Modeling the Centrality of Developer Output with Software Supply Chains

Audris Mockus*
Peter C. Rigby†
audris@meta.com
pcr@meta.com
Meta Platforms, Inc.
Menlo Park, CA, USA

Rui Abreu
Parth Suresh‡
ruiabreu@meta.com
parthsuresh@meta.com
Meta Platforms, Inc.
Menlo Park, CA, USA

Yifen Chen
Nachiappan Nagappan
yifenchen@meta.com
nnachi@meta.com
Meta Platforms, Inc.
Menlo Park, CA, USA

ABSTRACT

Raw developer output, as measured by the number of changes a developer makes to the system, is simplistic and potentially misleading measure of productivity as new developers tend to work on peripheral and experienced developers on more central parts of the system. In this work, we use Software Supply Chain (SSC) networks and Katz centrality and PageRank on these networks to suggest a more nuanced measure of developer productivity. Our SSC is a network that represents the relationships between developers and artifacts that make up a system. We combine author-to-file, co-changing files, call hierarchies, and reporting structure into a single SSC and calculate the centrality of each node. The measures of centrality can be used to better understand variations in the impact of developer output at Meta. We start by partially replicating prior work and show that the raw number of developer commits plateaus over a project-specific period. However, the centrality of developer work grows for the entire period of study, but the growth slows after one year. This implies that while raw output might plateau, more experienced developers work on more central parts of the system. Finally, we investigate the incremental contribution of SSC attributes in modeling developer output. We find that local attributes such as the number of reports and the specific project do not explain much variation (R^2 = 5.8%). In contrast, adding Katz centrality or PageRank produces a model with an R^2 above 30%. SSCs and their centrality provide valuable insights into the centrality and importance of a developer's work.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**; **Collaboration in software development**.

KEYWORDS

Software supply chains, Developer productivity

*Mockus is also a professor at the University of Tennessee, Knoxville, USA.
†Rigby is also a professor at Concordia University in Montreal, QC, Canada.
‡This work was done while Suresh was at Meta

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3613873

ACM Reference Format:
Audris Mockus, Peter C. Rigby, Rui Abreu, Parth Suresh, Yifen Chen, and Nachiappan Nagappan. 2023. Modeling the Centrality of Developer Output with Software Supply Chains. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3611643.3613873

1 INTRODUCTION

Improving and measuring software productivity is difficult and many researchers and practitioners have simply measured output as the number of pull requests, modification requests, or commits a developer has produced. In this work, we hypothesize that the work context can be partly characterized via structural properties of a network representing explicit and implicit relationships among software artifacts and people. Software supply chains (SSCs) represent the relationships between developers and artifacts in a software project. For example, one common SSC is the network of files that change together in a commit, with a node being a file and edges between files in the same commit. Investigating how the structural properties of SSCs explain the variations in developer output has significant scientific and practical value. In this work, we aim to a) construct software supply chains within a large and diverse (in terms of programming languages, project size, and application types) industry code base observed over a period of 10+ years; b) create and fit a set of models for developer output starting with the attempts to replicate historical results and then by enhancing the model with factors derived from several kinds of software supply chains; c) factor the Katz centrality and PageRank of nodes in the SSC into the models to understand how the centrality of a developer's work impacts output.

Methodologically, we rely on version control, code review, and human resources systems to reconstruct the complete history of software supply chains for the entire code base within the company. We then model how the SSC properties affect the output and proceed to operationalize and calculate developer output and the SSC network-derived measures. These SSC network measures are calculated for the trailing 12 months and are then used to predict developer output for the subsequent month using Generalized Additive Models (an extension of multiple regression that accommodates modeling nonlinear relationships). The model, in addition to providing the output growth curve, allows the quantification of the relationships between these factors and developer output.

We provide evidence for the following research questions related to the tenure of developers and the raw output, the centrality of

the output, and a combined model that considers organizational and reporting structure.

RQ1. Output vs. Tenure: How quickly do new developers become productive in terms of number of commits?

Onboarding onto a new team requires substantial time and effort [20, 37]. Zhou and Mockus [45] modeled how long it took developers to become productive in terms of the number of commits. They found that the number of commits a developer writes plateaus at one year. We replicate this work by modeling how the commit count, CC, per developer increases with tenure.

Result: At Meta, we find that the developers' rate of output either plateaus or continues to grow at a slower pace depending on the project.

RQ2. Output Centrality: Does the centrality of a developer's work plateau over time?

Prior work used a simple local measure of degree centrality: the number of developers who touched a file, with more developers indicating a more central file [45]. We, instead, use the entire SSC and global measures of centrality (Katz centrality and PageRank) that take into account not just the neighboring nodes but their importance as well.

Result: Prior work concluded that the centrality of the work continues to grow at a linear rate for the entire study period of three years [45]. Intuitively, it is unreasonable for work to grow linearly forever. We find that the SSC-derived centrality measure of the developer's output continues to grow for at least three years, but the growth starts decelerating after approximately one year.

RQ3. Full Model: What is the relative contribution of the SCC measures in explaining the number of commits?

We want to understand which aspects of the SCC help explain the variations in individual developer output. In this research question, we progressively add SSC measures to model CC.

Result: The basic measures, including tenure, project, and number of reports, have low predictive power of the developer's output, $R^2 = 5.8\%$. In contrast, factoring either measure of centrality into the model produces a R^2 above 30%.

The remainder of this paper starts with related work in Section 2 and the development process at Meta in Section 3. In our methodology and background, Section 4, we describe key network concepts such as communities and network centrality and how they may be used to differentiate parts of the system. We also discuss output measures, measure operationalization, and statistical modelling methodology. We answer our research question in Section 5. Our findings are discussed in Section 6, limitations in Section 7, and conclusions in Section 8.

2 RELATED WORK

The problem of measuring developer productivity is different from better studied problems of software cost [8] and effort estimation [30]. In the former case, models are used to associate properties of the product with the cost (or effort needed) to build it. For example, perhaps the best known COCOMO [8, 9] model relates the logarithm of cost to the size of the software and other factors. The productivity in software projects is distinct from the performance on programming assignments as the developers are not faced with clearly defined programming tasks but need to perform maintenance and enhancement activities including learning about the system and coordinating their work with others' [11].

Developer productivity, on the other hand, concerns the value of developer output relative to effort needed to produce that output. In software industries where developers work full time, many studies assume that overall developer effort is roughly proportional to calendar time (with exceptions for vacations or training) [4, 5, 15] and suggest models that, based on different properties of the tasks developers concurrently work on, tease out the relative contribution of these properties on the effort needed to accomplish the task. In this paper, we make a similar assumption that the developers spend similar amount of effort per sufficiently large unit of time.

