Social and Technical Sources of Risk in Sustaining Digital Infrastructure

Kaylea Champion

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2024

Reading Committee:

Benjamin Mako Hill, Chair

Kirsten Foot

Andrea Forte

René Just

Program Authorized to Offer Degree:
Communication

University of Washington

Abstract

Social and Technical Sources of Risk in Sustaining Digital Infrastructure

Kaylea Champion

Chair of the Supervisory Committee:
Benjamin Mako Hill
Communication

Significant risks to our shared digital infrastructure—communication systems, servers, and applications—can be identified by examining the social and technical conditions of the communities which produce that infrastructure. Exploration of these production communities reveals the deeply contingent processes of collective action that sustain them—processes that are innovative and powerful but sometimes fragile. As this shared body of digital infrastructure has grown, some crucial pieces have become neglected, leading to *underproduction*: the phenomenon of highly important, low-quality software packages. Underproduction is a form of what I will call a *social production failure*, and a substantial source of risk to digital infrastructure that today is used by billions of people. This dissertation is framed around a series of methodological and empirical projects. The first proposes a method for measuring underproduction risk in cross-section and demonstrates the application of that method to the Debian GNU/Linux community. Next, I examine the social and technical conditions of the Debian community and test hypotheses about how these conditions are associated with underproduction. I then develop a method to measure underproduction longitudinally, and apply this method to projects in Debian written using the Python programming language. I close by synthesizing these results with respect to my proposed theory of social production failures, and offer propositions and proposals for future work.

ACKNOWLEDGMENTS

This work would not have been possible without the generosity of the Debian GNU/Linux community and the many practitioners who joined me in reflecting on this problem and these results. I am indebted to these volunteers who, in addition to producing Free/Libre Open Source Software software, have also made their records available to the public. I also gratefully acknowledge support from the Sloan Foundation through the Ford/Sloan Digital Infrastructure Initiative, Sloan Award 2018-11356 as well as the National Science Foundation (Grant IIS-2045055). This work was conducted using the Hyak supercomputer at the University of Washington as well as research computing resources at Northwestern University.

This work was only completed due to the kindness and support of a crowd. My committee (Kirsten Foot, Andrea Forte, René Just, Charlotte P. Lee) and especially my chair and advisor Benjamin Mako Hill have tirelessly mentored me to grow and become myself as a scholar in ways that have been a surprise and a delight. My colleagues in the Community Data Science Collective (especially Wm Salt Hale, Aaron Shaw, Jeremy Foote, Molly de Blanc, Emilia Gan, Ellie Ross, Matthew Gaughan, Sohyeon Hwang, Floor Fiers, Nathan TeBlunthuis, Stefania Druga, Yibin Fan, Haomin Lin, Kevin Ackermann, Hazel Chu, Nicholas Vincent, and Dyuti Jha) were constantly encouraging and a source of guidance and motivation. Morten Warncke-Wang's kindness—and his work in Wikipedia—has been a tremendous source of encouragement and inspiration throughout this project. My colleagues in the Social Computing Reading Group (especially David McDonald and Mark Zachry) supported my quest for interdisciplinary rigor. Early comments from Matthew Powers have continued to reverberate and encourage me to continue to pursue explanation and an understanding of the

social.

My family has supported me from the very beginning—my husband David, my children Ari, Mira, and Milo, my mom Patsy, and my brother Nate. The journey has been strange and complicated, and you put up with me being busy on so many days. Thank you. Chad Kainz supported me from so early I can't number all the ways. Anita Nikolich was there for me at just the right moment and inspires me to this day. Jeremy Szteiter was a calm voice who always asked the right question. Peter J. Taylor, you were a bright light of support that always managed to cut through the fog of doubt: I wish you were here to see me finish the work you never doubted I could do.

DEDICATION

This dissertation is dedicated to the myriad of public goods contributors but most particularly to my favorite among all of them—my husband, David Champion.

TABLE OF CONTENTS

Page

List of Figures viii

List of Tables xii

Chapter 1: Introduction 1

Chapter 2: Background 5
2.1 Collective Action 5
2.2 Commons-based Peer Production 6
2.3 Communication Public Goods 7
2.4 Social Production Failures 8
2.5 Underproduction 10
2.6 The Case of Digital Infrastructure 11
2.7 Dissertation Overview and Structure 13

Chapter 3: Underproduction: An Approach for Measuring Risk in Open Source Software 15
3.1 Introduction 16
3.2 Background 19
3.2.1 Detecting and Measuring Software Risk 19
3.2.2 Peer Production and FLOSS 20
3.2.3 Systematic Comparison of FLOSS in Software Repositories 20
3.3 Conceptual Framework: Underproduction 21
3.4 Empirical Setting 23
3.5 Application of Framework 25
3.5.1 Step 1: Assemble a Collection of Artifacts 25
3.5.2 Step 2: Identify a Measure of Quality 25

3.5.3 Step 3: Identify a Measure of Importance . . . 30
3.5.4 Step 4: Select a Baseline Relationship . . . 31
3.5.5 Step 5: Measure Deviation . . . 31
3.6 Results from Experiments . . . 32
3.6.1 Experiment 1: Identifying Underproduced Software . . . 32
3.6.2 Experiment 2: Validation Using Alternate Indicator . . . 37
3.7 Threats to Validity . . . 38
3.8 Discussion . . . 40
3.8.1 The Long Tail of GUI Underproduction . . . 40
3.8.2 Implications for Software Engineering Research . . . 41
3.8.3 Implications for Practice . . . 41
3.9 Conclusion . . . 42

Chapter 4: Sources of Underproduction in Open Source Software . . . 44
4.1 Introduction . . . 45
4.2 Background . . . 46
4.2.1 The Production of Free/Libre Open Source Software . . . 46
4.2.2 Alignment Between Supply and Demand in Open Source Software . . . 47
4.3 Methods . . . 52
4.3.1 Empirical Setting . . . 52
4.3.2 Data . . . 54
4.3.3 Measures . . . 57
4.3.4 Analytic Plan . . . 61
4.3.5 Ethics . . . 62
4.4 Results . . . 62
4.4.1 H1, H2, and H3: Age . . . 62
4.4.2 H4: Contributor Count . . . 64
4.4.3 H5: Maintainer Turnover . . . 64
4.4.4 H6: Organizing into Teams . . . 64
4.4.5 H7 and H8: Collaboration Networks . . . 66
4.5 Discussion . . . 66
4.5.1 The Role of Technology Choices in Underproduction . . . 66
4.5.2 Organizing to Address or Prevent Underproduction . . . 67

4.5.3 Key Takeaways for Practitioners . . . 68
4.6 Limitations . . . 69
4.7 Conclusion . . . 71

Chapter 5: Detecting Risk Inside Projects Using Underproduction Measures . . . 72
5.1 Introduction . . . 74
5.2 Background . . . 75
5.2.1 Peer Produced Open Source Software . . . 75
5.2.2 Underproduction Analysis . . . 76
5.2.3 Relationship to Existing Methods . . . 79
5.3 The Dynamic Underproduction Analysis Method . . . 79
5.4 Methods . . . 80
5.4.1 Empirical Setting . . . 80
5.4.2 Data and Measures . . . 80
5.4.3 Analytical Plan . . . 81
5.4.4 Ethics . . . 84
5.5 Results . . . 84
5.5.1 Underproduced Python Packages . . . 84
5.6 Discussion . . . 87
5.6.1 Interpreting Underproduction (and Overproduction) from Multiple Perspectives . . . 87
5.6.2 Implications for Research . . . 88
5.6.3 Implications for Practice . . . 89
5.7 Limitations . . . 89
5.8 Conclusion . . . 89

Chapter 6: Concluding Reflections . . . 91
6.1 A Theory of Social Production Failure . . . 92
6.1.1 Connections to Market Failure . . . 93
6.1.2 Connections to Social Failure . . . 94
6.1.3 Propositions on Social Production Failure . . . 95
6.2 A Research Agenda for Investigating Social Production Failure . . . 98
6.3 Implications for Research . . . 100
6.4 Implications for Regulators and Civil Society Leaders . . . 102

6.5 Implications for Practitioners . . . 102

Bibliography . . . 103

Appendix A: Methods Notes from Chapter 3 . . . 132

Appendix B: Underproduction Case Studies . . . 137
B.1 Kazam: The Worst Case Scenario . . . 137
B.2 Pip: The Median Case . . . 141
B.3 SEAMicro Client: The Most Overproduced . . . 143
B.4 Looking Across Cases . . . 144

