Metrics Driven Reengineering and Continuous Code Improvement at Meta

Audris Mockus*†, Peter C Rigby*‡, Rui Abreu*, Anatoly Akkerman*, Yogesh Bhootada*, Payal Bhuptani*, Gurnit Ghardhora*, Lan Hoang Dao*, Chris Hawley*, Renzhi He*, Sagar Krishnamoorthy*, Sergei Krauze*, Jianmin Li*, Anton Lunov*, Dragos Martac*, Francois Morin*, Neil Mitchell*, Venus Montes*, Maher Saba*, Matt Steiner*, Andrea Valori*, Shanchao Wang*, and Nachiappan Nagappan*
*Meta Platforms, Inc.
†The University of Tennessee, Knoxville
‡Concordia University, Montreal

***Abstract*—The focus on rapid software delivery inevitably results in the accumulation of technical debt, which, in turn, affects quality and slows future development. Our primary aim is to discover how companies keep their codebases maintainable and how code improvements might be automated. *Method:* we investigate Meta practices by collaborating with engineers on code quality (via action research) and by analyzing rich source code change history using mixed-methods to reveal a range of practices used for continual improvement of the codebase. *Results:* Code improvements at Meta range from completely organic grass-roots done at the initiative of individual engineers, to regularly blocked time and engagement via gamification of Better Engineering (BE) work, to major explicit initiatives aimed at reengineering the complex parts of the codebase or deleting accumulations of dead code. Over 14% of changes are explicitly devoted to code improvement and the developers are given "badges" to acknowledge the type of work and the amount of effort. Based on the interactions with development teams we suggest metrics to help prioritization of code improvement efforts. Finally, our models of the impact of reengineering activities revealed substantial improvements in quality and speed and reductions in code complexity. Overall, code improvement activities are relatively effort intensive yet simple enough to be prime targets for automation.**

***Index Terms*—Refactoring, dead code, better engineering, gamification**

I. Introduction

Over time, the codebase increases in complexity due to evolution in the functionality, ongoing maintenance, and developer churn [1]. It accumulates technical debt via design decisions that often focus more on the need to resolve issues fast instead of ensuring long-term maintainability [2]. Similarly, new developers may not be familiar with original design decisions [3] and, by making incompatible changes, complicate future maintenance. Focus on rapid delivery, such as continuous integration and deployment [4], place even more emphasis on speed [5]. The presence of these – and similar factors – accelerates the degradation of the codebase, resulting in a tangled web of hacks, workarounds, dead code, and unfinished tasks that ultimately make source code more difficult to maintain. This, in turn, should dramatically slow down the development, defeating the original purpose of rapid delivery. While the practice of rapid delivery is used at Meta, this predicament is avoided. We aim to discover what slows down, prevents, or reverses such natural code decay by studying software code improvement practices at Meta. Furthermore, while we discovered a number of tools to support code improvement, more of these activities may be automated, freeing engineers to work on more complicated tasks.

While a substantial literature exists on how to improve software processes, few studies attempt to discover or catalogue a broad range code improvement practices in a large software company practicing rapid delivery. We employ several methodological approaches each suitable for a part of our study. First, using mixed methods, we analyze the internal documentation, version control, and issue tracking artifacts to identify a range of code improvement activities. Second, we actively participate in code improvement prioritization via action research (AR). Third, we replicate aspects of several prior industry case studies that quantify the impact of code improvement activities.

Specifically, **RQ1:** what types of code improvement practices and activities exist in a large software company? If we discover these practices, it is still not clear how to prioritize them beyond general approaches of reducing the codebase, code complexity, or code smells. Therefore, **RQ2:** what information do engineers need to prioritize major code improvement efforts? While it is relatively easy to identify major top-down code improvement activities, it may be much harder to detect efforts that emanate from individual engineers (if such exist). Finally, were these activities effective, i.e., **RQ3:** what is impact of the code improvement in terms of quality, productivity, lead time, and code centrality? To answer this, we replicate several aspects of previous empirical studies investigating the impact of reengineering [6]–[8].

In summary, by analyzing the text of commit messages we discover bottom-up BE approaches, we participate in the strategic code improvement initiative by helping identify the most central parts of the codebase, we analyze BE efforts by discovering past BE activities recorded in the issue tracking data, and we replicate aspects of a previous industry study [6]–[8] on the impact of reengineering based on the outcomes of explicit reengineering tasks.

We elaborate our contributions in Section **??**, discuss the background on code decay (and improvement) studies in Section II, describe the methods and describe BE practices and

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00268

tooling in Section III. We then turn to prioritization of code improvement efforts in Section IV evaluation of the impact in Section V and discussion in Section VI. We conclude with this study's limitations in Section VII and conclusions in VIII.

Contributions

First, we discover and catalogue a wide range of code improvement efforts within a large company, Meta. Previous catalogs have focused on identifying problem areas (e.g., [9]), developer perceptions (e.g., [10]), or experiences with agile development (e.g., [11]) at a high level. Our contribution lies in the methods used to discover these efforts and the resulting catalog of micro-practices that span from organic, developer-driven initiatives to top-down strategic efforts, as well as policies and tools that support these practices and increase engagement levels. Second, we directly participate in and steer some of the code improvement initiatives. Third, we investigate an important question of how best to allocate the reengineering effort. Previous work primarily focused on code smells or code complexity, e.g., [12], but such narrow focus does not capture the variety of criteria and constraints extant in large software organizations. Fourth, we replicate and extend many elements of the prior studies investigating the impact of reengineering [6]–[8]. Fifth, we use the software supply network for measures of combined call graph, co-change, and authorship centrality as well as direct measures of productivity, development interval, and outages to help engineers prioritize which parts of the code need more attention.

II. Background and Literature

Code Decay and Technical Debt. The observation that software becomes harder to maintain over time is quite old. For example, almost quarter century ago the code decay [1] phenomena was defined as "code is more difficult to change than it should be" and introduced indicators such as excessively bloated code, a history of frequent changes and faults, widely dispersed changes, kludges, and numerous interfaces to measure code decay. That work also discussed causes for software decay that are as relevant today, including inappropriate architecture, violations of the original design principles, imprecise or changing requirements, time pressure, inadequate programming tools, organizational environment, programmer variability, and inadequate change processes with bad project management.