While there is no single measure of productivity, prior works have seen that developers become more proficient and start engaging with more complex or difficult tasks with practice and that this increase should be most clearly seen in newcomers of large and complex software projects. Studies of learning in software development show that the learning trajectories over time vary among different types of tasks [6], most tasks take less time to accomplish with practice [36], and practice makes the difference between traditional mastery (accuracy) and true mastery [7] (i.e. fluency, accuracy + speed). The Legitimate Peripheral Participation (LPP) approach argues that the learners' participation of practice is at first legitimately peripheral but increases gradually in engagement and complexity [23] or, in other words, people engage in different (often more complex) tasks as they learn. LPP was applied by [44] to explain the change of roles in Open Source Software (OSS) development. Von Krogh *et al.* [42] looked at the strategies and processes by which newcomers join the existing OSS community, and how they progressed to the stage of contributing code. Furthermore, Mockus [27] found that a highly experienced developer may need up to six new replacement developers in large software projects. One goal of this work is to use Meta's data to understand the impact of centrality on the output of developers.

We go beyond the centrality of the nodes and also obtain the SSC structure by leveraging community detection [18, 19]. Prior works have used the changes made to the same function block and committer-author relationship to link authors. The principal difference with our approach is that we are not just constructing the author network but the entire SSC that also includes files, utilizes call graph, include graph, and co-changes (see Section 4.1).

We frame our study partly as a replication study on the multi-dimensional framework of how developers acquire expertise [45]. That work separated multiple dimensions of developer expertise, proposed ways to measure them, and quantified the learning curves that describe how each type of expertise is acquired. Specifically, they observed that the nature and complexity of development tasks differ greatly between novices and experts, that these differences involve the difficulty and importance of the tasks. The replicated work uses the term *centrality* not in the network sense, but as an indication of the impact of the task. More *central* developers tend to engage in more impactful tasks that tend to modify more critical areas of the code. Such *task centrality* thus can be transferred to developers and code. In [45], they defined four dimensions: the importance of a task based on its long-term impact (e.g., adopting a new framework), its potential to directly affect a customer,

a product, or the development team. Since we extensively use the word centrality as a precisely defined network property, we try to replace the term "task centrality" with task "impact." We then argue that network centrality appears to approximate at least some aspects of the less formally defined task impact. They measured the number of modification requests (MRs) completed each month and found that it plateaued after 6 to 12 depending on the project size [45]. Furthermore, they measured developer centrality growth over developer tenure and found it to linearly increase over the observed three-year interval. In our work, we first replicate these core quantitative findings and then argue that certain network measures of SSCs appear to be better suited to measure "task impact" or centrality than previously proposed measures.

3 SOFTWARE DEVELOPMENT PROCESS AT META

Meta like other companies runs software on their own servers and does not install it at customer locations. This enables rapid updates to the software and allows fine-grained control over versions and configurations. At Meta, this deployment has led to a practice of daily and weekly "push" of new code to production. Before being pushed, code is subject to peer review, internal use, and extensive automated testing. After the code push, engineers carefully monitor the apps' behavior to identify any signs of trouble.

Similar to Open Source Software (OSS), at Meta, the developers are also users, so they have first-hand knowledge of what the system does and what services it provides. Engineers continuously develop new features and make them available to users because of the need to constantly evolve to satisfy not only changing user needs but also competition from other companies. As in many other companies, Meta's new code is deployed as a series of small changes as soon as they are ready. Since most of the functionality is on the server side, deploying new software to the servers immediately makes it available to all users, without any need for downloads and local installation. The ability to deploy code quickly in small increments behind guards and feature toggles enables rapid innovation [34].

At Meta, every line of code that is written is reviewed by another engineer. This serves multiple purposes: the original engineer is motivated to ensure that the code is of high quality, the reviewer comes with a fresh mind and might find defects or suggest alternatives, and, in general, knowledge about coding practices and the code itself spreads throughout the company. Phabricator [1] is the backbone of Meta 's Continuous Integration system and is used for modern code review, through which developers submit changes (commits) and comment on each others' commits, before they ultimately become accepted into the code base or are discarded.

In addition to developer and release engineer code and regression tests, Meta employees effectively test the latest code while using it internally. This exercises the code under realistic conditions, and employees can report any defects they encounter. A helpful property of having employees double as testers is that as the number of code changes grows with the company, the number of testers follows suit automatically.

Phabricator and the version control systems are also used to measure the development process where both events associated with the code change and developer and reviewer actions and the current state of all commits are recorded. When developers submit their code for review, they make a commit (create a version of the code) representing the initial version of the commit. If reviewers notice any issues or suggest improvements, they may make additional revisions until the commit is either approved and is incorporated into the code base "is landed," or it may be abandoned. We do not consider abandoned commits in our analysis. Each commit has an author create date, a list of files modified, as well as some statistics: non-empty lines changed.

Static analysis tools are widely deployed at Meta; we use Glean[2] databases for C/C++ code and for Hack[3] to measure dependencies between files. We combine these datasources to understand centrality and developer output.

4 METHODOLOGY AND BACKGROUND

In this section, we describe key concepts and methods used, starting from the definition of SSCs, basic graph measures, measures of developer output, and operationalizations we used in the study.

4.1 Background on Software supply chains

Contemporary software development rarely creates all the functionality from scratch as was the case half a century ago. First, a massive number of platforms, development tools, programming languages, frameworks, and packages have been created and are ready to be reused via package managers. We refer to such reuse as the technical dependency software supply chain. An example is "file A uses a function implemented in file B" or "file A imports package B." In some scenarios, instead of introducing a dependency, developers choose to *copy* the code. We refer to copy-based reuse as the second type of software supply chain. The third type of software supply chain is the exchange of information when developers create or modify the source code. We refer to it as knowledge-based SSC, and it is the most fundamental type for our study of developer output.

All three types of SSCs can be detected empirically using version control systems, code review support systems, and source code analysis tools. The interconnectedness of software and software development can be revealed by reconstructing and measuring these supply chains. As described in detail below, suitable types of network analysis reveal communities of developers, modules of source code, parts of the code forming infrastructure, key developers responsible for various areas of the code, knowledge transfer or loss, and other aspects of the software supply network. Project and reporting structure also provides additional information about the structure of the projects and products in the organizations such as Meta, where the codebase for many products is colocated in the same repository.

4.2 Basic network measures of SSCs

Since SSCs are networks, we use techniques borrowed from network analysis to characterize the impact of the nodes (developers

[1] http://phabricator.org

[2] Glean is an open-source system for working with facts about source code. Available at https://glean.software/

[3] Hack is a programming language for the HipHop Virtual Machine (HHVM) and is a dialect of PHP.

