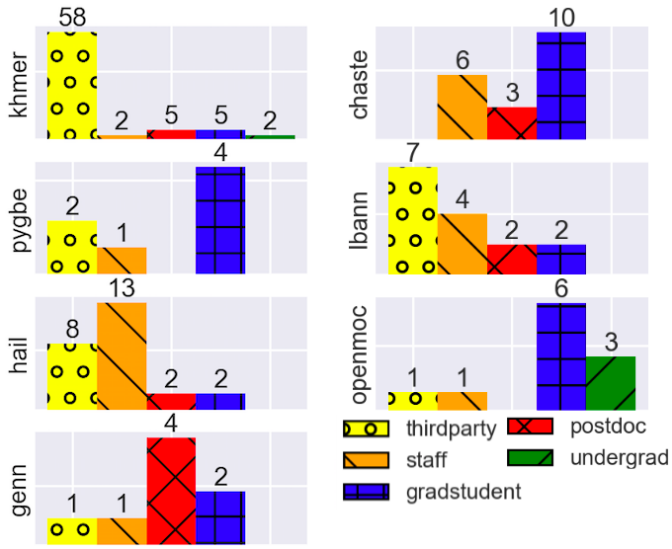

Fig. 4: For **RQ2**, for each project we present a bar chart with totals of identified contributors sorted by role.

We showcase our results on Figure 4. Taken together, JUNIOR members make up the majority of team contributors (average 69%; median 80%), with the remainder being SENIORS (average 31%; median 20%). Meanwhile, in all but one of the projects we studied, we were able to identify thirdparty contributors (average 20%; median 27%). Additionally, as a rule, both team members and thirdparty contributors that we identified have a background relevant to the domain of the project, something which we learned by analyzing biographical information used to classify contributors by role.

RQ3: How much of the development work is done by different contributors by role?

For **RQ3**, we want to characterize the amount of development work that is done by these different actors. To begin, we consider the share of commits produced by different contributors, the number and frequency of commits being well-worn metrics for engagement and investment in a project [26], [28]. Taking averages across all projects that we studied, half or more of commits are made by senior members (average 50.76%; median 63.29%). However, the majority of the other half are commits by junior contributors (average 42.9%; median 36.7%). Moreover, for 4 out of 7 projects (Khmer, Chaste, Hail, and openMOC), senior researchers are responsible for a plurality of commits (with an average commit share of 77.12%; median 73.34%); the opposite is true for Genn, LBANN, and PyGBe (average share 23.92%; median 22.58%), with LBANN having a more even split between junior and senior members. We also note that, for the projects we have studied, thirdparty contributors tend to play a very minor role in this regard; only LBANN has a notable share of commits attributable to thirdparty users (17.7% versus an average of 3.59% and median of 0.06%). Overall, in all projects but one (Hail), junior contributors produce a significant share of commits, meaning that even when seniors do most of the heavy lifting, juniors play an essential role in the realization of the project.

However, while both junior and senior alike generate significant amounts of commit activity, this is not to say that the *scope* of their activities is comparable. To better understand this, we consider interactions with and ownership of files. For the purposes of this work, we record a user as interacting with a file each time they make a commit that touches that file. By that measure, the typical junior interacts with a much smaller percentage of files compared to a senior (average 5.85% vs. 20.35%; median 0.65% vs. 11.51%). This is to say that a distinguishing characteristic of junior developers in our corpus is that they often have a narrow focus on a particular subset of a project. Meanwhile, the same is especially true for thirdparty contributors who interact with an even smaller percentage of files (average 1.64%; median 0.66%).

Likewise, we can also consider file creation. Earlier work by Poncin et al. [29] addresses file creation in their operationalization of "core" developers, as frequent creation and modification of files indicates that a user is helping to drive the vision or direction of the software. Related to this, in a recent study of large-scale open source projects, Lin et. al [25] found users who created files tended to be longer-term contributors than those who modified files. In 5 out of 7 of the projects we studied (Khmer, Chaste, Hail, openMOC, and Genn), senior team members created the majority of files (average of 69.44%); LBANN is almost evenly split by this measure, and PyGBe, as a student-driven project, has only a quarter of its files originating from senior members.

RQ4: What kinds of maintenance and evolution activities do contributors perform?