Appendix C: Countering Underproduction of Peer Produced Goods . . . 146
C.1 Introduction . . . 146
C.2 Background . . . 147
C.2.1 Commons-based Peer Production and Underproduction . . . 147
C.2.2 Motivation, Experience, and Task Selection . . . 148
C.3 Research Design . . . 151
C.3.1 Empirical Setting . . . 151
C.3.2 Data . . . 152
C.3.3 Measures . . . 153
C.3.4 Analytical Plan . . . 154
C.3.5 Ethics . . . 154
C.4 Results . . . 154
C.5 Limitations . . . 160
C.6 Discussion . . . 161
C.6.1 Born, Made, or Something Else? . . . 161
C.6.2 How May Underproduction Be Countered? . . . 162
C.6.3 Problematizing Retention and Account Creation . . . 164
C.7 Conclusion . . . 165
C.8 Supplement to Appendix C . . . 166
C.8.1 Additional Notes on Measures . . . 166
C.8.2 Impact of Filtering Population-Level Dataset . . . 167
C.8.3 Distribution of Fixed Effects . . . 168

C.8.4 Building Reasonable Hypotheticals 168
C.8.5 Interpretation using hypotheticals 170
C.8.6 Divergent Paths for Long-term Contributors 171

Appendix D: Qualities of Quality: A Tertiary Review of Software Quality Measurement Research 173
D.1 Introduction 173
D.2 Related Work 174
D.3 Review Methodology 175
D.3.1 Automated Screening 176
D.3.2 Title, Abstract, Quality, and Full-Text Screening 177
D.3.3 Synthesis Through Thematic Analysis 179
D.4 Findings: Three Perspectives on Quality 180
D.4.1 Quality Research is a Methodological Challenge 182
D.4.2 Quality Research is Oriented to Real-World Outcomes 186
D.4.3 Quality Research is Holistic 193
D.5 Findings: Trends in Conducting and Publishing Secondary Reviews on Software Quality 194
D.6 Findings: Shared Challenges 197
D.6.1 Popular Measures of Questionable Validity 198
D.6.2 Missing Vital Details in ML Studies 199
D.6.3 Data Sources 199
D.7 Discussion 200
D.7.1 The Three Perspectives on Quality 200
D.7.2 Roadblocks to Evidence-Driven Engineering 202
D.7.3 Toward a Theory of Software Quality 203
D.7.4 Implications for Software Engineering Research 205
D.7.5 Implications for Software Engineering Practice 205
D.8 Threats to Validity 206
D.9 Conclusion 207

Bibliography 211

Appendix E: Institutional Statements as a Driver for OSS Project Success: A Bayesian Replication and Extension of Yin et al.'s "Open Source Software Sustainability" . . . 219
E.1 Introduction . . . 219
E.2 Background . . . 220
E.2.1 The Yin et al. Analysis . . . 220
E.3 Methods . . . 222
E.3.1 Data . . . 222
E.3.2 Measures . . . 222
E.3.3 Analytic Plan . . . 224
E.3.4 Ethics . . . 231
E.4 Results . . . 231
E.4.1 Institutional Statements . . . 231
E.4.2 Discussion Topic Prevalence . . . 233
E.4.3 Causal Effect of Mentors' Institutional Statements . . . 233
E.5 Discussion . . . 235
E.5.1 Replication Observations . . . 235
E.5.2 The Role of Mentors' Institutional Statements . . . 236
E.5.3 Implications for Software Development Organizations . . . 236
E.6 Limitations . . . 237
E.7 Conclusion . . . 238
E.8 Bayesian Methods Supplement for "Institutional Statements as a Driver for OSS Project Success: A Bayesian Replication and Extension of Yin et al.'s 'Open Source Software Sustainability"' . . . 238
E.8.1 Institutional Statements . . . 239
E.8.2 Speculative Causal Analysis . . . 242

Appendix F: Engineering Formality and Software Risk in Debian Python Packages . 247
F.1 Introduction . . . 248
F.2 Background . . . 249
F.2.1 Governance in CBPP . . . 249
F.2.2 Governance of FLOSS Engineering . . . 250
F.2.3 Governance and Underproduction . . . 251
F.3 Methods . . . 253

F.3.1 Empirical Setting 253
F.3.2 Data 253
F.3.3 Measures 254
F.3.4 Analytic Plan 256
F.4 Results 256
F.4.1 H_A: Formality Score 256
F.4.2 H_B: Concentration of Developer Responsibility 257
F.4.3 H_C: Formal Work Process Management 257
F.5 Discussion 258
F.6 Limitations and Future Work 259
F.7 Conclusion 259

LIST OF FIGURES

Figure Number Page

3.1 A conceptual diagram locating underproduction in relation to quality and importance. 17

3.2 A Kaplan-Meier curve that shows the number of bugs of different severities that remain open over time. 28

3.3 Credible intervals for U_j for every package in Debian. All packages whose CIs include zero are "aligned"; those whose CIs are entirely above 0 are labeled "underproduced;" those whose CIs are entirely below zero are labeled "overproduced." 33

3.4 A heatmap of software alignment. Color intensity indicates the number of packages occupying a given ranking of quality and installation. Aligned packages appear along the lower-left to upper-right diagonal. The top heatmap includes all packages, while the bottom heatmap contains only those packages for which the 95% credible interval does not cross zero. 34

3.5 Packages displaying the highest mean levels of underproduction. Boxplots show the mean and interquartile range of the distributions of U_j and reflect uncertainty in the model of package-level quality. 36

4.1 A piece of the free/libre open source software supply chain. Software is typically developed "upstream", and then numerous software programs are packaged and integrated by Debian developers before being distributed as part of an operating system or using package management tools. Users may also directly install software from source files or precompiled binaries without the benefit of a package manager (not shown). 54

4.2 Visualizing package age based on when the package was added to Debian, with a generalized additive model (GAM) line to indicate a moving average. . . . 55

4.3 Violin plot of the data distribution broken down by the most commonly appearing languages. See Table 4.1 for models which test the relationship between language age and underproduction. This visualization contains data for 2,280 packages. On 135 occasions, the same package appears multiple times because it was consistently tagged as having been implemented in more than one language. 56

4.4 This visualization shows predicted underproduction probability from model M4 for two prototypical packages of different programming language ages where package age varies as shown along the x-axis. The package shown in blue is 25 years old, corresponding to a package written in a language as old as Java, while the package shown in red is 48 years old, corresponding to a package written in a language as old as C. The gray ribbon shows a 95% confidence interval around the prediction. 65

5.1 A conceptual diagram locating underproduction in open source software in relation to quality and importance, reproduced from Champion and Hill (2021) and annotated. In a cross-sectional analysis of underproduction, each dot would represent a different package–in this longitudinal approach, each dot instead represents the state of a package at a different point in time. 77

5.2 A conceptual diagram describing underproduction in open source software longitudinally for a simulated package's quality and importance trajectory. Underproduction analysis suggests that the package was at substantial risk at the period in its lifecycle where importance exceeded quality; given the drop in importance at the far ride side of the plot, current risk is low. We annotate the figure with dots corresponding to the same dots that appear in Figure 5.1. 78

5.3 Most recent values of quality and importance for each package in our dataset. The packages in blue are particularly concerning because their LOESS-predicted quality slope is negative while the LOESS-predicted importance slope is positive. 85

5.4 The predicted trajectories for quality and importance for our dataset's most concerning examples of underproduction. 85

A.1 Credible intervals for U_j for every package in Debian. We treat all packages whose CIs include zero as aligned; those whose CIs are entirely above 0 are labeled 'underproduced;' those whose CIs are entirely below zero are labeled 'overproduced.' 133

A.2 Packages displaying the highest mean levels of underproduction using "vote" as a measure of importance. Boxplots show the mean and interquartile range of our distributions of U_j and reflect uncertainty in our model of package-level quality. 135

A.3 A heatmap of software alignment. Color intensity indicates the number of packages occupying a given ranking of quality and importance ("vote"). Aligned packages appear along the lower-left to upper-right diagonal. The top heatmap includes all packages, while the bottom heatmap contains only those packages for which the 95% credible interval does not cross zero. 136