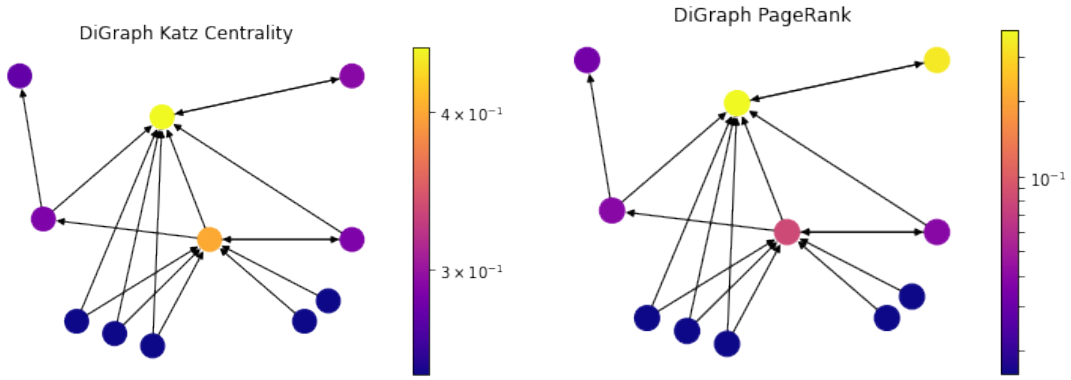

Figure 1: Example of Katz Centrality and PageRank [2]. The lighter the color the more central and important the node is. PageRank down-weights nodes with many out edges. Katz centrality does not require directions on edges which is an advantage for SSC because the direction is not always obvious, *e.g.*, when a developer changes a file which direction should the edge go?

and files) and detect communities in the megarepository housing all Meta 's code. From this repository we selected data related to Projects A and B that represent social networking software and for Project C the provides software for infrastructure used at multiple projects. The primary way to discriminate among nodes in a network is by measuring its centrality. The simplest network centrality measure is degree centrality. It counts the number of edges that start or terminate in a node. The main issue with the measure is that it does not take into account the importance of the nodes these edges lead to. Katz centrality [22] introduces two positive constants α and β to take into account not just the number of other nodes (degree centrality), but also the importance of these nodes, while addressing the problem of eigenvector centrality (where the node centrality is simply the corresponding component of the eigenvector computed for the largest eigenvalue of the adjacency matrix) with zero in-degree nodes: $x_i = \alpha \sum_j A_{ij} x_j + \beta$, where A_{ij} is an element of the adjacency matrix. β gives a free centrality contribution for all nodes, even though they do not receive any contribution from other nodes. α determines the balances between the contribution from other nodes and the free constant.

PageRank [31] tries to down-weight nodes with many out-links (e.g., a spam page that has millions of links). Some files may be changed with a lot of other files, such as release notes, other changes might be bot-generated and so forth. PageRank reduces the impact of such outliers on the calculation of centrality and is particularly suited to detect infrastructural parts of the system, i.e. parts on which the remainder of the system relies on. In PageRank, the contribution value of a node is divided by out-degree of the node: $x_i = \alpha \sum_j A_{ij} x_j k_j^o + \beta$, where $k_j^o = 1$ for zero out-degree nodes to avoid division by zero.

In Figure 1 the node at the top right only references a very important node, and it becomes much more important compared to the Katz centrality (left graph); on the other hand, the node in the center, which gets contributions from high out-degree nodes, loses its importance. One drawback of PageRank is that it needs a directed graph, and results vary if the direction is changed. Since SSCs are bi-graphs and co-changes (see below) are multi-graphs, it is not always obvious how to pick a direction for a link. Katz centrality helps in such cases, as it works for undirected graphs.

In addition to the characterization of each node, we expect to find a structure in the SSCs. Specifically, we are looking for a community structure that connects developers and files for the project, product, or a module they happen to work on. Since developers may move among projects and the same code may be used by many projects or products, one needs to detect communities empirically when projects are not clearly delineated.

A commonly used approach that maximizes modularity is the Louvain algorithm [1], which iteratively optimizes local communities until global modularity can no longer be improved given perturbations to the current community state. Modularity is measured as the difference between the actual number of edges in a community and the expected number of edges in the community. Louvain method has extremely fast and highly scalable implementations that work well for the networks of hundreds of millions of nodes [32]. Louvain method has a γ parameter with $\gamma = 0$ always leading to a single community, and $\gamma = 2$ leading to all nodes being singleton communities). We used the default value of $\gamma = 1$ in the networkit $C++$ function PLM [40, 41].

From our perspective determining the "correct" number of communities is not important: we simply need to subdivide the massive network of files and developers into distinct communities, so there was no need or criteria by which to choose some other value of γ. We calculate centrality (and communities) using the complete network of developers and files in order to ensure the technical structure transfers to and is affected by the social structure. For example, many files depend on a specific file F. Any developer who modifies F inherits some of that centrality by being linked to the central node.

4.3 Constructing SSCs at Meta

The raw data for constructing SSCs can be obtained from any version control system by collecting information about code changes. Specifically, each transaction or commit, modifies a set of files, specific lines, and is authored by a developer. It typically has one or

more dates associated with it, as well as a pointer to the parent transaction(s)/commit(s). Each commit thus represents a link between the author and the files they have created or modified. According to our definition, these links represent edges in the *knowledge-based software supply chain*. Since there are often multiple files in a single commit, there are relationships among files, and we refer to these as co-changed file-to-file relationships.

There are other file-to-file relationships; specifically, we construct a file include network (for $C/C++$) obtained from Glean where the nodes are files and edges indicate that a file includes another file. We also obtain file of function call to file of function definition for the Hack programming language to operationalize technical dependency SSCs.

In summary, for any interval of time and for each developer, we can measure their output via the following three quantities: commits and lines authored by a developer. We also have time-stamped links among the source code files based on co-change and from files to authors. For the technical dependency SSCs, however, we use the latest version of the codebase as computing call hierarchies for the entire code base for each month would be too computationally expensive.

Similar measurements could be done in most software development organizations hence the approach we propose is not unique to Meta. What is perhaps not as common is the fact that in most organizations such data would be kept in project-specific (and often numerous) version control and code review systems, while at Meta most projects use the same system (Phabricator) and a single Mercurial megarepository.

In addition to version control data, we also collect team membership data as it is likely to have an effect on developer productivity. This data resides in separate human resource databases and needs to be extracted, processed, and linked with the software development data described above. Typically, such systems record the latest state of things, such as the current management hierarchy and team membership. Fortunately, HR information at Meta maintains prior states and we can determine transitions between teams and changes of managers enabling the reconstruction of the management and project hierarchy at any time in the past. We use it to reconstruct these hierarchies at the time of each commit.