What value do different kinds of contributors add to a project? For example, once a gradstudent exits a project, in what ways did they influence the evolution of the software during their tenure? We can find some evidence for this through project pages and documentation when teams provide an itemized list of accomplishments of different contributors (as is the case for several of the projects in our study); it is typical to see juniors receiving credit for implementing novel features pertinent to their research, seniors for building out infrastructure and performing maintenance, and thirdparty contributors for providing support or helping improve the codebase. However, relatively few projects provide this kind of fine-grained information, and we would also like to be able to interrogate those claims in an empirical way.

To answer our question, we present two views of the development activities that elucidate the kinds of work that

different contributors produce. The first is an analysis of commit messages based on the approach of Hattori and Lanza [28], and the second is an analysis of file paths involved in commits based on the work of Vasilescu et al. [30]. In both cases, we are interested in categorizing commit activity according to their purpose or intent.

In the framework set out by Hattori and Lanza [28], commits are divided into four major categories of activity:

1) **Forward engineering (Fwd)**, for instance, adding new features;
2) **Reengineering (Reng)**, for instance, refactoring activities;
3) **Corrective (Corr)**, for instance, fixing bugs;
4) **Management (Mgmt)**, for instance, updating documentation.

In order to automatically classify commits into these categories, the authors compare the content of commit messages against predefined word banks for each commit type based on the earliest match found. For instance, consider the following commit message: "*This commit adds integrators supporting the combined, staggered, and pseudotransient forward sensitivity analysis methods where the sensitivity equations are solved alongside the forward equations*."[4] Unpacking the semantics of this commit message requires extensive domain knowledge and that is difficult to automate. However, the word "add" is a match in the word bank for forward engineering; it is reasonable to assume (in this case) that the commit is adding a new feature to the software.

This approach is limited in that it only considers the lexical content of messages, and it also fails to handle situations where a commit may belong to more than one category, but it remains useful as a diagnostic tool. To test the validity of this classification scheme against our corpus, we chose to manually classify a representative random sample of commits drawn from across all projects. Assuming that all projects have statistically similar commits (in the sense that the distribution of commits by type are roughly the same), a sample of 378 commits might reflect the overall population of roughly 23,000 commits with a confidence level of 95% with an interval of $\pm 5\%$. In order to arrive at the ground truth, our manual classification considers not only commit messages but also the artifacts (such as source code, documentation, and test data) that were modified and the context in which that occurred (such as preceding commits and related files).

TABLE II: Confusion matrix for validation of Hattori-Lanza [28] classification scheme. Unknown (Unk) commits are those which the algorithm failed to classify.

Automated \ Manual	Fwd	Reng	Corr	Mgmt
Fwd	53	11	1	14
Reng	3	60	0	13
Corr	1	4	60	5
Mgmt	7	14	11	16
Unk	9	36	5	55

[4]https://GitHub.com/trilinos/Trilinos/commit/e8e6d67

Table II shows the confusion matrix between the automated approach and the manual analysis. First, we note that because a commit may fail to match against the word bank, it is possible for this approach to fail to find a label. This happened for 27% of commits in our sample. We identified three causes for this: (1) manual classification relied on words that were outside of the word bank (e.g., "*vectorized* book-keeping kernels"), (2) commit messages were automatically generated and vague as to their purpose (e.g., merging an arbitrary pull request), and (3) messages could be highly ambiguous (e.g., "complete breakdown of intuition" or "it is all becoming clear"). However, we consider this to be a more gentle form of failure than attempting to shoehorn an unintelligible commit into an arbitrary category.

For commits which were classified automatically, the manual and automatic approaches agreed 69% of the time; much of the error was concentrated on management commits, only a third of which were correctly labeled. If we limit our consideration to just forward engineering, reengineering, and corrective commits, then the automatic approach agrees 89% of the time, with some minor confusion between forward and reengineering activities. As such, we limit our consideration to just those three.