B.1 Major events in the life of kazam 138

B.2 Quality and Importance of kazam; SD units 141

B.3 Quality and Importance of Pip; SD units 142

B.4 Quality and Importance of Python-Seamicroclient; SD units. 144

C.1 The marginal effect of having higher experience on the average alignment of an article selected for editing. Increasing values indicate increased levels of underproduction, i.e. low-quality but highly-viewed topics. 155

C.2 The left pane shows a GAM smoothed line fit to a 10% sample of the data from the random sample with the result of the linear model superimposed as a dashed line; the right pane shows the same GAM line with the result of the polynomial model superimposed as a dotted line. Note the log-scaled x-axis. 157

C.3 Marginal effect of increased experience on contributor task selection from our within-person sample. We use median individual-level fixed effects. Dashed lines are predicted values using a linear model, while dotted lines use a quadratic model, see Table C.2. 159

C.4 The distribution of fixed effects in our model. 170

C.5 Task selection trajectories for a random sample of individuals. 172

D.1 Stages of the systematic literature search and screening process 176

D.2 Visualizing secondary study coverage. Each of the 75 secondary studies we identified are represented as a line. A blue dot indicates year of publication, segment color indicates perspective on quality, and segment length indicates the duration of the primary studies they examine. Y-axis labels give the study identifier and the number of studies examined by each. 209

D.3 The three perspectives we identified operate at progressive layers of abstraction, from methods-oriented, to outcome-oriented, to making broad or holistic assessments. The evidence at each layer is often cumulative with respect to previous layers. 210

E.1 The union of all Granger relationships as described in Yin et al. (2022). . . . 226

E.2 Speculatively adding the outcome variable *graduation* as connected to each technical measure. 228

E.3 Dropping variables based on guidance from the Dagitty library to identify bad controls versus good controls, and focusing on mentor institutional statements simplifies our analysis. 230

E.4 Predicted log-odds of graduation associated with increases in average monthly institutional statements from mentors, committers, and contributors. 'a' is the model intercept (the log-odds of graduating with members of the group having made 0 institutional statements) and 'bg' is the log-odds associated with incremental change in the number of institutional statements. Red lines indicate an 80% confidence interval and black lines indicate a 95% confidence interval. 240

E.5 The prior we set on our model generates the logistic regression curve seen in black. Red indicates data simulated from the provided prior. Actual data in blue. 241

E.6 Our posterior model generates the logistic regression curve seen in green. Red indicates data simulated from the provided prior. Actual data in blue. 242

E.7 These traceplots display the behavior of the MCMC chain when fitting the institutional statements model for mentors, committers, and contrbutors. . . 243

E.8 Predicted log-odds causal effect on graduation of mentor institutional statements. Red lines indicate an 80% confidence interval and black lines indicate a 95% confidence interval. 243

E.9 The prior we set on our model for the causal effect of institutional statements generates the logistic regression curve seen in black, all controls set to the median of observed value. Red indicates data simulated from the provided prior. Actual data in blue. 245

E.10 Our posterior model for the causal effect of mentor institutional statements generates the logistic regression curve seen in green, all controls set to the median observed value. Red indicates data simulated from the provided prior. Actual data in blue. 245

E.11 Traceplot of the MCMC chains associated with fitting Causal estimation of Mentor Institutional Statements. 246

F.1 Plot showing the relationship between projects' MMT and mean underproduction factor. Plotted lines are drawn from the results of our model for H_B. 262

LIST OF TABLES

Table Number Page

3.1 Predicting non-maintainer upload (NMU). 37

4.1 These logistic regression models assess the extent to which underproduction is a function of a range of social and technical factors. Coefficients are untransformed log-odds estimates with a 95% confidence interval indicated in brackets. Note that the number of observations varies per model due to missing data. 63

5.1 Underproduced packages from our analysis. 86

C.1 Results from our revision-level hierarchical model of average alignment level of articles selected for editing. Bracketed values indicate a 95% confidence interval. 156

C.2 Results of cross-sectional panel modeling to examine within-person change in the alignment of articles selected. Bracketed values indicate a 95% confidence interval using robust standard errors. The projected R^2 indicates modeling this data with random effects for experience level but without fixed effects for the individual would result in poor model fit. 158

C.3 Dimensions of technosocial learning in peer production with examples. We might expect contributors to learn both how to contribute and about underproduction through these channels. Observing public interest and noticing low quality are implicit signals available without creating an account. 163

C.4 Population sizes after filtering steps, in sequence. 169

D.1 Search keywords 177

D.2 Screening Criteria 178

D.3 Articles present after screening stages 178

D.4 Reviews in each theme 181

D.5 Venues publishing the most software quality reviews. 196

E.1 Coefficients from fitting a Bayesian logistic regression where the predictor is increased mean monthly institutional statements from mentors. 231

E.2 Coefficients from fitting a Bayesian logistic regression where the predictor is increased mean monthly institutional statements from committers (bg). . . . 232
E.3 Coefficients from fitting a Bayesian logistic regression where the predictor is increased mean monthly institutional statements from contributors. 233
E.4 Coefficients from fitting a Bayesian logistic regression where the predictor is the project's centered mean percent of monthly discourse within each of 12 topics. All Rhat values are 1 (omitted for space concerns). 234
E.5 Coefficients from fitting a Bayesian logistic regression to assess the causal effect of increased institutional statements by mentors. Controls are graph density, num nodes, and weighted mean degree. All Rhat values are 1 (omitted for space concerns). 235
F.1 This table displays the relationships between underproduction and three project metrics: formality, MMT, and milestones. Models for H_B and H_C include a control for age with factor variables; 0 to 9 years old is the baseline category. 261

Chapter 1

INTRODUCTION

At 12:48 p.m. GMT on July 2, 2021, a bug report arrived in a bug tracking system, carbon copied to the security team's issue tracker. In the report, *Alpha* described a series of security vulnerabilities in dovecot[1], the most widely-used software project for IMAP email servers[2]—if you read your mail in a desktop client other than a web browser today, you probably use IMAP and the server you connect to may well use dovecot.[3]

The word dovecot may be more familiar as a home for doves, but those of us with access to digital infrastructure likely encounter the software version more frequently than we do its namesake; often the ubiquity of technology in our lives can make it invisible. Much of the software on which we rely, like dovecot, is not only omnipresent, but free and built on top of free tools. We may also easily forget that behind the smooth daily operation of any email system is a complex world of system administrators, network engineers, software developers, hardware operators, and support specialists–themselves part of an overlapping set of volunteer groups, corporations, government agencies, and nonprofit organizations.

The recipients of the dovecot bug report were part of the self-directed volunteer community that develops Debian GNU/Linux—itself a carefully curated collection of over 20,000 pieces of software, tested and integrated to form a coherent operating system released for free to the public. Debian has been available since 1993, and today forms the backbone of the global web and cloud.[4] *Alpha*'s bug report included a list of links, many of which referenced

---

[1]https://www.dovecot.org/

[2]https://en.wikipedia.org/wiki/Internet_Message_Access_Protocol

[3]https://en.wikipedia.org/wiki/Dovecot_(software)

[4]https://en.wikipedia.org/wiki/Debian

official CVE numbers; CVEs are issued by the MITRE organization and used globally to keep track of security vulnerabilities. MITRE, in turn, is a non-profit corporation funded by the US Department of Homeland Security as part of their cybersecurity program.[5] The CVEs varied in their seriousness, but all of them could be fixed by installing the newest version of dovecot. These bugs had been announced and fixed by dovecot's developers—themselves a mix of volunteers and company-sponsored contributors—some six months before; the fixed version was still in the process of making its way out into the world. Hence *Alpha*'s message.

Individual email system administrators who had heard about the problem could apply the update immediately by hand, but often a software update is only installed in a live system once a tested and integrated package is available from a distribution like Debian. One reason for the slow pace of change is the complexity of the systems involved. Modern servers rely on thousands of packages, and each package may have dozens of versions, subversions, and sub-subversions, each with their own needs and problems. Meanwhile, new security vulnerabilities and associated fixes are announced each day—the CVE database has been in operation since 1999 and contains over 160,000 different vulnerabilities.

Updating even a single package on a system can trigger a wave of other changes, each potentially incompatible with the others in perhaps undocumented ways, dragging the system administrator in circles of wasted trial-and-error and potentially triggering system downtime. Further, any effort expended on a manual update might need to be repeated multiple times in the future. For Debian to distribute a fixed version of dovecot already integrated with other packages would be a significant benefit to both system administrators and the users relying on them.