4.4 Learning Curve and Developer Output

While it has been known since the early days of software engineering that developer output varies by more than one order of magnitude [12, 13], it was not clear what part could be attributed to experience and what part was personal. Here we focus on context factors, such as developer's past experience at Meta and on the properties of the SSCs to model developer's output. As in other organizations, developers' experience should play a role, hence we suspect that developers' output increases with the length of tenure [45]. SSC network structure and their position within the network may also matter. Developers' membership in different projects and reporting to different managers likely affect output since each project may have its own type of product, application, or other specialization that makes developers' output more uniform within that community than across communities. We first discuss the learning curve and then consider aspects related to SSCs.

A learning curve has been investigated extensively in the past for various kinds of tasks and, most recently, to model developer output as it grows with tenure [45]. A specific parametric form of the learning curve was proposed by Ritter and Schooler[35]: $T = C \cdot \text{Trials}^{-C_{\text{task}}}$, where T is the time it takes to perform a task, Trials is the number of times a person has performed that task, C is a constant, and C_{task} is a task-specific constant. The shape of the curve shows how performing more trials through practice leads to reduced performance time. In our study, we borrow this basic idea of learners getting faster through practice. Following [45], however, we use the number of tasks (represented by the number of commits in this study) performed per unit of time as the measure of developer performance, i.e., output per unit of time.

The learning curve tends not to be linear as exemplified by the equation above. To estimate the developer learning curve, we fit a generalized additive model (GAM) [16] implemented by Wood [43] in R [33]. GAM is a variation of the linear regression: $y = C + f_1(x_1) + \ldots + f_m(x_m) + error$, where $f_i, i = 1, \ldots, m$ are typically smoothing functions such as splines.

4.4.1 Commit per Developer Month. Our primary focus is to explain the variation in the output produced by a developer each month. While the developer output could be measured variously (e.g., lines of code, commits, and files modified), we use commits per month. In short, these are development tasks created by actual developers (not automation) that were reviewed and merged into the project. This definition excludes, for example, experimentation or other modifications that do not propagate to the project codebase. We refer to the output measures as CC (The Commit Count per Developer Month) and denote it as $O(d, m)$ where d denotes the developer and m the month. In other literature, commits may be referred to as Modification Requests (MRs) and in Open Source Software a merged Pull Request (PR) represents a concept similar to commit. Notably, CC is highly correlated to other output measures such as the number of lines changed and files touched.

Our key goal is to explain CC by modeling it using predictors quantifying developer properties such as tenure, their position within the organization, and the code they work on. To model CC, we first log-transform it for two primary reasons. First, we expect the predictors of CC to contribute in a multiplicative fashion as has been the convention of most effort models, e.g., COCOMO [8]. In simple terms it means that a developer with a 10% higher skill will produce 10% more commits as opposed to a fixed number of additional commits. The second reason is that the distribution of commits is highly skewed (the mean is much higher than the median) and not suitable for common statistical models that typically assume a more symmetric distribution; see, e.g., [25, 26, 28, 29].

4.5 Operationalizing SSCs and Measures

We take a holistic approach to include networks from developers, source code files, and reporting hierarchy to operationalize our software supply chain (SSC). Once we construct SSCs we then calculate the predictors of the CC based on the entire SSC. This approach is necessary because the local properties of the nodes may not be sufficient to fully characterize their impact on CC. For example, the importance of some files may be quantified only by

detecting that such files are modified only by the most experienced developers.

Our SSC consists of nodes and edges. The nodes are individuals (developers and managers), teams (groups of developers), commits (files modified in a single transaction as recorded by the version control system), and files. At any specific time, developers and managers are connected in a reporting hierarchy, while each developer belongs to one team. A commit is authored by a developer and modifies a set of files, providing an edge between a developer and the files they modify. A file may depend on another file (e.g., via C-language #include statement). Finally, a commit has edges to each commit used to implement it. We also use community detection algorithms to assign each node of the SSC to a community based on the edges listed above. These calculated community nodes have edges to each member of the community. In total, we have six types of nodes (developers, managers, teams, commits,files, and communities) and eight types of edges (reporting, authoring, part-of-team, part-of-commit, file-file, part-of-community) in our SSC. We denote edges via letter E, so that, for example, an edge $E_{reports}(a, b, m)$ indicates that the person a reports to person b during the month m. It is important to note that most edges have a timestamp associated with them. To avoid data leakage, we construct SSC only on the data predating the month m for which we calculate developer output $O(d, m)$

Below we present the calculation and the theoretical justification for each explanatory variable. The precise operationalizations are also given in Table 1.

Has reports: $R(d, m)$. As some of the developers start with management duties, their coding output should decrease as they become more involved in management and mentoring activities. We use an indicator variable which is 1 if the developer has anyone reporting to them. Supervising and mentoring should take some of the time that could have been devoted to coding, thus reducing their coding output. We calculate this by observing if the developer d had any outgoing reporting edges prior to month m. In short, $R(d, m) = |\{d_1 : \exists M < m : E_{reports}(d, d_1, M)\}| > 0$.

Prior Teams $|Tm|$ *and Prior Managers* $|Mg|$. We calculate the number of prior teams and managers a developer has had. This roughly approximates how often the reporting structure for a developer has changed. This also approximates the breadth of experience a developer has had.

Community size. $|C(d)|$.At Meta developers can freely access any part of the codebase. The codebase contains code for a diverse set of products and projects. Each one may use distinct practices, tools, and programming languages, resulting in potentially different counts for the same value of the output. We use community detection in the constructed SSC to empirically identify strongly connected groups of nodes using the Louvain algorithm [32]. The size of each community may serve as a numerical predictor of coordination needs. We expect that larger communities will lead to lower productivity as the coordination needs for the corresponding projects are likely to be higher.

Task importance. Katz centrality (K), PageRank (P). In most software projects, newcomers are not assigned the most important tasks, such as modifying parts of the system that are very complex, may affect many other parts of the system, may cause downtime, or may require major long-term changes in the ways the system is maintained. Prior work found that the importance of tasks showed continued growth throughtout the considered period of more than three years [45]. As discussed in the section describing graph properties, PageRank is optimal for pointing out the infrastructural parts of the system that are not frequently changed or not changed by many developers. We suspect that these parts are more complex and probably riskier to change. Katz centrality would be low for such inactive "roots," even if everything else depends on them. Complex parts of the code frequently modified by many developers, co-changing with many other files, and depending on or using many other components would have high Katz centrality.

Table 1 lists the exact operationalizations of response and explanatory variables. $g(range)$ stands for the graph constructed during the period $range$ (or the entire history if not specified), m represents the month for which the variables are calculated and d indicates the developer. We note that the centrality is calculated over the trailing 12 months and does not include data for the current month. Community is calculated over the entire date range to make sure it does not vary month-to-month. $Louvain(g)(d)$ indicates the community obtained from graph g for developer d while the project is determined by the largest number of commits a developer did during the month.