Technical debt [2] is a similar but broader and more recent concept defined in Wikipedia as "technical debt (also known as design debt or code debt) is the implied cost of future reworking required when choosing an easy but limited solution instead of a better approach that could take more time." Tom *et al.* [2] describe several types of technical debt. Code debt which appears to be similar to code decay, e,g., "unnecessary code duplication and complexity, bad style that reduces the readability of code, and poorly organised logic that makes it easy for a software solution to break when updated at a future point in time." The second type is design and architectural debt, such as, "piecemeal design with an absence of reengineering." The third type is environmental debt related to development-related processes, hardware, and other infrastructure and supporting applications. Knowledge distribution and documentation is the fourth type of debt related to developer churn without adequate knowledge transfer, see, e.g., [13]. The fifth type is testing debt manifested as lack of coverage or automation.

While the literature describes numerous indicators of code decay, we view it through a unifying supply chain perspective. The software supply chain concept and operationalization to combine dependency-based (call), logical (co-change), and knowledge (authoring and reviewing) supply changes into a single network [14]. The supply chain perspective helps us to deliberately investigate if the provided indicator set is complete and what part or type of supply chain is involved.

A key question is **how to reverse code decay?** One of the more commonly used techniques is code refactoring and reengineering. We use the term reengineering as a more broad term because many of the BE efforts go beyond refactoring which requires "transforming code without modifying semantics" [15]. In cases where software is not feature-complete and needs active maintenance or, especially, if it needs to accommodate features that were not considered in the original architecture, it may make sense to invest significant development effort to reengineer these parts of the codebase to make it easier to accommodate hitherto not contemplated features and, possibly, to reduce the technical debt accumulated over the years.

Despite the widespread acceptance of phenomena such as code decay and technical debt, surprisingly few studies investigate **the impact of reengineering activities.** For example, a large survey at Microsoft [8] found that engineers understand refactoring in much broader terms than simply semantics preserving code transformation (hence our use of the term "reengineering") and that reengineering entails substantial costs and risks [16]. The investigation found that the most reengineered binaries had reduced numbers of dependencies and reduced number of faults. Improvements in productivity and quality have been documented in earlier studies of reengineering, e.g., [6]. Reengineering may result in a more transparent codebase where it is easier not to overlook some unanticipated effects of a code change. Specifically, the study in [6] looked at reengineering 30 KLOC C++, ASN.1 generated code from a 3rd party protocol stack within a 7 MLOC system. The relevant part was modified by 40 different developers over five years. The effort resulted in virtual elimination of defects reported by end users, halving in the number of lines used to implement exactly the same functionality, and reduced the effort to implement an MR (an equivalent of a pull request) by 11%. A later study by Moser et al [7] found a similar result.

Most other studies of reengineering look at code metrics before and after reengineering, such as reduction in code smells [17], but the improvements in the code smell metrics may not correlate with any reductions in effort once reductions in total code size are accounted for [18]. Our contribution consists of cataloguing and reporting a variety of code improvement practices in a large company.

Once code decay is identified and suitable approaches to

remediate it are chosen, the question on **how to prioritize the remediation efforts** remains. Specifically, engineer-driven efforts may focus on parts of the system they are familiar with and where they do not need to coordinate their changes with numerous other developers. A study of Java projects [19] found that developers most frequently cited changing requirements as the main motivation for refactoring and an even larger study [20] confirmed that structural metrics including code smells do not play a significant role in refactoring decisions. The most reengineering benefits may, however, come from reengineering the most complex parts of the system where numerous engineers, often from different organizations, are actively making changes. Hence some of the reengineering work may need to be done in a top-down manner with a careful evaluation of the potential risks and rewards. Our contribution is to develop multiple criteria and associated operationalizations to help prioritize code improvement efforts.

Finally, **what can we expect as a result of code improvement methods?** Very few company studies exist, e.g., [6]–[8] and each may have been affected by the particular organizational context. Replications are essential to establish the generality of the findings in different contexts [21]–[25]. We, therefore, conduct a replication of the study by Geppert *et al.* [6].

III. Code improvement practices at Meta

A. Context: software development at Meta

We first describe the key aspects of the software development process and tools needed to explain the data acquisition and analysis presented later. At a high-level, Meta has a company-wide search engine that, as regular internet search engines, allows search for any keywords. Second, documentation is tracked in online documents, and company-wide wikis, as well as interactive training tutorials. Third, the same systems for tracking issues and code changes are used company-wide. Fourth, data associated with the usage of these tools is tracked in a data warehouse that has SQL access.

At Meta, we develop software for both our servers and client devices, including specialized hardware devices. This approach allows us to have fine-grained control over versioning and configurations, and enables us to quickly push new code updates to production. Before any code is deployed, it undergoes rigorous testing, including peer review, in-house user testing, automated tests, and canary tests. Once the code is deployed, engineers closely monitor logs to identify potential issues.

At Meta, we place a strong emphasis on code reviews as part of our development process. In addition to using IDEs, such as VS Code, version control system (mercurial) and numerous other testing and deployment tools, we use a central continuous integration system, which facilitates modern code reviews. Developers submit their code for review, creating a patch representing the initial version of the code. Reviewers can suggest improvements, leading to additional revisions until the Pull Request (PR) is either approved and incorporated into the codebase or rejected. This process promotes high coding standards, helps detect flaws, and spreads knowledge throughout the organization.

In addition to our focus on code reviews and testing, we also have a formal process for reporting and addressing bugs, outages, or incidents. By having a clear process for reporting and addressing these issues, we can ensure that we maintain the quality and reliability of our products.

B. Method for RQ1

To investigate company-wide code improvement initiatives we used mixed methods by gathering information from internal company documents that include meeting notes, planning documents, tutorials, wikis, issues, PRs (the term PR refers to a logical change to one or more files that can be reviewed, has a test plan, etc.) , and other documentation. We start from keyword “improving code” and retrieve and inspect 20 top results. We then inspect the resulting documents for indications if they are describing a code improvement tool or practice and use them to formulate additional keywords, such as, “better engineering,” “code quality,” “code health,” “code coverage,” “code complexity,” and “refactoring,” The procedure is then repeated for each of these subsequent keywords. We reached saturation and did not find new keywords that identified interesting documents and code.

To discover bottom-up efforts we search for specific PR tags and also for keywords in the title. Code improvement is “perfective” maintenance: “code changes made with intention to make future changes easier” [26]. We, therefore, started from keywords associated with it in [27], such as “cleanup”, “unneeded”, “remove”, “rework”. We also added synonyms, such as “delete”, more modern keywords such as “refactoring” and “dead code” as well company-specific terms such as “better engineering” and “BE.” We then sampled at least 20 PRs that contain these keywords as tags or in their title and refined the search to exclude occasional enhancements, like “add deleted code.” Finally, we investigated a mirror-image of code improvement, the so-called self admitted technical debt [28] by using keywords from that paper. Finally, we discuss the results with engineers from various parts of the company to check if our search missed any code improvement practices.