The second approach we use is derived from that of Vasilescu et al. [30], which categorizes commit activity by examining the filepaths involved in changes; file extensions (e.g. `.cpp` versus `.csv`) and hints in file paths (e.g. `/src/` versus `/test/` can clue us in to the purpose of a file and, by extension, the kind of labor that an individual provides a project through their interaction with those files. The classification algorithm itself is analogous to what was previously described: filepaths are matched against a bank of regular expressions that map to different categories of files. Unlike with the Hattori-Lanza scheme, these results are much less ambiguous because it is reasonable to assume that file extensions indicate actual file types. For this work, we made several addenda to the regexes used in the original paper in order to cover additional programming languages (e.g., Pascal and Ada), data storage types commonly used in scientific computing (e.g., HDF5 and FASTA), as well as a handful of previously unaddressed build and configuration artifacts (e.g., Dockerfiles and Gradle build files). On Table III and Table IV we provide results for our two analyses as an aggregate of all contributors in the projects we studied as a way of approximating a "typical" project. The former shows what percentage of commits made by an average individual are categorized as forward engineering, reengineering, and corrective activities; the latter asks what percentage of individuals have made at least one commit that interacts with a file of a given type.

As a group, senior contributors have the highest average share of forward engineering commits (33.37%), which is to say that commits made by seniors are more likely to include novel development work, such as adding or extending software capabilities. Senior developers also play a key role in realizing the supporting infrastructure of their projects, with a majority having interactions with `build`, `devdoc`, and

TABLE III: The relative share of automatically classified commits of an average junior, senior, or thirdparty contributor. Management commits are omitted.

	Fwd Share	Reng Share	Corr Share
Juniors	26.76%	21.11%	15.93%
Seniors	33.37%	16.07%	14.91%
thirdparty	19.98%	18.89%	39.45%

`metadata` files. Likewise, a plurality of seniors interacts with `data/database` files (such as data for validation tests and parameters for research models) and `test` files.

Next, the activities of junior contributors resemble that of senior contributors in many key respects. Like seniors, juniors universally interact with `code` files, and a comparable share of their commits go towards forward engineering (26.77% for juniors vs. 33.37% for seniors), reengineering (22.11% vs. 18.90%) , and corrective (15.93% vs. 14.91%) activities. Also like seniors, the majority of juniors interact with `build`, `devdocs`, and `test` files (though in smaller measures compared to seniors), and this was true in general for all projects we studied. This reinforces our earlier observations suggesting that their work is neither subordinate nor peripheral compared to the work of seniors, but is instead vitally important to the enterprise.

Finally, we consider thirdparty contributors who, as we determined earlier, are relatively minor players who as a rule only sporadically contribute to projects. `Code` and `devdocs` aside, they scarcely interact with any kind of file. One point that stands out, however, is that these contributors are more likely to make corrective activities (with an average commit share of 35.06%). This is to say that while they make very few commits, the commits they do make are more likely to be bug fixes; that suggests that thirdparty contributors are most likely to be users of the software who have the domain knowledge and development expertise needed to correct such bugs or "scratch their own itch". In essence, thirdparty contributor behavior is similar to the kind of work produced by peripheral developers, which are typically involved in bug fixes, and they have irregular or short-term involvement in a project [31], [32].

TABLE IV: The percentage of contributors in each category who have at least one commit that interacts with a given file type. N/A indicates that no matches were found for regexes in any projects studied for a given category

	Juniors	Seniors	thirdparty
Documentation	19.1%	26.0%	0%
Images	14.5%	22.2%	2.7%
Localization	2%	0%	0%
UI	N/A	N/A	N/A
Media	27.7%	33.3%	2.7%
Code	100%	100%	70.3%
Project Metadata	36.2%	51.85%	8.1%
Configuration	34.0%	33.3%	5.4%
Build	63.8%	77.8%	29.7%
Devdocs	63.8%	88.8%	66.7%
Data/Databases	36.2%	63.0%	18.9%
Test	74.5%	85.2%	27.0%
Libraries	N/A	N/A	N/A

RQ5: How do scientific software developers perceive their own software development process?

For our final research question, we consider how scientific software developers view their own projects based on our survey data; this provides us with points of comparison with our quantitative findings. Among our survey respondents, those we identified as being most responsible for their respective projects, 36% of them were postdocs, 30% were non-academic professionals, another 30% were students (undergraduate or graduate), and 14% were professors. The majority of them (62%) work for an university or college, 11% work for the government, 8.5% work in industry, and the other 18.5% play other roles. Regarding their highest academic degree, 76.4% had already received or were working towards their doctorate degree (18% have a master degree, and only 4% have bachelor degree). They worked in a variety of fields, including computer science, fine art, chemistry, political science, urban planning, and neuroscience.