5 RESULTS

In the following section we discuss our findings, as we address three fundamental research questions through their dedicated subsections.

5.1 RQ1. Output vs Tenure: How quickly do new developers become productive in terms of number of commits?

A major conclusion from [45] is that "Developers' productivity plateaus within 6-7 months in small and medium projects and it takes up to 12 months in large projects." As discussed earlier, they measured developer output in the number of commits or MRs a developer produced. To answer this question in the context of Meta, we create a simple model where the response variable is developer output and the only explanatory variable is the developer's tenure. To represent our models, we use the R language notation and use a GAM model as discussed in Section 4.4. For example, the formula $y \sim a + b * c$ means that "the response y is modeled by explanatory variables a, b, c and the interaction between b and c." Our model of output and tenure in R notation is: $Od(d, m) \sim T(d, m)$

At Meta we have similar findings to [45] shown in Figure 2. The simplest model relating tenure to the volume of changes appears to show a rapid increase in the rate followed by a plateau for Project A. Project B and the combination of smaller projects, Project C, show a gradual but slowing growth for up to two years. The results suggest that the findings in [45] could be replicated with respect to the rate of developer output growth as developers become familiar with the codebase over time. We also see a variation among projects where Project A shows a plateau that appears to start sooner, while other projects still exhibit some growth at the end of the observed period. The apparently pulsating growth rate in the category "other

Table 1: Descriptive statistics for the explanatory variables.

Name	Measure	Operationalization
Tenure in years	$T(d,m)$	$m - join(d)$
Has Reports	$R(d,m)$	$\|d_1 : \exists M < m : E(d,d_1,M)\| > 0$
Prior Teams	$\|Tm\|$	$Tm = \|team : M < m, Eteam(d,team,M)\|$
Prior Managers	$\|Mg\|$	$\|manager : M < m, Ereports(manager,d,M)\|$
Community Size	$\|C(d,m)\|$	$\|Louvain(g)(d,m)\|$
Katz centrality	$K(d,m)$	$K(g(M < m \&\& M \geq m - 13))$
PageRank	$P(d,m)$	$P(g(M < m \&\& M \geq m - 13))$

projects" is likely caused by the superposition of several different growth curves (corresponding to each project) that plateau at different times but are overlayed on top of each other. The successful replication to supports the applicability of the key notion of the learning theory in the context of software development tasks.

Developers' rate of output either plateaus or continues to grow at a slower pace depending on the project.

5.2 RQ2. Centrality: Does the centrality of a developer's work plateau over time?

The raw number of commits that a developer produces cannot continue to grow forever. Indeed, according to LPP theory and extensive empirical evidence discussed in Section 2, as developers become more experienced, they tend to take on more important (central) or difficult work. Prior work has found that developers' centrality continued to increase linearly across the entire tenure range of three years and the qualitative results suggested that for the largest projects, the tasks continue to grow in importance even after ten years [45]. As described in Section 4.5, we calculate PageRank $P(d,m)$ and Katz centrality $K(d,m)$ for each developer and each month based on the SSC constructed over the preceding twelve months. Instead of modeling raw CC, we now want to understand how the developers' centrality grows with their tenure. We have two models in R notation: $K(d,m) \sim T(d,m)$ and $P(d,m) \sim T(d,m)$. Since curves are similar for both models we only show $K(d,m) \sim T(d,m)$ in Figure 3. The figure shows continued growth over a much longer period which contrasts with the plateau in the raw output model in Figure 2. In particular, we do not observe linear growth in centrality as was reported in [45]. This may be attributed to the differences between the organizations studied, but it may also be a result of the differences in the modeled quantity. The replicated work modeled a very simple local measure of degree centrality calculated on a partial (containing only authoring edges) SCC. On the other hand, we model using a more complete SSC and a centrality measure that takes the entire network into account. Furthermore, the linear growth does not appear to be compatible with the key tenet of the learning theory: a decrease in the growth rate should be manifest.

Table 2: Relative contribution of SSC measures

Model	Explanatory Variables	R^2
1.	$T + R + Proj + \|Tm\| + \|Mg\|$	5.3%
2.	$T + R + Proj + \|Tm\| + \|Mg\| + \|C\|$	5.75%
3.	$T + R + Proj + \|Tm\| + \|Mg\| + \|C\| + P$	30.1%
4.	$T + R + Proj + \|Tm\| + \|Mg\| + \|C\| + K + P$	31.9%

The centrality of the developer's output continues to grow for at least three years, but the growth decelerates after one year.

5.3 RQ 3. Full Model: What is the relative contribution of the SCC measures in explaining the number of commits?

The different attributes of the SSC might have a different impact on developer output. We would like to determine attributes that are the most influential on (or, in statistical language, explain the most variation of) the CC.

Table 3: Sign of the Estimated coefficients for the full model (Model #5)

	Sign	t value	Pr(>|t|)
(Intercept)	+	28.00	0.00
$\|C\|$	+	12.00	0.00
R	-	-43.00	0.00
K	+	130.00	0.00
P	+	140.00	0.00
$O(Prod_B)$	-	-7.80	0.00
$O(Prod_A)$	+	3.30	0.00
$O(Prod_{other})$	-	-12.00	0.00
$\|Mg\|$	+	37.00	0.00
$\|Tm\|$	-	-30.00	0.00

To discover that, we start from simple models and measures and report a fraction of the variance explained as we add more and more sophisticated explanatory variables to the model. Table 2 shows how we progressively add explanatory variables to the model and compare the resulting model fit (R^2). The response variable is always developer commits per month, $O(d,m)$. The first model contains only the developer tenure, a categorical variable representing the project name, whether a developer has reports, and the number

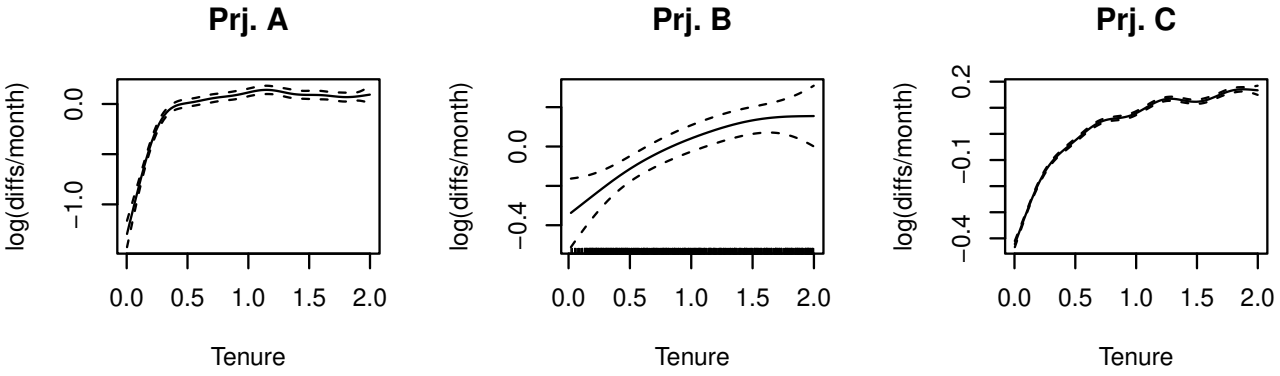

Figure 2: Developer output and Tenure. We see an increase followed by a plateau. Dashed lines show confidence intervals for the model.