C. Results for RQ1

a) Documents and automation: We discovered an internal course, wikis, announcements for code improvement events, and tools to support and measure code improvement and also to help increase engineer engagement for it. The materials contained topics ranging from improving coding style, code documentation, and code complexity, but also on practices for code reviews, testing, and code coverage. In addition to descriptions of best practices, we found a number of automation tools that support, measure, and report code improvement activities. Some of the tooling was explicitly targeting engineer engagement, such as profile badges for reviewing PRs, reporting bugs, having high code coverage and so on. In addition to the best-practices document, we also found larger initiatives, such as Better Engineering (BE). The following is a brief summary of BE at Meta.

b) Policies: Better Engineering at Meta is a company-wide initiative started in 2016 to improve engineering productivity in different codebases and tools. It drives improvements in tooling, codebases and culture to improve productivity and engineering efficiency. The size and complexity of Meta's codebases are increasing dramatically as it rapidly grows the worldwide engineering team, and continuous investment in this area is needed to keep the teams productive. Better Engineering is designed to improve engineering productivity across Meta by keeping code modern with consistent abstractions and frameworks. It also creates a culture that recognizes and fixes engineering issues that slow teams down. It helps Meta improve and optimize the developer experience and builds a sense of pride around the code and tooling. In general, Meta's guideline is that teams should allocate 20% to 30% of engineering effort to BE projects. The work on BE is important and is recognized. BE is also supported by training, tooling and even gamification elements with profile badges (See Figure 1) and scoreboards to encourage (and reward) participation [29], [30]. Gamification (drawing from game design) is the addition of gamified elements to a system such as badges or levels. A related concept of a public profile or "the quantified self [31]," which draws from wearables in the health realm and big data in the business realm, allow public display of various accomplishments.

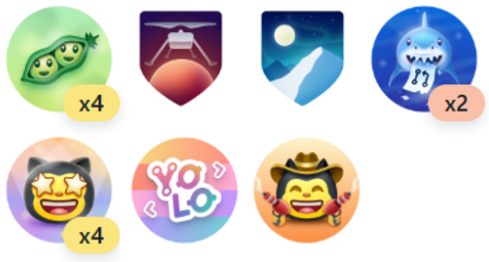

Fig. 1. An engineer's profile may reach 70 or more badges. The images of badges shown internally are replaced with the analogues "Achievement" badges from GitHub. The set of internal badges is quite rich and includes badges such as "Better Engineering (BE) gold": "completed 20 tasks for High-Priority issue types" and "Code Cleaner — Purple: deleted 100,000 or more lines of code"

TABLE I
PERCENT OF PRS CAPTURED BY EACH KEYWORD. THERE IS SOME OVERLAP IN SELECTED PRS USING DIFFERENT KEYWORDS AS THE PERCENTAGES SUM UP TO 16.5% WHILE THE TOTAL PERCENTAGE OF PRS AFFECTED BY ANY OF THE KEYWORDS IS 14.2%.

Keyword	% of the PRs selected
remove, delete, unneeded	7.26%
better engineering	4.28%
clean	3.01%
refactor, rework	1.47%
dead	0.52%

c) Analysis of PRs: The investigation of the "perfective" code change activity showed that a significant 14% of PRs are devoted to perfective maintenance (see Table I. Tag or title bearing "Better Engineering," is in 4.3% of all PRs, "refactoring" in 1.5%, and dead code removal in less than 1% of the PRs, however. This suggests that most of the perfective maintenance is organic and is not a direct result of specific initiatives. Interestingly, despite the focus on rapid development, we found very little presence of self-admitted technical debt, with fewer than one in 10K PRs containing any such keywords.

In summary, we found code improvement activities to take many forms, ranging from major quality initiatives that are well documented, to individual engineer driven actions that are less visible. Some of the code improvement initiatives target the strategically important parts of the codebase by producing a variety of indicators engineers could use to prioritize which parts of the codebase to target as described in Section IV. In addition to courses, tutorials, extensive documentation, and regular scheduled code improvement activities, code improvement is supported by tools that help track code metrics over time and also include various ways to engage developer via profile badges and even gamification of BE activities.

IV. TARGETING CODE IMPROVEMENT EFFORTS

Reengineering may be costly in effort and may not always have a significant impact (or, at least, challenging to quantify). Furthermore, the effort that could be devoted to reengineering initiatives such as BE is inherently limited for actively maintained codebases. As such, it is critical to pick the reengineering tasks in a way to maximize the return on investment or to minimize risks and maximize rewards. Specifically, we are concerned with parts of the code that should be considered first when deciding on the scope and focus of the reengineering effort.

A. Method for RQ2: prioritization criteria

While any code could be improved, the impact of that improvement may vary. Code decay and technical debt literature suggests starting with the most decayed and fragile parts of the system, specifically, where changes are made by numerous authors, are difficult to complete, take a long time, and are likely to cause faults. Such areas may not be the first choice for organic individual developer-driven efforts due to the substantial investment of time and the need to coordinate changes to such parts of the code with multiple organizations. To find such areas we used a number of indicators proposed in the literature (see Section II) to measure decay and technical debt, with examples shown in Table II. One novel indicator we also employ is the software supply chain (SSC) centrality [14] as it appears to be strongly related to quality, lead-time, and productivity. Obviously, making any modifications to such parts of the code is a high-risk activity (we discuss risks below) and should also be considered in the prioritization.

We ranked the top 300 files in terms of our proposed criteria and asked the senior engineers to provide their feedback. We then conducted several iterations refining the metrics with Table II, showing the final operationalizations of our criteria through multiple metrics. Specifically, in each iteration, we presented the metrics for the files to a team of engineers via a spreadsheet and ordered them by centrality/importance. The

engineers could resort to them based on other criteria and provide feedback on the files. The results and reflections of this genuine collaboration with participants are presented below.