The response from Debian came about two weeks later, on a Saturday evening. *Zeta* from the Debian Quality Assurance team asked Debian's dovecot team: could a fixed version of dovecot be incorporated into the next official Debian release? The next release was at that time in the end stages of testing. Official releases in Debian are termed *stable*, and are

---

[5]https://en.wikipedia.org/wiki/Mitre_Corporation

only issued about every two years. The following Monday, *Epsilon*, a member of the Debian dovecot team, confirmed that the timing was indeed very tight—implementing the change was complex and invasive. Doing so would require changes elsewhere, perhaps triggering their own maelstrom of updates—but they were working on it. The next morning, *Epsilon* confirmed success: the team at Ubuntu, another GNU/Linux distribution and one which inherits packages from Debian but often maintains these inherited packages with its own set of updates, integrations, and features, had already put in the effort to "backport"—forcing the necessary fix into an older version of the dovecot codebase: a version that had been tested as compatible with everything else that was almost ready for release. The security fix could be accomplished without endangering the release schedule. *Epsilon* would go on to make sure the changes propagated back upstream from Ubuntu into Debian, not only for the upcoming release but also in the previous official Debian release as well as Debian's rolling development version. The fix to dovecot would go on to be one of the numerous changes included in the official release of Debian version 11 on August 14, 2021.[6]

What is perhaps most remarkable about this story is that it is wholly unremarkable. Coordination of effort among disparate entities with different priorities and preferences flows almost noiselessly from screen to wire to screen crossing continents. Often with nary a meeting or contract in sight, software bugs and vulnerabilities are found and solved, repairs and workarounds are shared, and new versions are released as older ones are end-of-lifed. However, multiple alternate narratives were possible in this story and in all of the stories not included here.

The developers of dovecot might have written it without the bug to begin with; the problem could have been found sooner; the problem could have come to Debian's attention sooner; Debian might have replied more quickly; the fix might have been easier to write; the fix from Ubuntu could have been supplied to Debian proactively; the release cycle could have been shorter and hence the fix could have been disseminated more widely more quickly.

---

[6]https://www.debian.org/releases/

The bugs in dovecot could have been exploited to disrupt communication; perhaps they were exploited unnoticed. Further, what if *Alpha*, the person prompting the update, hadn't bothered? What if *Zeta* from Debian Quality Assurance and then *Epsilon* from the Debian dovecot team hadn't responded? What if the available fix had not been found, or could not have been so readily added to the existing cycle of test and release?

The deeply contingent yet coordinated action we observe in the maintenance of what I will call digital infrastructure—important components whose names we may not know but whose steady functioning is necessary for communication as we know it to continue—is often voluntary and self-organized (Eghbal, 2016). Ultimately, it is "human infrastructure" that sustains digital infrastructure (Lee et al., 2006). How can we know if this complex system is breaking down, such that difficulty, inattention, overuse, and burnout are clustering together, placing entire realms of our shared digital infrastructure at risk?

In this dissertation, I will show that significant risks to our shared digital infrastructure can be identified by examining the social and technical conditions of the communities which produce and maintain that infrastructure. Just as physical infrastructure undergirds trade and transportation, so also digital infrastructure is today the key support structure for our communication and commercial activities. Maintaining that infrastructure through cooperative effort is critical to society. Neglecting to do so, such that quality is low while importance is high, results in *underproduction*, a phenomenon I will characterize as a *social production failure.*

Chapter 2
BACKGROUND

The social and technical dynamics of communities producing communication public goods acting as infrastructure are a reflection of the kind of work they undertake, the priorities, preferences, and values of their participants, and larger social processes of organizing. To unpack this dynamic process, I draw from a range of theories originating from organizational communication, management, economics, sociology, and social computing in order to ultimately characterize underproduction as a social production failure.

2.1 Collective Action

Social theories up through the 1960s often assumed that because people are fundamentally sociable, we 'naturally' form groups to accomplish shared goals, including producing public goods through collective action. This assumption often went unchallenged and unsubstantiated until Olson (1965) pointed out that not only do many shared goals linger without being achieved (a collective action failure), collective action itself in many cases would be irrational because of the challenges of monitoring others' contributions and the temptation to free ride. Hence, Olson predicts that without selective incentives provided only to contributors or coercion from governments, public goods will not be created.

Marwell and Oliver (1993) developed a substantial challenge to Olson—observing that although not all collective action problems are solved, a great many are—sometimes through selective incentives and coercion, but also through a range of social phenomena: communities and organizations, mutual coordination, norms, altruism, and shared commitments. Rather than being bound to a fixed set of propositions and doomed by fear of free ridership as Olson (1965) predicts, Marwell and Oliver (1993) find that collective action depends on such

factors as the nature of the good being produced, the size of the group, the group's varying levels of interest and resources, the presence of a core group, and the "production function" of the public good, that is, the relationship between marginal contributions of resources and marginal progress toward the provisioning of the public good.

In this dissertation, I extend Marwell and Oliver's (1993) notion of the production function from the conceptual realm to the analytical realm (treating it as a supply function), and I expand this theory to include the notion of a 'consumption' function (treating it as a demand function). The analysis of production and consumption — or supply and demand, and the ways in which their joint behavior may be associated with risks to public goods (in particular, the notion of *social production failure*, elaborated at the end of this chapter), is a core focus of this work.

2.2 Commons-based Peer Production

Although many public goods are either naturally occurring (e.g., clean air) or provisioned by governments (e.g., the public highway system), other public goods emerge through the efforts of self-directed communities composed of individuals with diverse motives and resources. These individuals participate in a form of collective action called commons-based peer production (Benkler, 2006), in which individuals make modular contributions of varying sizes that can then be combined together in a relatively low-cost fashion. The end result of this distributed community-based effort is a body of public goods upon which our society relies, often in the form of information public goods in general or more specifically communication public goods. One advantage of commons-based peer production as Benkler (2006) pointed out is the efficiency gained by supporting self-direction: each contributor is an expert in their own skills and interests, able to match their capabilities, values, and creativity to a problem of their choosing. Modularity reduces coordination costs and allows for flexible levels of commitment. The massively distributed character of production decisions leads to a decentralized power structure that in turn creates space for experimentation.

The efficiencies and advantages of commons-based peer production are accompanied by

certain inefficiencies and disadvantages. The interests of self-directed producers may not align with what work is most needed by the public. Decentralization makes it difficult for producers, much less users, to know when a misalignment of effort is occurring, or even to keep track of the quality and importance of goods. Production communities may grow organically around a small group of high-volume contributors, then struggle to adapt their structure as needs and conditions change. Original assumptions about the design and traits of the good may not age well or prove unsustainable. Nonetheless, the development of commons-based peer production by these communities has over the last three decades produced profound innovations, often in the form of information and communication public goods.

2.3 Communication Public Goods

Public goods are defined as non-rivalrous and non-excludable; common examples include security, education, public health, the environment, and infrastructure. Fulk et al. (1996) expanded the concept of public goods to include a characterization of communication public goods, such as communication networks and knowledge bases, pointing out their *connective* and *communal* properties. When describing connective communication public goods, Fulk et al. (1996) observe that some goods may be highly oriented to making connections in a relatively straightforward and material sense—a phone network allows the flow of communication among people, and is made more valuable with each additional connection.

However, digital infrastructure is not only connective: it supports and has emerged through processes with significant communal properties, requiring collaboration, organizing, and coordination. Understanding these communal properties is key to understanding the risks these communities—and the public goods they produce—face. The Internet allows a vast range of people and communities to not only connect with one another but to collaborate in transformative ways (Shirky, 2010).

Scholars have examined the communal properties of communication public goods in a range of settings, including how development communities and organizations form and function, their structure, and their coordination practices. These sites of inquiry range from the

peer production communities producing free/libre open source software (e.g., Berdou, 2011; Bird et al., 2008; Crowston et al., 2007; Crowston, 2005; Eghbal, 2020), to a wide variety of knowledge bases—encyclopedia-building efforts like Wikipedia (e.g., Anthony et al., 2009), open mapping projects like Open Street Map (e.g., Budhathoki & Haythornthwaite, 2013; Thebault-Spieker et al., 2018) as well as in communities of practice (e.g., Ardichvili, 2008), online innovation communities (Safadi et al., 2021), and workplace knowledge bases (e.g., Arazy & Gellatly, 2013; Holtzblatt et al., 2010). The value of a peer-produced communication public good emerges through group-level processes of exchange and collaboration, with indirect reciprocity among participants (Fulk et al., 1996; Monge et al., 1998).