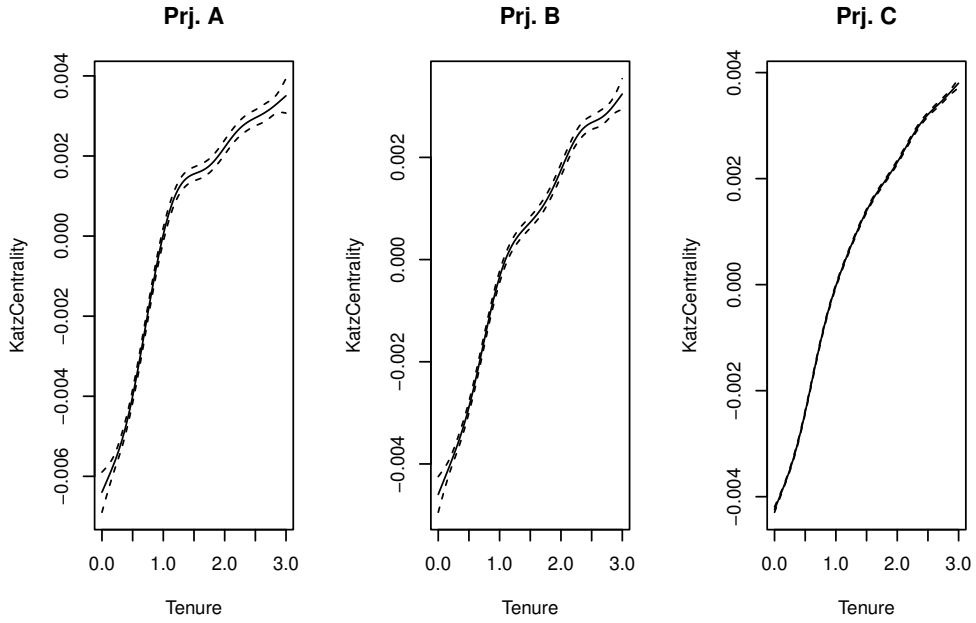

Figure 3: Developers' Katz centrality versus tenure. The centrality of developer work continues to grow, but at a sublinear rate.

of managers and projects. Despite this long list of explanatory variables, the model only explains the output $R^2 = 5.3\%$ of the observed variation in the output. Adding the size of the empirically-derived community has a negligible impact, but adding either of PageRank or Katz centrality results in a much better model fit with an R^2 above 30%.

To further understand the impact of each explanatory variable, Table 3 shows the direction of the predictor effects on the rate of output. Specifically, the size of the community $|C|$, Katz centrality K, PageRank P, and having more managers all increase output, while managing others R and hopping over many teams T decrease it. Significant differences between products exist, as was hypothesized. The only relationship that is opposite to our intuition is that the increase in the number of teams in which a developer has participated appears to decrease the rate of output. This may require further study to better understand the reasons for such decrease, but it may reflect the fact that repeated re-orgs and project changes may impact CC. Below we follow with a general discussion of the findings.

The basic measures including tenure, project, and number of reports have low predictive power of developer output, R^2 = 5.8%. In contrast, factoring either measure of centrality into the model produces a R^2 above 30%.

6 DISCUSSION

We have partially replicated the quantitative part of a prior study investigating the growth of developer output and fluency in a dissimilar software organization [45]. Our primary aim was not only to establish the accuracy and generalizability of the scientific claims, but, more importantly, to find models that have interpretable attributes explaining developer output and to make progress on identifying potential candidates that provide more nuanced measures of output and productivity.

While we were able to replicate some of the prior findings, we also observed some empirical and logical discrepancies that strongly suggest a revision to the hypotheses posed previously. The growth rate in the centrality of output in Figure 3 appears to have a slowing growth rate unlike the linear growth observed in [45]. The simplistic measure of degree of centrality (how many developers modified a file) used in prior work may not accurately gauge the file importance unlike Katz centrality or PageRank (measures that also take into account the centrality of other developers). This slight but important difference in growth of output centrality from the inclusion of dependencies among code and graph-theoretic considerations of Katz vs. degree centrality makes more sense because it is unreasonable to assume a linear growth continuing over a very long period of time. Replication, whether successful or not, is an important way to validate scientific [24] and replications of industry studies in software engineering are exceedingly rare [14]. In fact, in many fields even for exact replication studies most research results can not be replicated [3, 10, 17]. We, therefore, conclude that our finding, albeit for a single study, suggest that a successful replication of industry study may be possible if the original method is described in sufficient detail.

In addition to replication, we have focused on representing software development as a network involving multiple types of nodes and edges and, more importantly, theorizing how the properties of such network might enable us to quantify key aspects of Legitimate Peripheral Participation (LPP) theory: peripheral and central developers, code, or tasks. In fact, it appears that the proposed SSCs might represent not just great predictors of developer output, but also serve as proxies of difficulty or impact. Since such measures can be calculated in most software projects, they may lead to software tools that exploit these differences in importance to prioritize software development or quality assurance work.

From the practical perspective of outlining directions towards better proxies of developer productivity than the simple raw output, we introduced a number of robust measures that are not incompatible with the conjecture that they capture some aspects of value and centrality that go beyond developer output. Specifically, these measures of centrality are calculated from the entirety of the software supply chain including technical dependencies among source code files, implicit dependencies among files changed in a single commit, and authorship connecting source code and developers who modify it. Once developers start working in a project, they tend to start from simpler, less important tasks and, with more experience, move towards more complicated and more important tasks. By constructing the SSC for every month, we can capture the dynamic nature of developers' code and use it to explain a large share of the variation of developer output. In particular, the two measures of developer's centrality reflect global properties of the entire SSC and thus potentially address the concerns that developer's output is an individual measure that may or may not help the productivity of the entire the team.