The majority of projects in our survey (61%) were developed by a team of people, though it is worth nothing that a significant number of projects were the work of individuals (39%). On average, these teams had 3.6 contributors (3rd Quartile: 4.2, max: 15); this roughly aligns with the number of active contributors in a given year in the repositories which we mined (average: 3.8, 3rd quartile: 5, max: 11).

When we asked (Q5) what and how do they contribute, we observed that 50 respondents reported software development-oriented activities such as fixing bugs, developing scripts to support research, improving documentation, and adding tests. Strongly tied with software development activities, 18 respondents reported to contribute to non-software activities, such as paper writing, grant writing, running experiments, etc. Along these lines, three respondents perceive their contributing role as "*Conducting research that feeds back into the project*". Regarding how do scientific software developers get trained to do their jobs (Q6), 70% of the respondents were self-taught, although some of them received mentorship from senior contributors (e.g., "Shadowing a more senior developer for a week or two"), while others benefited from online training programs (e.g., "The Molecular Sciences Software Institute (molssi.org) training programs."), took advantage of their own documentation (e.g., "We make sure that the docs are self-contained to ease onboarding for remote teams"), or even the pull-request process (e.g., "By first contributing some pull requests and getting code reviews"). Only four respondents were trained through their academic degree.

When considering the responsibilities they need to take to prepare for the departure of a team member (Q7), 20 respondents mentioned the importance to keep the documentation updated (e.g., "We simply try to ensure that all developments are adequately documented at the time, to help the understanding of future developers"). Interestingly, 8 respondents mentioned that this never happened, which is partially because they are working on a small or solo team (e.g., "No one has departed yet (or joined...)"). Other respondents mentioned the need to

train other, to push code, or to add tests.

Of those projects run by teams by teams of two or more people, 35 out of 41 specifically called attention to the role played by juniors in developing their software. 20 of these described them as being responsible for developing specific, non-core features of the software; projects that followed this pattern offered up explanations such as "[juniors] — by necessity – start with smaller peripheral bugs and features. Core development requires a lot of experience and knowledge." Another twelve projects, however, cast juniors as being developers of core infrastructure, typically for the reasoning that senior members "cannot afford to put much time into development".

Meanwhile, 24 of the 41 team projects emphasize the role of seniors in development. 8 of these said seniors developed the core of the software and 4 the periphery. Those that did so often emphasized the need for experience in development, insofar as "the more education a team member has (software development life-cycle, good coding practices, etc.), the better they are at seeing 'big picture' development tasks [...] these people often lead development". However, in contrast to this, ten respondents characterized seniors as being visionaries first and developers second. In this view, the role of seniors is to "coordinate activities", "drive the direction of the project", "guide the conceptual development", and to provide the "theoretical details".

Lastly, only 9 out of the 72 projects gave recognition to thirdparty contributors. Among these projects, the typical view was that while thirdparties "contribute seldomly", they were also a common sources of bug fixes, a finding echoed in our findings from **RQ4**. Likewise, these contributors were also responsible for "[submitting] small patches to make [a] tool better meet their own niche use cases".

TABLE V: A summary with descriptions of typical contributors, based on the findings in this work.

Contributor	Findings
Seniors	• Are often active on a project for four or more years. • Make the majority of contributions to a typical project (average 50%). They create the most files and touch the most code. • Are most often responsible for forward engineering activities, development of the core of the software, and infrastructure tasks. • Provide guidance and visionary leadership to junior contributors, especially when they do not have the time or resources to work on the code themselves.
Juniors	• Are active for no more than 2 years. Roughly 25%-35% of their time is spent on software development. • Make up the majority of team members. • Perform many of the same development activities as seniors, and, collectively, generate 42% of commit activity on average. • Are most likely to develop peripheral, non-core features of the software.
Third Parties	• Are active for only one day. • Have a background in the domain of the project and an interest in using the software. • Make only a handful of commits. These commits are most likely to be bug fixes.

V. Discussion

We have summarized the major findings in Table V, and now consider the potential implications of our work.

Training. As a group, juniors have long been the subject of science public policy literature. Novice researchers are "canaries in the mine" for the health of the scientific enterprise, as it is during this period that they are meant to learn the values and skills needed to participate fully as scientists [33]. However, while software development is an increasingly important skill, the amount of direct experience they acquire may be limited by competing demands in their academic careers (see **RQ1**). This brings into focus a number of different topics regarding software sustainability (e.g., the importance of good practices such as code reviews) as well as educational policies (e.g. the need for more formal software development training in STEM curricula).