B. Definition and extract of metrics

a) Change activity: As we observed in practice, the most decayed parts of the codebase may, however, be rarely or never changed. The potential benefits of being able to make changes faster, with less effort, and lesser chances of a fault are thus greatly diminished or eliminated (if no change will be needed in the future). Thus the total benefit of reengineering should take into account not just the effort savings for an individual change but also multiply it by the anticipated number of changes in the future. Actively changed files serve as a proxy of future effort and, even small improvements in such areas should yield significant overall results due to the frequency of changes. Reengineering of frequently changed code should yield much larger relative (per change) improvements to justify the reengineering effort and risk associated with reengineering.

b) Knowledge loss: It is important not to underestimate several hidden benefits of reengineering related to knowledge loss. Specifically, parts of the system originally designed and maintained by engineers who no longer work on the project carry significant risks [13]. Even a single change, if needed, may lead to serious issues and require significant effort. Reengineering such parts of the system makes current project members more aware of the specific design decisions needed to understand its operation and to simplify its maintenance. Each developer, by changing the codebase both learns from it and also imparts their own understanding. As such, parts of the code where a lot of changes were done by developers who left Meta may not be well understood by anyone who is still with Meta and require, if not reengineering, but at least a strong ownership.

c) Authoring Speed: PR Authoring Time (PAT) tries to capture exactly how much time it takes to author/write and land a PR. We, therefore, argued that in order to maximize the speed, the files that have the largest normalized PAT should be reengineered first.

d) Centrality: Different parts of the codebase often vary in importance. The core or central functionality requires better quality controls and, generally, more attention. Hence, reengineering it may bring the most benefit. Previous work found that engineers perceive several distinct dimensions of code centrality [32] and several measures of code centrality were shown to be important predictors of engineer productivity [14]. Code supply chain centrality can be calculated on a combined network of code dependencies, author-to-changed-file, and co-changes (all files changed in a single commit). These networks may also be considered separately resulting in multiple centrality measures, such as call-graph centrality and co-change centrality. For example, in a study of code decay [1] numerous interfaces was one of the indicators of decay, while a Microsoft study [8] considered the number of dependencies among binaries. Both of these represent "degree centrality" (the number of adjacent nodes), but we use Katz centrality [33], which also takes into account centrality of the adjacent nodes as well. Specifically, we prioritize the parts of the codebase that are the most central with respect to SSC network, are actively changed (and are expected to be so in the future), have prior changes that lead to outages, and have experienced a significant knowledge loss [13]. Each dimension is described in more detail below.

e) Reliability: At Meta, all outages are investigated to identify their trigger. In case the trigger is a code change, the offending PR is noted in a special SEV database. Outages are extremely infrequent hence only a tiny fraction of PRs are associated with SEV. Hence files that have high severity or large numbers of SEV fixed in them may need to be considered a priority. At the same time, modifying them carries high risk of introducing an outage.

C. Engineer feedback on metrics for reengineering

We present representative examples the feedback that engineers gave us on the metrics.

- For many of the most central files in the spreadsheet, the feedback was "yes, this is a really bad file and we are already improving (are planning to improve) it."
- We also got more nuanced responses, such as (slightly paraphrased) "This one feels better than the other. It is touched a lot: any new logic in class X has to go through this file. But it is not really that complex: just a large function registering stages one by one."
- "This file has small complex parts that are not touched as much and very simple parts touched on day-to-day. Here is where analysis at the granularity of individual methods can help."
- "This is just an enum with 400 entries and a map. Not sure how it ended up here. There definitely bigger enums (maybe not changed as frequently?)" Regarding the bundling of files together we had a response: "I would argue that the cpp file is more complex and central than the header... should be bundled as a single unit in all cases." However, in another case, a comment for .h file:
- "Solid "bad" file, but not sure if we want to bundle with all of its cpps files."
- Regarding the persistence of improvements we had: "Looks like this one has been simplified a lot over the last 1-2 years. Not sure if it still should be so high up in the list". Also, "Yes, this file is bad. People keep adding independent structs to it all the time. It is mainly bad for build speed. At some point I split a whole bunch of them into separate files, but it didn't cause a change in developers behaviors"

Centrality was a good starting point for investigation because it gives a general sense of importance of the files. Engineers were then able to use the other metrics, for example, size and complex logic, to determine if the central files should be refactored. We did not get the feedback that we missed important files, hence the filtering criteria using recent and numerous changes and authors appear to be sound. The results

TABLE II
VARIOUS PRIORITIZATION CRITERIA PRESENTED WITH EACH CANDIDATE FILE. ENGINEERS MAY SORT THE LIST USING ANY COLUMN.

Column	Definition	Purpose
TotPAT	PAT sum over all PRs (over the last two years) modifying the file	Shows the overall time in PRs that modify the file.
AvgPAT	Geometric average: $\exp(average(\ln(PAT))$	Geometric average over PRs: shows average PR time
totNormPAT	$\sum {}^{PRPAT}/_{PRNfiles}$ normalized PAT where the focus file takes only 1/files modified by a PR fraction of PAT	For each PR attributes 1/files modified proportion of PAT to the file
avgNormPAT	geometric average of the normalized PAT: $\exp(average(\ln({}^{PAT}/_{nfiles})))$	Geometric average over PRs for normalized PAT: shows average PR time attributable to file
NPR2Y	Number of PRs modifying the file over past two years	Some files have been deprecated or less often changed in recent past, hence should be less a focus of refactoring
nSEV	Number of SEV-related PRs modifying file	Refactoring may improve quality
SEV_{level}	Highest severity over all SEVs	
nAuthor	Number of authors	Files with many authors may indicate potential coordination problems
nPRs	Number of PRs (over the entire history)	
nMnth	number of months during which there were PRs (over the last two years)	Is the file being constantly modified?
PRsPerMonth		Intensity of activity – to compare recent and older files
fr	Date of the first PR (within last two years)	
to	date of the last PR (within last two years)	Has the file been changed recently?
min and $\max_{cent}$	Lowest and highest file centrality over past two years	Centrality trend
Avgdc	average difference in file and developer centrality (over the last two years).	Developers with low centrality should have many more problems with the high-centrality files, so positive values for avgdc may indicate problems.
Pagerank and centrality	represent code dependencies	These network measures represent to what extent central or isolated the file is in the dependency network
knowLost	percent of PRs by authors who are no longer with Meta	High fraction may suggest the need to strengthen ownership
AuthLeft	percent of authors who are no longer with Meta	
sloc	lines of code	to gauge the size of the file
complexity	cyclomatic complexity	to gauge complexity of the file (highly correlated with size)
nFilePerPR	average number of files modified by PRs touching the file	Is the file relatively isolated or tied to other files?
nTotAuth	Total number of authors over the last two years	
TopCochanged	Other files that appear in at least 20% of the PRs	To show if the files is tightly logically connected to other files

of these metrics and feedback is to develop dashboards that support prioritization by displaying some key metrics.