Future work could investigate other centrality measures beyond PageRank and Katz centralities and whether other types of network characteristics may be suitable to measure different aspects of value produced by developers. We also suspect that PageRank may be a useful metric to identify code infrastructure and Katz centrality may be best suited to identify code that requires a particular attention and may not be an optimal first assignment for newcomers. We also anticipate using centrality rankings in tasks ranging from reviewer assignments, code refactoring prioritization, knowledge loss and numerous other software development tasks.

7 LIMITATIONS

It is difficult to generalize from studies like ours because they investigate software development in a specific company using specific tools and practices. Therefore, we replicate existing studies from other companies that had substantially different development practices and tools. Section 2 documents key differences we could identify between our and replicated work.

Some of the concepts like task importance and value and the associated measures of productivity are not measurable directly. To increase construct validity, we build indirect evidence by first pointing out conceptual and empirically demonstrated limitations of using traditional measures of output as proxies for productivity and then by presenting models of growth in centrality that appear to represent an intuitive expectation of output growth for an average developer as they gain experience with code and project.

Statistical and network analysis are sensitive to data quality. We undertook an enormous effort to a) clean up the data, b) to employ techniques that would be robust to whatever imperfections may remain in the data after cleaning, and c) used various diagnostic techniques to identify potential problems not addressed by a) or b). Specifically, for data cleaning we used several standard techniques employed at Meta to identify and exclude copious bot-induced activity. For each network, we use diagnostics to identify the nodes with the highest centrality, for example, in the co-change we manually inspected the most central 1000 files and filter out files changed with a lot of unrelated files, such as *changes.txt* containing release notes. We used top nodes in the author-to-file network to identify (and exclude) files with high numbers of authors, such as file containing all global constants.

Meta and many other companies track commits per month as it tends to vary much less than commits at finer granularity (a week) yet is fine enough to capture time trends. Developers' output includes more than code. The presented model can be enhanced by adding other trackable types of outputs, such as review activity, but senior developer tend also to be more involved in training and

coordination tasks that are harder to track. For example, developers' managing others have lower commit output than developers with no reports. The commit was chosen as a fundamental unit of work because at Meta as in In many other companies, it is strongly discouraged to have large commits due to issues with review and testing they pose. To handle vacations months with no output were excluded.

As is commonly necessary for statistical models in software engineering [8, 25, 26, 28, 29], we log-transformed variables with highly skewed distributions and inspected correlations among predictors in the regression as well as inspecting residuals and doing standard regression diagnostics. The only predictors that showed high correlations (above 0.7) involved the various output measures we considered: commits and lines modified per developer per month. As noted above models for each output measures produced similar results. Only commit-based models are presented in the paper. Although R^2 of 30% may not seem very high, we did not include inherent developer variability. Previous research has incorporated random effects for each developer, but the extremely large scale of our dataset was beyond what was possible to analyze using standard tools.

We investigated the stability of the community detection algorithm by plotting the modularity, the number of clusters, and the largest size of the cluster and found no observable jumps (indicating instabilities) in the output.

8 CONCLUSIONS

In 1984 Curtis [13] wrote that until "the many sources of variation among individuals have been compared in the same set of data, it will not be possible to determine … the most important predictor of success." In this work, we argued why SSCs should be used to explain variations in developer output and provide statistical models indicating that it is possible to explain a large fraction of variation using SSC-derived measures. The primary theoretical implication is that even individual developer's output is an SSC property, not an individual property. In this work, we make the following contributions:

(1) We conduct a partial replication of an industry study quantifying output growth when on-boarding: empirical studies in industrial setting are necessary [21, 38] (replication is one of the essentials of the experimental methods and successful replication increases the validity and reliability of the outcomes observed in an experiment) but extremely uncommon [14, 39], and we are not aware of another published industry replication involving software artifacts conducted in a completely different industry context.
(2) We propose a unified concept for the Software Supply Chains that encompasses key measurable types of relationships among people, software artifacts, and tasks.
(3) We theorize why and how SSC networks may capture aspects of the development context that would explain output.
(4) We propose specific ways to characterize (converting into scalar values associated with specific nodes) SSC networks for use in statistical modelling.
(5) We take into account the dynamic nature of the SSCs when characterizing development context.
(6) We suggest global measures of SSC and demonstrate that they explain the variations in developer output better than simpler local measures of the SSC.

Related to the specific findings from this study. In RQ1, we replicated [45] at Meta and found the same plateau in raw counts of number of commits per month. This plateau appears to represent the time it takes for new developers to become reasonably proficient on the project. In contrast, for RQ2, the centrality measure of the number of developers that touched a files used in [45] appears to be too simplistic a as it continues to grow linearly. In contrast, our results from RQ2, follow the Legitimate Peripheral Participation theory that argues that the learners' participation of practice is at first legitimately peripheral but increases gradually in engagement and complexity or, in other words, people engage in more central work as they become more experienced with a system. Meta 's software is massive and complex and even the most experienced developers may still be learning as they work on increasingly important problems that have broader, longer term, and profound impact. Finally, in RQ3, we found that centrality was the best predictor of the variation of individual developer output with an R^2 above 30%. The model's coefficients and SSC centrality measures provide ways to differentiate among SSC nodes such as developers, different parts of the system, and organization, and we hope that in future work these SSCs will lead to actionable decisions that could target specific parts of code or organization.

9 DATA AVAILABILITY

The proprietary nature of the data and systems makes it impossible to release the data.

REFERENCES

[1] [n. d.]. Louvain Method. https://en.wikipedia.org/wiki/Louvain_method.
[2] Can Güney Aksakalli. [n. d.]. https://aksakalli.github.io/2017/07/17/network-centrality-measures-and-their-visualization.html#katz-centrality. https://creativecommons.org/licenses/by-nc-sa/3.0/.
[3] Jens B Asendorpf, Mark Conner, Filip De Fruyt, Jan De Houwer, Jaap JA Denissen, Klaus Fiedler, Susann Fiedler, David C Funder, Reinhold Kliegl, Brian A Nosek, et al. 2016. Recommendations for increasing replicability in psychology. (2016).
[4] D. Atkins, T. Ball, T. Graves, and A. Mockus. 1999. Using Version Control Data to Evaluate the Effectiveness of Software Tools. In *1999 International Conference on Software Engineering*. ACM Press, 324–333. papers/ve
[5] D. Atkins, T. Ball, T. Graves, and A. Mockus. 2002. Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor. *IEEE Transactions on Software Engineering* 28, 7 (July 2002), 625–637. papers/vedraft.pdf
[6] Arthur Gilbert Bills. 1934. *General Experimental Psychology*. Kessinger Publishing. 197-206.
[7] C. Binder, E. Haughton, and B. Bateman. 2002. *Fluency: Achieving true mastery in the learning process.* Technical Report. University of Virginia Curry School of Special Education. Professional Papers in Special Education.
[8] B.W. Boehm. 1981. *Software Engineering Economics.* Prentice-Hall.
[9] B. W. Boehm, B. Clark, E. Horowitz, and et al. 1995. Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. *Annals of Software Engineering* 1, 1 (November 1995), 1–24.
[10] Colin F Camerer, Anna Dreber, Felix Holzmeister, Teck-Hua Ho, Jürgen Huber, Magnus Johannesson, Michael Kirchler, Gideon Nave, Brian A Nosek, Thomas Pfeiffer, et al. 2018. Evaluating the replicability of social science experiments in Nature and Science between 2010 and 2015. *Nature Human Behaviour* 2, 9 (2018), 637–644.
[11] Marcelo Cataldo, Patrick A Wagstrom, James D Herbsleb, and Kathleen M Carley. 2006. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. 353–362.
[12] B Curtis. 1981. Substantiating programmer variability. In *Proceedings of the IEEE 69*.