Building Communities. Much has been written about the value of openness in science and the need for community support of scientific software. As we noted in section III-C, 40% of contributors stay on for only a day. Our analysis suggests that many of these may be thirdparty users who have a formal background in a domain relevant to the project; even though these users only be involved for a short time, they can still make valuable contributions such as fixing bugs. However, despite the widespread presence of short-term committers in the population and thirdparty contributors being present in all but one of the projects in our sample, only 12% of respondents in our companion survey mentioned these contributors. Based on this, we believe that better community policies could help attract these contributors, such as providing guides for new contributors and explicitly giving credit to these users.

Supporting Sustainability. Scientific software projects are known for being long-lived and under constant pressure to keep pace with scientific advances. It is common for senior project members to provide visionary leadership to guide that process, but how this translated to software construction was unclear. Our findings place seniors in a primary role as core developers who are most likely responsible for forward engineering and infrastructure tasks. Our findings suggest directions for future research, such as how research priorities generate software development tasks and when and how those tasks are delegated. A more complete understanding of this phenomenon would help software engineers develop better tools and techniques to support that effort.

VI. Threats to Validity

First, not all members of a project show up as contributors to the repository; for example, senior staff may offer guidance and support while leaving the actual implementation work to others. Second, people who stop contributing code may still be part the project. This is frequently the case for graduate student contributors, who may refocus on completing coursework or a thesis towards the end of their tenure. Third, team websites are not always up-to-date, and not all contributors are given

explicit recognition for their work; in one case, an undergraduate student was uncredited on a project site, but a subsequent web search uncovered a university press release detailing a research grant that was awarded for them to do specific work on the project in question. Finally, not all contributors are project members; as with most open-source software projects, open-source scientific software attracts third-party contributors, typically senior researchers who benefit from using the software.

Coding individuals using our approach means addressing several potential ambiguities. First, on a few occasions, a subject may have belonged to different categories at different times (e.g., a staff member starting off as a postdoc). When this occurred, we labeled them according to the role in which they made the majority of their commits. Second, for large research institutions (e.g., national labs), a software project may receive contributions from people nominally part of the same organization, but unaffiliated with the research team; we resolved this by treating those contributors as thirdparty contributors.

The commit analysis performed to answer **RQ4** was an automated process which, when applied to a large-scale number of commits, can silently yield false-positives (i.e., commits that were unable to be categorized), since commit messages might lack semantical sense [34] or are even empty [35] (i.e., zero words). To mitigate such bias, we conducted a manual analysis over a representative sample of 378 commits (confidence level of 95% with an interval of $\pm 5\%$). We observed a low number of uncategorized commits. Although uncategorized commits still exist, we believe this approach is the fairest way to categorize the commit intention because, since we are dealing with scientific software projects, the domain of our studied projects is highly specific and complex. Therefore, any other attempt to categorize commits using our own domain experience would introduce even more bias.

Lastly, one could argue that this study does not provide a novel contribution, e.g., "obviously graduate students are largely responsible for adding new features". However, such common-sense assumptions are often not backed up by empirical evidence. This paper piles such evidence and, more importantly, quantifies the phenomenon; even though some perceptions are confirmed (e.g., "gradstudents stay longer than postdocs and undergraduates"), other are uncovered (e.g., "thirdparty members survive only one day on average").

VII. Related Work

Studies on scientific software development. Turk [6] presents techniques for encouraging community engagement with scientific software in the astrophysics community, arguing that attracting thirdparty contributors requires intentional actions designed to encourage their participation. Likewise, Bangherth and Heister [13] outlines the practices of successful open-source scientific software libraries, which includes a discussion of the value proposition behind open-sourcing software primarily written by juniors. Sletholt et al. [36] performs a case study of agile development practices among scientific software projects. Howison and Hersleb [8], a qualitative study of the incentives for creating and maintaining scientific software, identifies the general breakdown of contributors to projects that they study; however, both the focus and methods used in that work differ from ours, as they are not concerned with the specifics of how different people contribute to the software development. Finally, Storer [37] provides an excellent survey of the state of software engineering practice within the scientific community. Our work shares the same motivations as others, but, to our knowledge, is the first to tackle this subject from a software repository mining perspective.