V. THE IMPACT OF CODE IMPROVEMENT

Once we have discovered various types of code improvement and ways to prioritize it, a natural question arises: **RQ3:** what is the impact of code improvement in terms of quality, productivity, lead time, and code centrality?

A. Method for RQ3

We employ an industry case study approach for RQ3 and do empirical analysis of several types of reengineering tasks completed in a single six-month period. The tasks were assigned by a group at Meta engaged in efforts to continuously improve its source code (the same group we engaged for RQ2). Various goals for code improvement are set and tasks created every six months. For the studied period the tasks involved four types of code improvements: dead code removal, Cyclomatic Complexity Number (CCN) decomposition, large class decomposition, and platformization as described in Table III.

As outcome variables (see Table IV) we investigated SEV rate, PR Authoring Time (PAT), number of editing sessions, and how code complexity and centrality of the modified codebase has changed as a result of this reengineering. Our two primary aims were 1) to quantify the impact the reengineering had on quality (SEV), authoring time, and structural properties of the source code, and 2) to replicate prior industry studies of the code reengineering impact.

a) Data Collection and Measures: As described in Section III-A, the software changes are often a result of explicit tasks tracked in Meta's task tracking system. The links between tasks and changes are carefully tracked and are routinely used to determine the effort needed to complete the tasks or, as in our case, to evaluate their impact. We started by identifying PRs associated with the code improvement tasks and then used these PRs to identify all modified files (we refer to them as "reengineered files"). PRs associated with the reengineering tasks are referred to as "reengineering PRs". We then identified all PRs modifying reengineered files during three months prior to and after the intervention (we refer to them as pre- and post-reengineering PRs correspondingly). We then obtain reengineered file properties before and after the reengineering and compare them as well as properties of pre- and post-reengineering PRs.

Since many of the files have been renamed, removed,

TABLE III
DESCRIPTION OF THE TYPES OF REENGINEERING TASKS.

Type	Description	Risks
Dead code removal	Unused methods, data fields, classes, build targets, configs, feature flags, etc. detected via internal static analysis tools, then checking team-specific guidelines, and consulting with code owners [34].	"dead" actually used in emergency; "dead" actually work in progress
Cyclomatic Complexity, CCN-based decomposition	The focus is on ways to reduce code complexity, and includes a variety of tasks such as refactoring fields or classes, unifying access to class fields, moving code to appropriate location, refactor a class into more logical units, organize functions and fields into more intuitive groups, and many other types of modifications	Due to the variety of the tasks combined under this label, the risk would vary with task type.
Large class decomposition	The action is to do a subclass extraction, where tightly connected code in a "super" class is extracted and moved into a newly created class.	To reduce risk typically done in three steps: (a) declaring a new class with all data dependencies (b) moving method by method (or all methods at once) to a new class (retaining their body in the original file); (c) moving methods to a new class implementation file.
Platformization	The action is to replace Service Locator pattern using dependency inversion to enable a subsequent migration to a, for example, DAG execution framework that can properly capture data dependencies and state mutation; decompose god-objects and replace service-locator usage of god-objects with standard Dependency Injection	Service Locator usage suffers from tight coupling and implicit data dependencies, making code and system state difficult to reason about

or created by the reengineering PRs, we could not directly compare pre to post changes at the individual file level. Instead, we compare the distribution of code metrics over the entire set of reengineered filenames as they appeared both before and after the change. For example, metrics for deleted files were included to obtain pre-intervention distribution and metrics for created files in the post-intervention distribution. For comparisons of process metrics, such as review time, we compared all PRs modifying at least one of the reengineered files over a three-month period immediately preceding the intervention and a three month period immediately following the intervention. We also obtained all PRs that modified at least one of the files involved in reengineering and marked which of these PRs caused a SEV. One of the key challenges was that a substantial number of the files have been renamed as is expected for the nature of the changes. We need to include past data on the renamed files to fully account for the codebase and activity before and after the code improvement takes place. To simplify that task, we always use the files current name, i.e., for past PRs we convert the filename at the time of the event to the current filename. We thus obtained full association between files and all PRs that modified these files in the past and in the future as well as all PRs that caused SEV.

We use the code metrics database table to obtain the cyclomatic complexity number (CCN) prior to the intervention and its latest value. Finally, we obtain review time for each PR that modified at least one of the files involved in reengineering. We use Fisher's exact test for contingency tables [36] and regular (and paired where appropriate) t-test or paired Wilcoxon tests [37] to compare the pre- and post-reengineering states.

Code centrality can be calculated on a combined supply-chain network of code dependencies, author-to-changed-file, and co-changes (all files changed in a single commit). These networks may also be considered separately resulting in multiple centrality measures, such as call-graph centrality and co-change centrality. We use Katz centrality [33] as it takes into account not just the number of neighbors but also their importance. Comparing centrality before and after the intervention is complicated, because centrality depends on the entire graph, not just on the refactored files. The set of nodes will be different as some of the files will be added and others deleted (we use the same ID for renamed files). Furthermore, we use a normalized measure of centrality that forces all values to range from 0 to 1, essentially dividing all values by the (unnormalized) centrality of the most central node in the entire graph. Even small changes in centrality of that node (even if it is not directly related to the refactored files), will affect centrality values for the remaining nodes.

To compare the centrality of the reengineered files before and after the intervention, we selected a random sample of not-reengineered files that matched the distribution of the refactored set in terms of file size, type, and centrality. This is a widely used case matching method that helps with estimating causal effects from observational data [38]. Specifically, it is desirable to replicate a randomized experiment as closely as possible by obtaining treated and control groups with similar covariate distributions. In other words, for each refactored file f_r, we find a file f_{nr} of the same size and programming language that has similar centrality in the period before refactoring. We then compare

$$c_{adj}(f_r, t_{pre}) = c(f_r, t_{pre})/c(f_{nr}, t_{pre})$$

with $c_{adj}(f_r, t_{post})$, where $c(f, t)$ is the centrality of file f at time t and t_{pre}, t_{post} are times before and after reengineering.

TABLE IV
DESCRIPTION OF THE UNITS AND MEASURES USED.