2.4 Social Production Failures

I argue that some of the inefficiencies and disadvantages of commons-based peer production as a mode of the development of communication public goods can be considered a form of *social production failure*—and that this failure is associated with traits in the community producing the goods. Commons-based peer production is a particularly vital form of social production (Benkler, 2006) because of the way the products of this process have become embedded in the daily life and functioning of modern society. Social production failures are a concept I develop in this dissertation as an extension and novel application of the concepts of market failures and social failures. A market failure occurs when a mutually beneficial exchange does not occur (Bator, 1958), while a social failure occurs when a mutually beneficial social interaction does not occur (Piskorski, 2014). By extension, a social production failure can occur if some mutually beneficial social production activity fails or is not sustained. These social production activities are not constrained to digital infrastructure—social production includes such phenomena as carpooling, potlucks, barn-raising, loaning a tool to a neighbor, communities of practice, crowd-sourcing, and citizen science, as well as commons-based peer production of public goods (Benkler, 2002, 2006; Shirky, 2010). Social production failures may have profound consequences—not only in the sense of missed opportunities for society, but also in the event that a relied-upon good fails. The social production process which

generated these heavily relied-upon goods in the first place may fail to adequately maintain them despite their importance.

One example of a social production failure is the vulnerability event called "Heartbleed". Heartbleed[1] was a vulnerability in the OpenSSL package. OpenSSL is an implementation of the Secure Sockets Layer (SSL), and is widely used to encrypt web traffic; the presence of an 's' in a url, e.g., the difference between http and https, indicates the presence of the security provided by SSL. As the vulnerability unfolded, it came to light that many development best practices were not being followed and that code had not been modernized using common tools (Wheeler, 2014a). The development organization had received only $2000 in donations that year; only one developer was working on the project full time with help from three part-time volunteers, despite its extraordinary importance to global communications and commerce (Eghbal, 2016; Perlroth, 2014; Walden, 2020).

Heartbleed was a social production failure in that the varying interests and motivations of the wide range of individuals, communities, and organizations who engage in commons-based peer production or who rely on digital infrastructure proved insufficient to prevent the problem by maintaining the quality of the package. Further, the relatively low level of dedicated effort likely hampered the speed and effectiveness of the response.

In the wake of this crisis, a massive influx of labor from the community led to substantial code quality improvements throughout the project (Walden, 2020), but systems using the package were updated only slowly; audits conducted more than 6 years later found hundreds of thousands of systems were still unpatched (Hope, 2020; KernelCare, 2020). One lesson of Heartbleed is that key components of our global communication infrastructure are at risk because we do not adequately maintain them. At issue, then, is how we might detect, prevent, and mitigate social production failures like Heartbleed without losing the tremendous flourishing of creative and innovation capacity that the invention of commons-based peer production has unlocked. I will return to these concepts of market failure, social failure and

[1] https://heartbleed.com/

social production failure, including propositions and areas for future work, in Chapter 6.

2.5 Underproduction

My work is focused on a single form of social production failure: *underproduction*. Underproduction was originally elaborated in economics as a form of market failure—that is, a case where a mutually beneficial exchange does not occur—and refers to the case when demand far outstrips supply. In a market, demand and supply are aligned through price. Underproduction emerges when the demand for a good is much higher than the supply; in this case, economics predicts that price would rise to reestablish equilibrium, perhaps motivating an increase in supply or a decrease in demand.

How does this basic set of axioms adapt to the case of communication public goods, which like other goods also have production functions, and as I would add to the characterization in Marwell and Oliver (1993), consumption functions? The digital nature of these goods makes the supply of new copies essentially infinite; additional units can be produced at essentially zero cost. With respect to supply, I argue that measures of *quality* are the best analog rather than, say, quantity of units. Quality might be operationalized through a range of perspectives, such as security, reliability, or sustainability in the case of software goods, or accuracy, breadth, or depth in the case of a knowledge good. Here I follow and expand the work done by Warncke-Wang et al. (2015) in the context of Wikipedia: the key dimension in the provisioning of communication public goods is whether they are indeed fit for purpose and capable of meeting demands placed upon them. With respect to demand, I argue that signals of consumer interest in the case of communication public goods can be measured as *importance*, which might be operationalized by considering measures like usage, deployment context, or the nature of our dependency on them.

A purely economic analysis would point to price as serving to align supply and demand, but in the case of communication public goods, the price is "free." Therefore, I follow Warncke-Wang et al. (2015) and Gorbatai (2011a) and define underproduction in communication public goods as the condition that arises when the quality of these goods is relatively

low while their importance is relatively high; the Heartbleed example described in the previous subsection was underproduced. However, left open to further inquiry is the nature of the force or forces that may serve to align importance and quality. With no money changing hands, no direct marketplace, no shared metrics for importance and quality, and no hierarchy commanding production priorities or consumption decisions, the situation is instead complex and contingent.

2.6 The Case of Digital Infrastructure

The form of social production with which I am concerned (peer produced digital infrastructure) can be further differentiated from other forms of collective action by the nature of ongoing effort or maintenance. Some collective actions lead to a relatively finished result, after which the effort to maintain them may be relatively modest; in the terms presented in Marwell and Oliver (1993), the production function can achieve its goal. For example, collective action to pass a law supporting public health standards may include extensive negotiations and advocacy work, but once the law is passed, it may become part of standard operating procedures for government entities, allowing activists to generally turn their attention elsewhere or disband their organization.

Other forms of collective action result in a public good that needs some level of ongoing monitoring and maintenance, with an understanding that it has some lifespan and will need a thorough overhaul or replacement in the future. Capital projects often take this form. Bridging a river may be a massive undertaking, but once built, bridge-related effort diminishes to ongoing upkeep and repairs–and, if the government is forward-thinking, a plan is made for eventual replacement some decades in the future.

However, standing in contrast to a law or a bridge, a third type of good is that for which the effort to maintain it may far exceed the effort to create it, and for which a comprehensive tear down-and-replace plan would be quite complex if it could be written at all. Although my examples in this dissertation are oriented toward the technological, other examples here might look more like a system, such as a trade network, transportation system, or democratic

governance.

I propose that digital infrastructure—in particular the software on which daily life, trade, and travel all depend—is an example of this third type of good. Studies of software cost have estimate maintenance costs as meeting or often far exceeding the cost of development (Davis, 2009; Koskinen et al., 2003), even for products with a lifespan limited to a few years—and not all products are short-lived: some components of digital infrastructure, such as NTP (network time protocol, responsible for synchronizing clocks across the internet) date back to the 1980s.[2] Meanwhile, the complexity of this infrastructure often continues to grow, with each innovation carrying with it the potential to make the overall environment more complex to maintain. The decentralized nature of software installation and control makes removal and replacement quite complex.

Broadly speaking, digital infrastructure poses numerous challenges to the way we build communication public goods. Given its importance to society, it is vital that we understand when and how we are failing to sustain it. Analysis of the Heartbleed vulnerability often focused on two key weaknesses as sources of underproduction: the state of the development community—in particular the lack of a robust organization dedicated to supporting and sustaining the package—and the state of the code itself. I contend that the state of the code again implicates the community which produced and sustained (or failed to sustain) it; we should attend to what Lee et al. (2006) calls the "human infrastructure" that lies behind infrastructure.

My thesis is that significant risks to our shared digital infrastructure can be identified by examining the social and technical conditions of the communities which produce it. I focus on underproduction as one such risk.

I will answer four key research questions in pursuit of this thesis:

1. How can we measure underproduction in digital infrastructure?

---

[2]http://www.ntp.org/ntpfaq/NTP-s-def-hist.htm

2. How are social and technical factors associated with underproduction?

3. How can we extend measures of underproduction longitudinally?

4. What can underproduction tell us about social production failure?

2.7 Dissertation Overview and Structure

The studies I describe in this dissertation take place in the context of development communities. In Chapters 3-4, I study the Debian Project (which functions as a curator, integrator, aggregator, and distributor of thousands of software packages). In Chapter 5, I shift attention to the 200+ communities located upstream from Debian in the software supply chain—the organizations directly producing the software Debian then distributes.