[13] B. Curtis. 1984. Fifteen Years of Psychology in Software Engineering: Individual Differences & Cognitive Science. In *ICSE'84*. 97–106.

[14] Fabio QB Da Silva, Marcos Suassuna, A César C França, Alicia M Grubb, Tatiana B Gouveia, Cleviton VF Monteiro, and Igor Ebrahim dos Santos. 2014. Replication of empirical studies in software engineering research: a systematic mapping study. *Empirical Software Engineering* 19, 3 (2014), 501–557.

[15] Todd L. Graves and Audris Mockus. 1998. Inferring Programmer Effort from Software Databases. In *22nd European Meeting of Statisticians and 7th Vilnius Conference on Probability Theory and Mathematical Statistics*. Vilnius, Lithuania, 334.

[16] T. J. Hastie and R. J. Tibshirani. 1990. *Generalized Additive Models*. Chapman & Hall.

[17] John PA Ioannidis. 2005. Why most published research findings are false. *PLoS medicine* 2, 8 (2005), e124.

[18] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. 2015. From Developer Networks to Verified Communities: A Fine-Grained Approach. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 563–573. https://doi.org/10.1109/ICSE.2015.73

[19] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. 2015. From developer networks to verified communities: A fine-grained approach. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 563–573.

[20] Maggie Johnson and Max Senges. 2010. Learning to be a programmer in a complex organization: A case study on practice-based learning during the onboarding process at Google. *Journal of Workplace Learning* (2010).

[21] Natalia Juristo and Omar S Gómez. 2010. Replication of software engineering experiments. In *Empirical software engineering and verification*. Springer, 60–88.

[22] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* 18, 1 (1953), 39–43.

[23] J. Lave and E. Wenger. 1991. *Situated Learning. Legitimate Peripheral Participation*. Cambridge University Press, Cambridge.

[24] Stephanie Miceli. 2019. Reproducibility and replicability in research. *I: The National Academies In Focus* 18 (2019), 12–14.

[25] Audris Mockus. 2007. Software Support Tools and Experimental Work. In *Empirical Software Engineering Issues: Critical Assessments and Future Directions*, V Basili and et al (Eds.). Vol. LNCS 4336. Springer, 91–99. papers/SSTaEW.pdf

[26] Audris Mockus. 2008. Missing data in software engineering. In *Guide to Advanced Empirical Software Engineering*, J. Singer et al. (Ed.). Springer-Verlag, 185–200. papers/missing.pdf

[27] Audris Mockus. 2009. Organizational Volatility and Developer Productivity. In *ICSE Workshop on Socio-Technical Congruence*. Vancouver, Canada. papers/orgvolatility.pdf

[28] Audris Mockus. 2010. Organizational Volatility and its Effects on Software Defects. In *ACM SIGSOFT / FSE*. Santa Fe, New Mexico, 117–126. http://dl.acm.org/authorize?309271

[29] Audris Mockus. 2014. Engineering Big Data Solutions. In *ICSE'14 FOSE*. https://dl.acm.org/authorize?N14216

[30] Kjetil Molokken and Magne Jorgensen. 2003. A review of software surveys on software effort estimation. In *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings*. IEEE, 223–230.

[31] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web*. Technical Report. Stanford InfoLab.

[32] Xinyu Que, Fabio Checconi, Fabrizio Petrini, Teng Wang, and Weikuan Yu. 2013. Lightning-fast community detection in social media: A scalable implementation of the louvain algorithm. *Department of Computer Science and Software Engineering, Auburn University, Tech. Rep. AU-CSSE-PASL/13-TR01* (2013).

[33] R Development Core Team. 2008. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org ISBN 3-900051-07-0.

[34] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. 2016. Feature Toggles: Practitioner Practices and a Case Study. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/2901739.2901745

[35] FE Ritter, LJ Schooler, et al. 2002. The learning curve. International encyclopedia of the social and behavioral sciences. In *Amsterdam: Pergamon)*. 8605.

[36] F. E. Ritter and L. J. Schooler. 2002. *International Encyclopedia of the Social and Behavioral Sciences*. Pergamon, Amsterdam, Chapter The learning curve, 8602–8605.

[37] Paige Rodeghero, Thomas Zimmermann, Brian Houck, and Denae Ford. 2021. Please turn your cameras on: Remote onboarding of software developers during a pandemic. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 41–50.

[38] Martin Shepperd, Nemitari Ajienka, and Steve Counsell. 2018. The role and value of replication in empirical software engineering results. *Information and Software Technology* 99 (2018), 120–132.

[39] Dag IK Sjøberg, Jo Erskine Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, N-K Liborg, and Anette C Rekdal. 2005. A survey of controlled experiments in software engineering. *IEEE transactions on software engineering* 31, 9 (2005), 733–753.

[40] Christian L Staudt and Henning Meyerhenke. 2015. Engineering parallel algorithms for community detection in massive networks. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2015), 171–184.

[41] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. 2016. NetworKit: A tool suite for large-scale complex network analysis. *Network Science* 4, 4 (2016), 508–530.

[42] Georg von Krogh, Sebastian Spaeth, and Karim R. Lakhani. 2003. Community, joining, and specialization in open source software innovation: a case study. *Research Policy* 32, 7 (July 2003), 1217–1241.

[43] S. N. Wood. 2006. *Generalized Additive Models: An Introduction with R*. Chapman & Hall.

[44] Yunwen Ye and Kouichi Kishida. 2003. Toward an understanding of the motivation of Open Source Software developers. In *ICSE 2003*. Portland, Oregon, 419–429.

[45] Minghui Zhou and Audris Mockus. 2010. Developer Fluency: Achieving True Mastery in Software Projects. In *ACM SIGSOFT / FSE*. Santa Fe, New Mexico, 137–146. http://dl.acm.org/authorize?309273