Unit	Metric	Explanation
PR	H1: SEV-trigger	whether or not the PR was determined to be a cause of a SEV
PR	H2: PAT	PR Authoring Time starts with the first trace of activity that can be traced to a PR (e.g. first VS Code session) and ends with the last session. Work on sessions connected to other PRs is excluded.
PR	H3: Number of Sessions	The number of code editing sessions obtained from PAT
File	H4: Supply Chain Centrality	Katz centrality [33] of co-change, author-to-file, and file-to-file call-graph (function definition to function invocation) network
File	H5: Cyclomatic Complexity	The number of linearly independent paths [35]

b) Hypotheses: The measures obtained as described in the previous section were based on certain hypotheses concerning expected impact of reengineering. The primary objective of reengineering is to make it less likely that serious bugs are introduced. Hence:

H1: We expect the simplified code base will lead to a lower chance that a PR will cause a SEV;

Additional advantages of the simplified codebase is that it is more clear where and how to make changes, resulting in a shorter authoring time:

H2: PAT will be lower after the code is rejuvenated, and

H3 The simplified code base will streamline PRs requiring fewer code editing sessions.

With limited published evidence it is not obvious how reengineering may affect centrality. Since centrality reflects relative importance of the file, it may not change. Furthermore, if only the files that are anticipated to be more important in the future are reengineered, the centrality may increase after reengineering. Some of the reengineering tasks involved refactorings of large classes creating multiple files (for each refactored class). These files may contain calls among them and be co-changed, thus increasing centrality. Similarly, refactoring repeated code into a single function may also increase centrality as the resulting function would be called from all locations where that code was repeated. Hence:

H4: We expect the supply-chain centrality to increase after reengineering.

One of the intended outcomes of reengineering is that the most complex parts of the code will be simplified. Hence:

H5: We expect that the cyclomatic complexity of the reengineered codebase will be lower;

B. Results for RQ3

The engineering teams took on over 1000 reengineering changes and modified over 1000 files. Table V provides a summary of the number of changes per intervention/metric type.

To answer H1, the fraction of PRs modifying any of the reengineered files that caused an SEV are in Table VI. The table shows statistically significant decreases that occurred only for files affected by dead code removal and for CCN-driven decompositions. These improvements are substantial.

Substantial improvements in quality have been previously documented in other studies of reengineering, e.g., [6]. Often this is a result of a more transparent codebase where it is easier not to overlook some unanticipated effects of a code change.

TABLE V
DESCRIPTIVE STATISTICS

Tasks	Number of PRs	Number of modified files
Dead code removal	193	316
CCN-driven decompositions	897	789
Large class decompositions	262	114
Platformization	136	132

TABLE VI
H1: SEV RATE. THE ODDS RATIO IS 5.2 FOR DEAD CODE REMOVAL ($p < 0.01$) AND 1.55 FOR DECOMPOSITIONS ($p < 0.01$) SHOWING 90% AND 55% DECREASE IN SEV-CAUSING PRS MODIFYING REENGINEERED FILES. THE RESULTS ARE NOT STATISTICALLY SIGNIFICANT FOR PLATFORMIZATION AND LARGE CLASS DECOMPOSITIONS (FISHER EXACT TEST)

Type	Period	PRs with no SEVs	PRs triggering SEVs
Dead code removal	Before	57%	76%
	After	43%	24%
CCN-driven decomp.	Before	72%	80%
	After	28%	20%
Large class decomp.	$p > 0.05$		
Platformization	$p > 0.05$		

TABLE VII
H2: PAT. THE DECREASE IS THE AUTHORING TIME IS STATISTICALLY SIGNIFICANT WITH REENGINEERED CODE TAKING LESS TIME TO AUTHOR (MANN-WHITNEY TWO SAMPLE TEST)

Type	Relative change
Dead code removal	0.59, $p < 0.01$
CCN-driven decomp.	0.23, $p < 0.01$
Large class decomposition	0.41, $p < 0.01$
Platformization	0.52, $p < 0.01$

To answer H2, a simpler codebase should also make it easier and more straightforward to write the code. Indeed, Table VII shows 41% to 77% reduction in PAT for all types of refactoring. We expect that this shorter authoring time will require fewer code editing sessions (and fewer time-consuming task context switches [39]).

To answer H3, the number of sessions shown in Table VIII go down the most for Platformization and Large class decom-

position. Dead code removal had the smallest but still sizable effect.

TABLE VIII
H3: IDE CODE EDITING SESSIONS. THE MEDIAN NUMBER OF IDE SESSIONS DECREASED FOR REENGINERED CODE (T TEST)

Type	Relative change
Dead code removal	0.19, $p < 0.01$
CCN-driven decompositions	0.41, $p < 0.01$
Large class decomposition	0.67, $p < 0.01$
Platformization	0.65, $p < 0.01$

To answer H4, we find that adjusted centrality shows significant increases as shown in Table IX. Several reasons for the increase were noted earlier: it makes sense to reengineer more important files and splitting classes and refactoring repeated code are likely to increase centrality. Particularly large increases for "Large class decomposition" appear to support this conjecture. Several contributors to centrality, such as co-change and author-to-file relationships need to be collected over a significant time period, thus reengineering may not have an immediate effect and needs to be observed over a longer period. We, however, found similar increases in the call-flow centrality (which does not require history) as well.

TABLE IX
H4: CENTRALITY. WE FIND AN INCREASE IN THE CENTRALITY OF RENGINEERED CODE (WILCOXON PAIRED TEST))

Type	Avg adj increase in centrality
Dead code removal	50%, $p < 0.01$
CCN-driven decompositions	100%, $p < 0.01$
Large class decomposition	113%, $p < 0.01$
Platformization	95%, $p < 0.01$

In addition to process measures, such as SEV, PAT, and the number of code editing sessions, we expect the structure of the codebase to change as well. Specifically, (per H5) since many of the tasks were targeting cyclomatic complexity, we expect it to be lower after reengineering. As expected, we see the largest improvements (of 26%) for CCN-driven decompositions, but even dead code removal and large class decompositions also lead to decreases as shown in Table X.

TABLE X
H5: CCN. EXCEPT FOR PLATORMIZATION, WE FIND A DECREASE IN CYCLOMATIC COMPLEXITY FOR REENGINEERED CODE (WILCOXON PAIRED TEST)

Type	Avg decrease in CycCmplex
Dead code removal	7%, $p < 0.01$
CCN-driven decompositions	26%, $p < 0.01$
Large class decomposition	7%, $p < 0.01$
Platformization	$p > 0.05$

VI. DISCUSSION

It is not surprising that code quality receives significant attention and, at Meta, multiple courses, tutorials, wiki pages are devoted to the topic. Similarly, software engineers like to create tools to support their work, including work on code improvements. Hence we see numerous specialized tools that are explicitly devoted to code quality that go beyond the traditional version control, issue tracking, code review, build, and deployment tooling. For example, the dead code and data detection and removal tool helped eliminate millions of lines of code and petabytes of data [25]. Many of the tools and practices at Meta are deployed across the entire organization and not, as is more typical, siloed within product units. While individual products have their specialized needs, leveraging common tools (and adapting them to serve these special needs) allows for a much larger number and variety of tools than a single product unit could support. It would be interesting to study the relative impact of various initiatives, such as considering code improvement in performance reviews, displaying progress in personal badges and gamification tools, on the intensity and quality of code improvement efforts. Despite the highly visible efforts to promote better engineering practices, most of the code improvement effort is done outside this umbrella. Notably, our study was focused solely on code improvement and did not consider the much larger space of unique company-specific developer productivity tools.