Each of the three empirical projects derive from the common theoretical position I have described in the previous section—stepping backward through the layers described above from the specific to the general, this throughline might be described as: underproduction in digital infrastructure is a case of social production failure in communication public goods, which in turn have been generated through commons-based peer production as a form of collective action.

Chapter 3 and 4 are previously published, and I have made minor revisions to formatting and language usage. Chapter 5 is under revision for publication. For each of these three chapters, I offer a prologue placing the work in conversation with this throughline. Following the three empirical pieces, I offer a concluding synthesis across the methods and empirical findings with implications for my theory of social production failure.

The resulting body of work will deliver on four contributions:

1. an ecosystem-level underproduction measure, suitable for cross-sectional analysis (*methodological*, Ch. 3)

2. an analysis of some social and technical factors associated with underproduction (*empirical*, Ch. 4)

3. a within-project underproduction measure, suitable for longitudinal analysis (*methodological*, Ch. 5)

4. specific observations of cases of underproduction as found in demonstrating the new methods developed for this dissertation (*empirical*, Ch. 3, 5)

Six appendices follow the main body of the dissertation. Appendix A contains additional methodological information related to Chapter 3. Appendix B contains case study material I used in the development of the methods I describe in Chapter 5. The final four appendices contain empirical projects which have informed the work presented in the main body of this dissertation: a series of findings on underproduction in the context of Wikipedia (Appendix C), a systematic literature review of software quality research (Appendix D), an exploratory causal analysis on the relationship between mentorship and software communities successfully evolving to achieve a more mature governance model and self-sustaining momentum (Appendix E), and an analysis of the association between underproduction and formality–both of organization and process (Appendix F).

Chapter 3

UNDERPRODUCTION: AN APPROACH FOR MEASURING RISK IN OPEN SOURCE SOFTWARE

In this chapter, I describe an effort to analyze underproduction according to a concrete set of measures, in ways that identify the key components of a social production failure, which I define as occurring if some mutually beneficial social production activity fails or is not sustained. I contend that the mutual benefit in this activity is among Debian users as well as between Debian users and Debian developers. Certainly, technology users build off of one another's actions and innovations. However, without developers, users would not receive any goods, and without users, developers have little reason to conduct the bulk of their work or to take on the additional burden involved with doing this work in public (Eghbal, 2020). The social production activity described in this chapter is the production and consumption of software packages. The failure comes in the breakdown of the relationship between production and consumption, such that for some cases, despite relatively high consumption, production (in the form of quality) is relatively low.

What I observe is that these failures occur in software packages produced inside a community that is in many senses a best case scenario: Debian is a high-functioning community with a core organization, skilled practitioners, a strong history, robust procedures, democratic governance, respect from across the industry, and a broad reach. These findings makes the understanding of how digital infrastructure comes to be underproduced all the more urgent, and underscores the necessity of "basic science" type methods like mine to support rigorous future analysis.

This chapter was published as: Kaylea Champion and Benjamin Mako Hill. (2021) "Underproduction: An approach for measuring risk in open source software" *28th IEEE*

International Conference on Software Analysis, Evolution and Reengineering (SANER). I led all parts of this project and have made minor typographical and formatting adjustments.

3.1 Introduction

In 2014, it was announced that the OpenSSL cryptography library contained a buffer over-read bug dubbed "Heartbleed" that compromised the security of a large portion of secure Internet traffic. The vulnerability resulted in the virtual theft of millions of health records, private government data, and more. OpenSSL provides the cryptographic code protecting a majority of HTTPS web connections, many VPNs, and variety of other Internet services. OpenSSL had been maintained through a "peer production" process common in Free/Libre and Open Source Software (FLOSS) where software development work is done by whomever is interested in taking on a particular task. For OpenSSL in early 2014, that had involved only four core developers, all volunteers. OpenSSL was at risk of an event like Heartbleed because it was an extraordinarily important piece of software with very little attention and labor devoted to its upkeep (Eghbal, 2016; Walden, 2020). In this paper, we describe an approach for identifying other important but poorly maintained FLOSS packages.

Over the last three decades, millions of people working in FLOSS communities have created an enormous body of software that has come to serve as digital infrastructure (Asay, 2019). FLOSS communities have produced the GNU/Linux operating system, the Apache webserver, widely used development tools, and more (Crowston et al., 2012). In an early and influential practitioner account, Raymond argued that FLOSS would reach high quality through a process he dubbed "Linus' law" and defined as "given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone" (Raymond, 1999). Benkler (2002) coined the term "peer production" to describe the method through which many small contributions from large groups of diversely motivated individuals could be integrated together into high quality information goods like software.

A growing body of research suggests reasons to be skeptical about Linus' law (Schweik

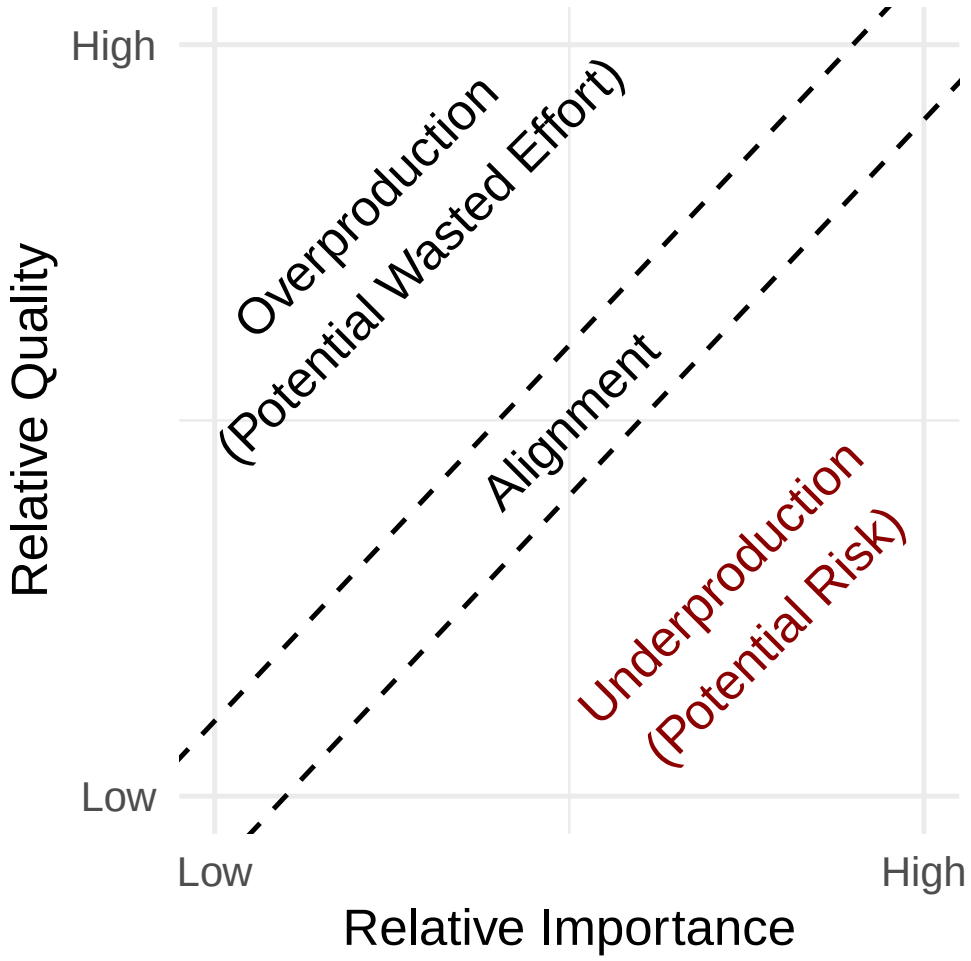


Figure 3.1: A conceptual diagram locating underproduction in relation to quality and importance.

et al., 2008) and the idea that simply opening the door to one's code will attract a crowd of contributors (Benkler et al., 2015; Schweik & English, 2012). However, while a substantial portion of labor in many important FLOSS projects is paid (Germonprez et al., 2019), most important FLOSS projects are managed through peer production and continue to rely heavily on volunteer work (Eghbal, 2020). Many FLOSS projects that incorporate paid labor have limited tools to coordinate or direct work, either paid or volunteer (Eghbal, 2020). Although some FLOSS projects are now produced entirely within firms using traditional software development models, peer production remains a critical feature of open source software production.