Studies on roles of contributors in open source projects. The study of core developers, i.e., developers that play an essential role in developing the system architecture and forming the general leadership structure, in open source projects is a fruitful research area [38], [39], [40]. Core developers are well-known from being active contributors. A general rule of thumb suggests that contributors with more than 80% of the overall contributions are considered core developers [31], [41]. Indeed, for some projects, this number is even higher. Recent work indicates that several well-known, active open source projects rely on 1–2 core developers to drive most of their maintenance and evolution tasks [42]. On the periphery side, research indicates that peripheral developers accounts for more than 90% of the contributors of a project [31]. Moreover, several authors have acknowledged the existence and the growth of casual contributors (i.e., developers that contribute just once) [43], [32], [44]. Here we enriched the understanding of core and peripheral developers. We also introduced the notion of third-party contributors, which share some of the behaviors commonly observed in peripheral developers (e.g., they have a short term relationship with the project, and most of their contributions are intended to fix bugs).

VIII. Conclusion

Scientific software projects are critical to the advancement of the scientific enterprise, and software engineering research can directly help those efforts through tailored tools, techniques, and practices. However, there has historically been a lack of hard data on who contributes to scientific software and how they behave. In this work, we mined logs from seven non-trivial open source scientific software projects in order to provide answers to these questions. Among our findings, we found that while senior researchers perform the lion's share of the work in many projects, junior researchers are often on the frontlines driving the software development. We also considered the habits of thirdparty contributors who, while often operating at the periphery of projects, have a valuable role to play in fixing and improving code.

For future work, we plan to conduct ethnographic studies of scientific software projects to better understand topics such as feature creation and bug fixing. We also plan to study how scientific software projects compare with conventional projects.

Acknowledgments. We thank the reviewers, the 72 respondents, and PROPESP/UFPA and CNPq (#406308/2016-0).

REFERENCES

[1] T. Hey, S. Tansley, K. M. Tolle *et al.*, *The fourth paradigm: data-intensive scientific discovery*. Microsoft research Redmond, WA, 2009, vol. 1.

[2] J. C. Carver, N. P. Chue Hong, and G. K. Thiruvathukal, *Software Engineering for Science*. CRC Press, 2016.

[3] E. S. Mesh and J. S. Hawker, "Scientific software process improvement decisions: A proposed research strategy," in *Software Engineering for Computational Science and Engineering (SE-CSE), 2013 5th International Workshop on*. IEEE, 2013, pp. 32–39.

[4] C. Letondal and U. Zdun, "Anticipating scientific software evolution as a combined technological and design approach," in *Second International Workshop on Unanticipated Software Evolution*, 2003.

[5] J. Segal, "Scientists and software engineers: A tale of two cultures," 2008.

[6] M. J. Turk, "Scaling a code in the human dimension," in *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*. ACM, 2013, p. 69.

[7] J. Howison and J. D. Herbsleb, "Incentives and integration in scientific software production," in *Proceedings of the 2013 conference on Computer supported cooperative work*. ACM, 2013, pp. 459–470.

[8] ——, "Scientific software production: incentives and collaboration," in *Proceedings of the ACM 2011 conference on Computer supported cooperative work*. ACM, 2011, pp. 513–522.

[9] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson, "How do scientists develop and use scientific software?" in *Proceedings of the 2009 ICSE workshop on Software Engineering for Computational Science and Engineering*. IEEE Computer Society, 2009, pp. 1–8.

[10] P. E. Stephan, *How economics shapes science*, 2012, vol. 1.

[11] M. Heroux, "Software engineering for computational science and engineering: What can work and what will not," Invited talk, presented at the 2017 International Workshop on Software Engineering for High Performance Computing in Computational and Data-Enabled Science and Engineering, 2017.

[12] J. Howison, E. Deelman, M. J. McLennan, R. Ferreira da Silva, and J. D. Herbsleb, "Understanding the scientific software ecosystem and its impact: Current and future measures," *Research Evaluation*, vol. 24, no. 4, pp. 454–470, 2015.

[13] W. Bangerth and T. Heister, "What makes computational open source software libraries successful?" *Computational Science & Discovery*, vol. 6, no. 1, p. 015010, 2013.