By discussing the question of targeting code improvement efforts, including major architectural changes to a large codebase, we arrived at over 20 distinct measures that could be used to identify parts of code that, if improved, would yield largest dividends. The main idea is that actively changing code that is high risk and takes a lot of effort should be the focus of reengineering, rather than code with code smells that is peripheral, rarely changed, and is not involved in outages. While peripheral code may be easier to reengineer, the impact of such efforts is likely to be low. While some criteria are simple and obvious, like complexity and size, many were not previously considered in the literature for the task of prioritizing code rework. We also found that supply chain network properties, in many cases, provided partial indicators of problematic areas.

Our attempt to replicate prior code reengineering study also found substantial improvements in quality and speed. Perhaps the most unexpected result was the unusually high fraction of effort for perfective maintenance. Other industry case studies are needed to confirm that the focus on rapid delivery also requires substantially more perfective maintenance to ensure the codebase can be easily changed with minimal risk.

VII. THREATS TO VALIDITY

a) Generalizability: Generalizing conclusions from a case study in software engineering is complex because of a large number of potentially relevant context variables. The analyses in the present paper were performed at Meta, and it is possible that outcomes would differ elsewhere. We cannot release our data, even in an anonymized format, because it would violate legally binding employee privacy agreements. However, our study involves a variety of products and developers. The software systems involved have millions of lines of code and 10's of thousands of developers who are both collocated and working

at multiple locations across the world. We also cover a wide range of domains from user facing social network products and virtual and augmented reality projects to software engineering infrastructure, such as calendar, task, and release engineering tooling. To determine if the findings generalize, a comparison with the results obtained elsewhere is needed. We find that many, but not all our findings have been consistent with prior work in industry settings, increasing the confidence that the results generalize. Some, for example a much larger fraction of perfective maintenance, may be unique to Meta or may be more common at present.

b) Construct Validity: In our study, we used the outcome measures that are commonly used at Meta such as outages, PRs, code complexity, and PR duration metric (PAT). The PAT metric leverages existing tools in Meta to get an accurate estimate of time spent working on a PR. We also use more complex collaboration measures relying on software supply chains and centrality [40]–[45].

We analyzed all files touched by reengineering PRs, but some of the modified files may not have been the actual targets of reengineering, but were changed together with the reengineered files. We do not expect many such instances as the reengineering initiative was undertaken separately from the regular coding and maintenance activities. Organic reengineering efforts, on the other hand, might be undertaken as part of regular coding activities and are more likely to include files that are not targets of reengineering.

c) Internal Validity: We have used only the most basic statistical tests and checked when needed if the assumptions on the distributions were reasonable. As is commonly necessary for statistical models in software engineering [46]–[50], we log-transformed variables with highly skewed distributions. Despite the fact that our hypotheses are based on findings in prior industry studies (and, consequently, we are not trying to introduce new relationships), we still use a stronger (than the commonly used 0.05) threshold for p-value of 0.01 to serve as a form of Bonferroni correction due to the number of the hypotheses that are being tested.

VIII. Conclusion

Across Meta and the broader software engineering community there is sparse data on how much code improvement activity happens, how it is organized or encouraged, how it is (or should be) prioritized, and what impact it has on key software engineering outcomes. We conduct a multifaceted analysis of the reengineering effort undertaken at Meta, by doing a search for related practices, tools, and reward mechanisms. We also conduct a bottom-up approach to quantify the relative number of changes corresponding to different types of code improvement.

Code improvement activities do take many forms, ranging from major quality initiatives that are well documented, to individual engineer driven actions that are less visible. In addition to courses, tutorials, extensive documentation, and regular scheduled code improvement activities, extensive tool support is provided to help track development activity and code quality metrics over time and also include various ways to engage developer via profile badges and even gamification of code improvement activities. We found that over 14% of the changes were made explicitly for code improvement: substantially higher than previously reported 4% in [27]. We further discuss how to target strategically important parts of the codebase via many criteria (some not previously reported in the literature) producing a variety of indicators engineers could use to prioritize their work. Such indicators may play a role in making major investments in code improvement and, at the same time, may serve as quantitative metrics to evaluate automated code improvement.

Finally, we analyze the reengineered files to track the impact of the reengineering work over time. We observe several types of these targeted activities ranging from removal of dead code to sophisticated major restructuring aimed to achieve greater modularity by inverting dependencies. We found significant reductions in authoring time (PAT) and the number of coding sessions for the code targeted by the reengineering work, substantial reductions in SEV incidence, and reductions in code complexity. Supply chain centrality, on the other hand, increased. This may represent the importance of the reengineered files or may be simply a result of large class decompositions (that break up large files) and refactoring repeated code.

Our findings suggest that code improvement and reengineering practices need to be continuous in nature and supported by tools, as, for example, continuous build, to counteract potential issues that inevitably get introduced in active and rapid development. Since many code improvement tasks are relatively simple, tedious, and effort-intensive, they may be good candidates for GenAI tools. In particular, various measures we proposed may be used automatically to evaluate solutions generated by GenAI.

References

[1] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE transactions on software engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[2] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, 2013.

[3] J. Herbsleb and A. Mockus, "Formulation and preliminary test of an empirical theory of coordination in software engineering," in *2003 International Conference on Foundations of Software Engineering*. Helsinki, Finland: ACM Press, October 2003. [Online]. Available: http://dl.acm.org/authorize?787510

[4] B. Fitzgerald and K.-J. Stol, "Continuous software engineering and beyond: trends and challenges," in *Proceedings of the 1st International Workshop on rapid continuous software engineering*, 2014, pp. 1–9.

[5] J. Taplin, *Move fast and break things: How Facebook, Google, and Amazon have cornered culture and what it means for all of us*. Pan Macmillan, 2017.

[6] B. Geppert, A. Mockus, and F. Robler, "Refactoring for changeability: A way to go?" in *11th IEEE International Software Metrics Symposium (METRICS'05)*. IEEE, 2005, pp. 10–pp.