Over time, it has become clear that peer produced FLOSS projects' reliance on volunteer labor and self-selection into tasks has introduced types of risk that traditional software engineering processes have typically not faced. Foremost among these is what we call 'underproduction.' We use the term underproduction to refer to the fact that although a large portion of volunteer labor is dedicated to the most widely used open source projects, there are many places where the supply of quality software and volunteer labor is far out of alignment with demand. Because underproduction may go unnoticed or unaddressed until it is too late, We argue that it represents substantial risk to the stability and security of software infrastructure. As a result, the key research question of this work is: *How can we measure underproduction in FLOSS?*, which we seek to answer both conceptually and empirically.

This paper contributes to software engineering research in three distinct ways. First, we describe a broad conceptual framework to identify relative underproduction in peer produced FLOSS repositories: identifying software packages of lower relative quality than one would expect given their relative popularity. Second, we describe an application of this conceptual framework to a dataset of 21,902 source packages from the Debian GNU/Linux distribution using measures derived from multilevel Bayesian regression survival models. Finally, we present results from two experiments. The first experiment identifies a pool of relatively underproduced software in Debian. The second experiment seeks to validate this application of the framework for identifying underproduction by correlating underproduction with an

alternate indicator of risk.

The rest of the paper is structured as follows. We describe prior work on underproduction in §3.2 and present a conceptual framework in §3.3. We then describe Debian, the empirical setting for this work, in §3.4 and our approach for applying this framework to Debian in §3.5. We present the results of two experiments in §3.6. Finally, we identify significant threats to the validity of this work in §3.7 and potential implications of the work in §3.8 before concluding in §3.9.

3.2 Background

3.2.1 Detecting and Measuring Software Risk

Prevention, detection, and mitigation of risk in software development, maintenance, and deployment are the subject of substantial research interest (e.g., Gritzalis et al., 2018; Meidan et al., 2018; Natella et al., 2016). Risk detection and management techniques examine overall system safety, develop predictive models to prioritize effort (such as reliability growth models), and seek development techniques to make a project less error-prone and more fault tolerant (Bennett et al., 1996). One line of work in software quality analysis and risk detection seeks to identify issues by locating "bad smells" and anti-patterns. This includes code smells (Santos et al., 2018; Sobrinho et al., 2018), as well as architectural smells (ill-considered fundamental design decisions that may trouble the project later Le et al. (2018)) and community smells (early warning signs of problems in a community). Of particular interest to both software engineering researchers and practitioners are smells that are empirically related to failures (D. A. A. Tamburri et al., 2019).

A range of other strategies have been employed to measure risk. Code-level metrics look at complexity, change-prone code, or defect-prone code. Other approaches consider the extent that a codebase takes adequate preventative measures against risk, such as thorough testing (Wong et al., 2005). Finally, multi-factor approaches, as in decision-support analysis, take a risk management point of view and incorporate organizational factors, management

practices, and areas of potential risk throughout a project's lifecycle (Pasha et al., 2018).

3.2.2 Peer Production and FLOSS

Free/Libre and Open Source Software (FLOSS) is software released under a license that allows unrestricted use, modification, redistribution, and collaborative improvements (Crowston et al., 2004). FLOSS began with the free software movement in the 1980s and its efforts to build the GNU operating system as a replacement for commercial UNIX operating systems (Stallman, 2002). Over time, free software developers discovered that their free licenses and practices of working openly supported new forms of mass collaboration and bug fixes (Crowston et al., 2012).

'Peer production' is a term coined by Yochai Benkler to describe the model of organizing production discovered by FLOSS communities in the early 1990s that involved the mass aggregation of many small contributions from diversely motivated individuals. Benkler et al. (2015) defines peer production for online groups in terms of four criteria: (1) decentralized goal setting and execution, (2) a diverse range of participant motives, including non-financial ones, (3) non-exclusive approaches to property (e.g. copyleft or permissive licensing), and (4) governance through participation, notions of meritocracy, and charisma, rather than through property or contract. In Benkler's (2002) foundational account, archetypes of peer production include FLOSS projects like the GNU operating system or the Linux kernel as well as efforts like Wikipedia.

3.2.3 Systematic Comparison of FLOSS in Software Repositories

The process of building and maintaining software is often collaborative and social, including not only code but code comments, commit messages, pull requests, and code reviews, as well as bug reporting, issue discussing, and shared problem-solving (Robles et al., 2009). Non-code trace data may include signals of technical debt (Zampetti et al., 2020), signs that a given code commit contains bugs (Falcao et al., 2020), or serve as indicators of committed developers, a high-quality software project, or a healthy, sustainable project community

(Coelho, Valente, Silva, & Shihab, 2018; Dabbish et al., 2012; Valiev et al., 2018). Prior research has found that digital trace data capturing online community activity can provide significant insight into the study of software (Dabbish et al., 2012).

These collaborative and social systems offer a data source for understanding both developer team productivity, as in Choudhary et al.'s (2020) study of "collaboration bursts" as well as for analyzing macro-level dynamics of software production. For example, Gonzalez-Barahona et al. (2014) used the repository of *glibc* as a site to evaluate Lehman's "laws of software evolution" including the law of organizational stability which states that work rates on a system are constant. The team found that the laws are frequently not upheld in FLOSS, especially when effort from outside a core team is considered. This work suggests that the effort available to maintain a piece of FLOSS software may increase as it grows in popularity.

Prior studies have suggested that bug resolution rate is closely associated of a range of important software engineering outcomes, including codebase growth, code quality, release rate, and developer productivity (Abou Khalil et al., 2019; Kim & Whitehead, 2006; Michlmayr & Senyard, 2006). By contrast, lack of maintenance activity as reflected in a FLOSS project's bug tracking system can be considered a sign of failure (Coelho & Valente, 2017).

3.3 Conceptual Framework: Underproduction

Repositories of peer produced FLOSS are susceptible to underproduction—a concept and term that we borrow from several Wikipedia researchers who use the term to describe a dynamic that emerges when volunteers self-select into tasks. In particular, we were inspired by a study of Wikipedia by Warncke-Wang et al. (2015) who build off previous work by Gorbatai (2011b) to formalize what Warncke-Wang calls the "perfect alignment hypothesis" (PAH). The PAH proposes that the most heavily used peer produced information goods (for Warncke-Wang et al. (2015), articles in Wikipedia) will be the highest quality, that the least used will be the lowest quality, and so on. In other words, the PAH proposes that if we were to rank peer production products in terms of both quality and importance—for example, in the simple conceptual diagram shown in Figure 3.1—the two ranked lists will be perfectly

correlated. In Gorbatâi's terminology, misalignment such that quality is high but demand is low results in 'overproduction.' Peer produced goods are 'underproduced' when demand is high but quality is low.

In an economic market, supply and demand are said to be aligned through a price mechanism. Alignment is reached because lower demand decreases prices, which disincentivizes production and returns a market to equilibrium. Because there is no price signal in Wikipedia to bring consumer demands and producer supply into equilibrium, it is unsurprising that Wikipedia deviates substantially from the predictions of the PAH (Warncke-Wang et al., 2015). Indeed, "perfect alignment" serves not as a serious prediction of the relationship between Wikipedia articles' quality to the interests of the general public but as a baseline from which to identify lacunae in need of attention (Warncke-Wang et al., 2015). Research on Wikipedia has sought to characterize the negative impacts on information consumers from divergence from the PAH baseline and to identify sociological processes through which underproduction might emerge (Gorbatai, 2011b; Warncke-Wang et al., 2015).

Despite the central role that FLOSS plays in peer production, no prior efforts exist to conceptualize or measure underproduction in software. This is surprising for two reasons. First, widespread underproduction seems likely in FLOSS given that FLOSS is characterized by self-selection of software developers into tasks, varying motives among contributors, and the frequent absence of market forces for allocating producers' labor (Crowston et al., 2012; Lakhani & Wolf, 2005; O'Neil, 2009). Second, the consequences of underproduction are particularly stark in FLOSS where popular software acts as infrastructure (Eghbal, 2016, 2020). A low quality Wikipedia article on an extremely popular subject seems likely to pose much less risk to society than a bug like the Heartbleed vulnerability described earlier which could occur when FLOSS is underproduced. In this way, underproduction in software reflects an important, if underappreciated, type of risk in FLOSS.

To answer the research question (*How can we measure underproduction in FLOSS?*) in conceptual terms, our approach to detecting underproduction in software is composed of five steps as follows:

1. Assemble a collection of software artifacts that can be consistently measured as described below. These might be software packages, modules, source files, etc.