[14] A. M. Smith, K. E. Niemeyer, D. S. Katz, L. A. Barba, G. Githinji, M. Gymrek, K. D. Huff, C. R. Madan, A. C. Mayes, K. M. Moerman *et al.*, "Journal of open source software (joss): design and first-year review," *PeerJ Computer Science*, vol. 4, p. e147, 2018.

[15] G. R. Mirams, C. J. Arthurs, M. O. Bernabeu, R. Bordas, J. Cooper, A. Corrias, Y. Davit, S.-J. Dunn, A. G. Fletcher, D. G. Harvey *et al.*, "Chaste: an open source c++ library for computational physiology and biology," *PLoS computational biology*, vol. 9, no. 3, p. e1002970, 2013.

[16] M. R. Crusoe, H. F. Alameldin, S. Awad, E. Boucher, A. Caldwell, R. Cartwright, A. Charbonneau, B. Constantinides, G. Edvenson, S. Fay *et al.*, "The khmer software package: enabling efficient nucleotide sequence analysis," *F1000Research*, vol. 4, 2015.

[17] C. D. Cooper, J. P. Bardhan, and L. A. Barba, "A biomolecular electrostatics solver using python, gpus and boundary elements that can handle solvent-filled cavities and stern layers," *Computer physics communications*, vol. 185, no. 3, pp. 720–729, 2014.

[18] B. Van Essen, H. Kim, R. Pearce, K. Boakye, and B. Chen, "Lbann: Livermore big artificial neural network hpc toolkit," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*. ACM, 2015, p. 5.

[19] E. Yavuz, J. Turner, and T. Nowotny, "Genn: a code generation framework for accelerated brain simulations," *Scientific reports*, vol. 6, p. 18854, 2016.

[20] G. Pinto, I. Wiese, and L. F. Dias, "How do scientists develop scientific software? an external replication," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, 2018, pp. 582–591.

[21] J. M. Sheltzer and J. C. Smith, "Elite male faculty in the life sciences employ fewer women," *Proceedings of the National Academy of Sciences*, vol. 111, no. 28, pp. 10 107–10 112, 2014.

[22] B. L. Berg, "Methods for the social sciences," *Qualitative Research Methods for the Social Sciences. Boston: Pearson Education*, 2004.

[23] A. Wood, P. Rodeghero, A. Armaly, and C. McMillan, "Detecting speech act types in developer question/answer conversations during bug repair," in *Proc. of the 26th ACM Symposium on the Foundations of Software Engineering*, 2018.

[24] E. L. Kaplan and P. Meier, "Nonparametric estimation from incomplete observations," *Journal of the American statistical association*, vol. 53, no. 282, pp. 457–481, 1958.

[25] B. Lin, G. Robles, and A. Serebrenik, "Developer turnover in global, industrial open source projects: Insights from applying survival analysis," in *Proceedings of the 12th International Conference on Global Software Engineering*. IEEE Press, 2017, pp. 66–75.

[26] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in software engineering research," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 466–476.

[27] K. Ferguson, B. Huang, L. Beckman, and M. Sinche, "National postdoctoral association institutional policy report 2014: Supporting and developing postdoctoral scholars," *Washington, DC: National Postdoctoral Association*, 2014.

[28] L. P. Hattori and M. Lanza, "On the nature of commits," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2008, pp. III–63.

[29] W. Poncin, A. Serebrenik, and M. Van Den Brand, "Process mining software repositories," in *Software maintenance and reengineering (CSMR), 2011 15th european conference on*. IEEE, 2011, pp. 5–14.

[30] B. Vasilescu, A. Serebrenik, M. Goeminne, and T. Mens, "On the variation and specialisation of workload—a case study of the gnome ecosystem community," *Empirical Software Engineering*, vol. 19, no. 4, pp. 955–1008, 2014.

[31] K. Crowston, K. Wei, Q. Li, and J. Howison, "Core and periphery in free/libre and open source software team communications," in *39th Hawaii International International Conference on Systems Science (HICSS-39 2006), 4-7 January 2006, Kauai, HI, USA*, 2006.

[32] G. Pinto, I. Steinmacher, and M. Gerosa, "More common than you think: An in-depth study of casual contributors," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18*, 2016, pp. 112–123.