[7] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A case study on the impact of refactoring on quality and productivity in an agile team," in *IFIP Central and East European Conference on Software Engineering Techniques*. Springer, 2007, pp. 252–266.

[8] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[9] R. L. Hackbarth, A. Mockus, J. D. Palframan, and D. M. Weiss, "Assessing the state of software in a large enterprise," *Empirical Software Engineering*, vol. 15, pp. 219–249, 2010.

[10] E. K. Smith, C. Bird, and T. Zimmermann, "Beliefs, practices, and personalities of software engineers: a survey in a large software company," in *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*, 2016, pp. 15–18.

[11] D. Leffingwell, *Scaling software agility: best practices for large enterprises*. Pearson Education, 2007.

[12] R. Malhotra, A. Chug, and P. Khosla, "Prioritization of classes for refactoring: A step towards improvement in software quality," in *Proceedings of the Third International Symposium on Women in Computing and Informatics*, 2015, pp. 228–234.

[13] P. C. Rigby, Y. C. Zhu, S. M. Donadelli, and A. Mockus, "Quantifying and mitigating turnover-induced knowledge loss: case studies of chrome and a project at avaya," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1006–1016.

[14] A. Mockus, P. C. Rigby, R. Abreu, P. Suresh, Y. Chen, and N. Nagappan, "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*, 2023, pp. 1809–1819.

[15] M. Fowler, *Refactoring*. Addison-Wesley Professional, 2018.

[16] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2012, pp. 104–113.

[17] J. Al Dallal and A. Abdin, "Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 44–69, 2017.

[18] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2012.

[19] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, 2016, pp. 858–870.

[20] J. Pantiuchina, F. Zampetti, S. Scalabrino, V. Piantadosi, R. Oliveto, G. Bavota, and M. D. Penta, "Why developers refactor source code: A mining-based study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–30, 2020.

[21] R. Wieringa and M. Daneva, "Six strategies for generalizing software engineering theories," *Science of computer programming*, vol. 101, pp. 136–152, 2015.

[22] J. M. González-Barahona and G. Robles, "On the reproducibility of empirical software engineering studies based on data retrieved from development repositories," *Empirical Software Engineering*, vol. 17, pp. 75–89, 2012.

[23] A. Mockus, B. Anda, and D. I. Sjøberg, "Experiences from replicating a case study to investigate reproducibility of software development," in *Proceedings of the 1st International Workshop on Replication in Empirical Software Engineering Research*, 2010.

[24] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, "Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm," *Information and Software Technology*, vol. 99, pp. 164–176, 2018.

[25] W. Shackleton, K. Cohn-Gordon, P. C. Rigby, R. Abreu, J. Gill, N. Nagappan, K. Nakad, I. Papagiannis, L. Petre, G. Megreli *et al.*, "Dead code removal at meta: Automatically deleting millions of lines of code and petabytes of deprecated data," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1705–1715.

[26] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd international conference on Software engineering*, 1976, pp. 492–497.

[27] Mockus and Votta, "Identifying reasons for software changes using historic databases," in *Proceedings 2000 international conference on software maintenance*. IEEE, 2000, pp. 120–130.

[28] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 91–100.

[29] J. Hamari, J. Koivisto, and H. Sarsa, "Does gamification work?–a literature review of empirical studies on gamification," in *2014 47th Hawaii international conference on system sciences*. Ieee, 2014, pp. 3025–3034.

[30] K. Seaborn and D. I. Fels, "Gamification in theory and action: A survey," *International Journal of human-computer studies*, vol. 74, pp. 14–31, 2015.

[31] D. Lupton, *The quantified self*. John Wiley & Sons, 2016.

[32] M. Zhou and A. Mockus, "Developer fluency: Achieving true mastery in software projects," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 137–146.

[33] L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, vol. 18, no. 1, pp. 39–43, 1953.

[34] W. Shackleton, K. Cohn-Gordon, P. C. Rigby, R. Abreu, J. Gill, N. Nagappan, K. Nakad, I. Papagiannis, L. Petre, G. Megreli, P. Riggs, and J. Saindon, "Dead code removal at meta: Automatically deleting millions of lines of code and petabytes of deprecated data," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1705–1715. [Online]. Available: https://doi.org/10.1145/3611643.3613871

[35] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[36] R. A. Fisher, "Statistical methods for research workers," in *Breakthroughs in statistics: Methodology and distribution*. Springer, 1970, pp. 66–70.

[37] M. Holander and D. Wolfe, "Nonparametric statistical methods, vol. 497, john wilew and sons inc," *Publications, New York*, 1973.

[38] E. A. Stuart, "Matching methods for causal inference: A review and a look forward," *Statistical science: a review journal of the Institute of Mathematical Statistics*, vol. 25, no. 1, p. 1, 2010.

[39] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 19–29.

[40] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu, "Putting it all together: Using socio-technical networks to predict failures," in *2009 20th International Symposium on Software Reliability Engineering*. IEEE, 2009, pp. 109–119.

[41] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 531–540.

[42] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "Categorizing bugs with social networks: a case study on four open source software communities," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1032–1041.

[43] P. V. Singh, "The small-world effect: The influence of macro-level properties of developer collaboration networks on open-source project success," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 2, pp. 1–27, 2010.

[44] A. Sureka, A. Goyal, and A. Rastogi, "Using social network analysis for mining collaboration data in a defect tracking system for risk and vulnerability analysis," in *Proceedings of the 4th india software engineering conference*, 2011, pp. 195–204.

[45] S. Wang and N. Nagappan, "Characterizing and understanding software developer networks in security development," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 534–545.

[46] B. Boehm, *Software Engineering Economics*. Prentice-Hall, 1981.

[47] A. Mockus, "Missing data in software engineering," in *Guide to Advanced Empirical Software Engineering*, J. S. et al., Ed. Springer-Verlag, 2008, pp. 185–200. [Online]. Available: papers/missing.pdf

[48] ——, "Software support tools and experimental work," in *Empirical Software Engineering Issues: Critical Assessments and Future Directions*, V. Basili and et al, Eds. Springer, 2007, vol. LNCS 4336, pp. 91–99. [Online]. Available: papers/SSTaEW.pdf

[49] ——, "Organizational volatility and its effects on software defects," in *ACM SIGSOFT / FSE*, Santa Fe, New Mexico, November 7–11 2010, pp. 117–126. [Online]. Available: http://dl.acm.org/authorize?309271

[50] ——, "Engineering big data solutions," in *ICSE'14 FOSE*, 2014. [Online]. Available: https://dl.acm.org/authorize?N14216