[33] K. S. Louis, J. M. Holdsworth, M. S. Anderson, and E. G. Campbell, "Becoming a scientist: The effects of work-group size and organizational climate," *The Journal of Higher Education*, vol. 78, no. 3, pp. 311–336, 2007.

[34] W. Maalej and H. Happel, "Can development work describe itself?" in *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*, 2010, pp. 191–200.

[35] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013.

[36] M. T. Sletholt, J. Hannay, D. Pfahl, H. C. Benestad, and H. P. Langtangen, "A literature review of agile practices and their effects in scientific software development," in *Proceedings of the 4th international workshop on software engineering for computational science and engineering*. ACM, 2011, pp. 1–9.

[37] T. Storer, "Bridging the chasm: A survey of software engineering practice in scientific programming," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, p. 47, 2017.

[38] R. T. Fielding, "Shared leadership in the apache project," *Commun. ACM*, vol. 42, no. 4, pp. 42–43, Apr. 1999.

[39] S. A. Licorish and S. G. MacDonell, "Understanding the attitudes, knowledge sharing behaviors and task performance of core developers: A longitudinal study," *Information & Software Technology*, vol. 56, no. 12, pp. 1578–1596, 2014.

[40] J. Coelho, M. T. Valente, L. L. Silva, and A. Hora, "Why we engage in FLOSS: Answers from core developers," in *11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, 2018, pp. 1–8.

[41] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, Jul. 2002.

[42] G. Avelino, L. Passos, A. Hora, and M. T. Valente, "A novel approach for estimating truck factors," in *24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1–10.

[43] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider, "Creating a shared understanding of testing culture on a social coding site," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13.

[44] A. Lee, J. C. Carver, and A. Bosu, "Understanding the impressions, motivations, and barriers of one time code contributors to FLOSS projects: a survey," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 187–197.

IX. Appendix

TABLE VI: Addenda to File Extensions used in Vasilescu et al.[30]

Category	Addenda
Doc	".*\.md"
Code	".*\.pas((\.swp)?)(~?)", ".*\.pxd((\.swp)?)(~?)", ".*\.ads((\.swp)?)(~?)", ".*\.adb((\.swp)?)(~?)", ".*\.bin"
Devdoc	".*\.pdf",".*citation.*", ".*license.*",".*doxyfile.*", ".*\.wiki",".*\.tex", ".*\.bib",".*\.dox",".*authors"
Db	".*\.csv",".*\.xml",".*\.fa", ".*\.xlsx",".*\.zip",".*\.h5", ".*\.bz2",".*\.tar(\.gz)?", ".*\.fq(\.gz)?",".*\.pts", ".*\.pdb",".*\.pqr", ".*\.vert",".*\.node", ".*\.edge",".*\.param(eters)?", ".*\.phi0",".*\.prototext(\.bve)?", ".*\.pkl",".*\.pbs"
Build	".*\.build",".*dockerfile", ".*\.gradle"
Config	".*\.vcxproj((\.filters)?)(~?)", ".*\.qpg",".*\.dsp",".*\.epf"
Img	".*\.graffle"

A. Questions Used in the Online Survey

1) Which of the following best describes you (select all that apply)?

- Student (e.g. undergraduate or graduate student)
- Postdoc
- Professional
- Professor

2) If you are currently employed, which of the following best describes your current employer?

- Government
- University or college
- Business or industry
- Non-profit organization
- Other (please specify)

3) Please list the highest academic degree you have received or that you are currently working toward.

- High school degree or equivalent
- Associate's Degree
- Bachelor's
- Master's
- Doctorate

4) What is the subject of this degree?

5) How many people are on your team?

6) What and how do they contribute (e.g., adding new features to the core of the project, conducting research that feeds back into the project, writing tests, fixing bugs, et cetera.)?

7) How does a new team member get trained to do their job (e.g. they are self-taught)?

8) What (if anything) do you do to prepare for the departure of a team member (e.g. we ask them to document the undocumented parts of their code)?

9) What kinds of roles and responsibilities do different people play in the development of your software? In particular, consider the following groups: junior researchers (undergraduates, graduate students, and postdocs), senior researchers (seasoned staff), and third party contributors.

10) Are there differences in what different people provide your project? If so, please explain what (e.g. juniors are responsible for small issues while seniors drive the architecture and solve the main problems)?