2. Identify one or more measures of quality that can be recorded consistently and independently (perhaps repeatedly) across each software artifact in the collection.

3. Similarly, identify a measure of importance that can be recorded consistently and independently across the collection.

4. Propose an *ex ante* theoretical baseline relationship between the two measures that reflects alignment. Although this might involve any number of assumptions about an ideal relationship, this might also be a non-parametric claim that the relative ranking of artifacts in terms of quality and importance will be perfectly correlated.

5. Measure deviation from this theoretical baseline across artifacts.

The measure of deviation resulting from this process serves as a measure of (mis-)alignment between quality and importance (i.e., over- or underproduction).

In a sense, this conceptual approach involves laying out software packages on dimensions similar to those shown in Figure 3.1, empirically identifying an ideal relationship that reflects general alignment (i.e., the zone in the lower left to upper right diagonal), and then measuring deviation from that ideal. This basic conceptual framework can incorporate any number of ways of measuring quality and importance—both areas of active work in software engineering research. This approach can be carried out using a range of techniques for identifying alignment including entirely non-parametric rank-based approaches, machine learning-based ordinal categorization, or parametric regression-based techniques.

3.4 Empirical Setting

The first step of applying the conceptual framework involves assembling a collection of software artifacts. We draw the collection for this study from the Debian GNU/Linux distribu-

tion which acts as the empirical setting for all of these experiments. GNU/Linux distributions are collections of software that have been integrated, configured, tested, and packaged with an installer. The contributor community producing the distribution focuses primarily on the production of packages and package management tools for managing the installation and updating of software products produced by others. Distributions like Debian play an important role in the FLOSS ecosystem and are the way that the vast majority of GNU/Linux users install operating system software as well as most applications and their dependencies. With a community in operation since 1993, Debian is widely used and is the basis for other widely-used distributions like Ubuntu. Debian had more than 1,400 different contributors in 2020[1] and contains more than 20,000 of the most important and widely used FLOSS packages.

Debian provides detailed and consistently measured longitudinal data on all its packages and maintainers in the form of released databases, datasets, and APIs (Caneill et al., 2016; Hindle et al., 2010; Nussbaum & Zacchiroli, 2010). A body of research in software engineering has used this open data from Debian to understand a range of software development practices. The Debian distribution has served as a basis for applying techniques to detect and mitigate defects (Chen & Wagner, 2007; Michlmayr & Senyard, 2006), predict bugs and vulnerabilities (Pati & Shukla, 2014), detect the evolution of package incompatibilities (Claes et al., 2015), predict component reuse (Spaeth et al., 2008), demonstrate code clone detection techniques (Cordy & Roy, 2011), develop generalizable QA techniques for complex projects (Nussbaum & Zacchiroli, 2010), investigate package dependencies (Galindo Duarte et al., 2010), and as an example of an information processing network (Villegas et al., 2020).

[1] https://contributors.debian.org/ (Archived: https://web.archive.org/web/20201107231239/https://contributors.debian.org/)

3.5 Application of Framework

3.5.1 Step 1: Assemble a Collection of Artifacts

The unit of analysis is the Debian *source package*. Source packages are the format that Debian's package maintainers modify and publish, but they are not used directly by end-users. Instead, source packages are built by computers in a Debian network of "build daemons" into one or more *binary packages* that may, or may not, include architecture-specific executables. These binary packages are then distributed to end-users and installed on their computers. Debian also provides tools to allow users to download and build their own binary packages from corresponding source packages. A single source package may produce many binary packages. For examples, although it is an outlier, the Linux kernel source package produces up to 1,246 binary packages from its single source package (most are architecture specific subcollections of kernel modules).

The one-to-many relationship between source and binary packages presented a challenge for this analysis. Although our chosen measure of quality (bug resolution, described in §3.5.2) uses information stored at the level of the source package (Davies et al., 2010), our chosen measure of importance (installations, described in §3.5.3), is aggregated at the binary level. To map source packages to binary packages, we used the Debian snapshot database's public APIs[2] to identify all binary packages produced by all versions of every source package.

3.5.2 Step 2: Identify a Measure of Quality

The second step of this framework involves identifying a measure of quality for each Debian source package. Quality in software is difficult to measure (see Appendix D for a systematic literature review of measures of quality) Common strategies for measuring quality include analyzing bug counts (e.g., Ray et al., 2014) or assessing code internal design using a series of heuristics (e.g., Santos et al., 2018). However, software engineering researchers have noted

[2]https://snapshot.debian.org/ (Archived: https://perma.cc/HQW7-R4Y2)

that the quantity of bugs reported against a particular piece of FLOSS may be more related to the number of users of a package (Davies et al., 2010; Herraiz et al., 2011), or the level of effort being expended on bug-finding (Walden, 2020) in ways that limit its ability to serve as a clear signal of software quality. In fact, Walden (2020) found that OpenSSL had a lower bug count before Heartbleed than after. Walden (2020) argued that measures of project activity and process improvements are a more useful sign of community recovery and software quality than bug count.

Additionally, techniques to assess codebases using code design heuristics are not oriented to the work of a distribution development community. The primary focus of work in a community like Debian is on configuring and testing packages for interoperability, rather than making changes to internal code design. In applying this method to Debian for this study, we followed a series of authors who have argued for a focus on a community's effectiveness at resolving the inevitable issues that arise instead of artifact-focused measures (Adewumi et al., 2016; Aksulu & Wade, 2010; Crowston et al., 2012; Eghbal, 2020; Ronchieri & Canaparo, 2018; Ruiz & Robinson, 2011). Specifically, the measure of quality is the speed at which bugs are resolved. We treat the difference between opening and closing times as the *time to resolution*. Time to resolution has been cited as an important measure of FLOSS quality by a series of software engineering scholars (Abou Khalil et al., 2019; Crowston et al., 2012; Eghbal, 2020; Kim & Whitehead, 2006). Although there may be many reasons for protracted resolution time in a distribution (e.g., maintainer skill, task complexity, report quality, lack of resources, bugs in the underlying software package causing packaging problems), the unresolved bug still represents an issue for the distribution's maintainer community and a problem for end users, whatever the reason.

Calculating this measure requires interpretation of bug reports and resolution data which Debian tracks using a database where each transaction takes the form of a specially formatted e-mail message. We extracted comprehensive data on bugs from the Debian Bug Tracking System which is also referred to as "debbugs" or the "BTS." After obtaining an archival copy of all bugs from the BTS, we queried the Ultimate Debian Database (Nussbaum &

Zacchiroli, 2010) to map bugs to packages. We parsed all actions (i.e., e-mail messages) associated with each bug into columnar data for analysis. Using BTS data, we identified the date and time when each bug was opened and closed. We treated the marking of a bug as "closed," "forwarded" (i.e., not solved by the maintainer but rather referred to the "upstream" development team for the package itself and therefore no longer Debian's responsibility), or "merged" (i.e., designated as a repeat report of an issue already in the database) as closed.

This approach differs from the approach taken by Zerouali et al. (2019b) who measure bugs as closed based on when they are last modified. We diverge from their method because final modification to a bug typically occurs when a bug is archived through an automated process that occurs about 30 days after closure. In examining the database, we found that this process can be inconsistent and that unarchiving can occur as a function of administrative processing as well as due to true reopening of a bug.

Controlling for Bug Severity

A limitation relating to this measure of quality relates to the fact that not all bugs require equal attention. In recognition of this fact, bugs in Debian are assigned a *severity* by the submitter which can be modified by the package maintainer or others. Severity in Debian is one of the following categories: wishlist, minor, normal, important, serious, grave, and critical (in that order) with normal as the default. We extracted severity for every bug from the BTS to use as a control. We treated the 4,559 bugs where the severity is "not set" and the 2,239 bugs where the severity is listed as "fixed" as priority "normal."[3] We omitted all "wishlist" severity bugs (124,961) because in addition to being used to track bugs that are "very difficult to fix due to major design considerations," this category may be used for a wide range of non-bug "to-do list" items.

Although there may be many other reasons for extended resolution time (e.g., skill, resources, inherent difficulty of the task), we do not include controls for these considerations.

[3]The "fixed" severity was used from 2000 to 2002 to indicate a non-maintainer upload and then deprecated in favor of a "tagging" approach for this type of fix.

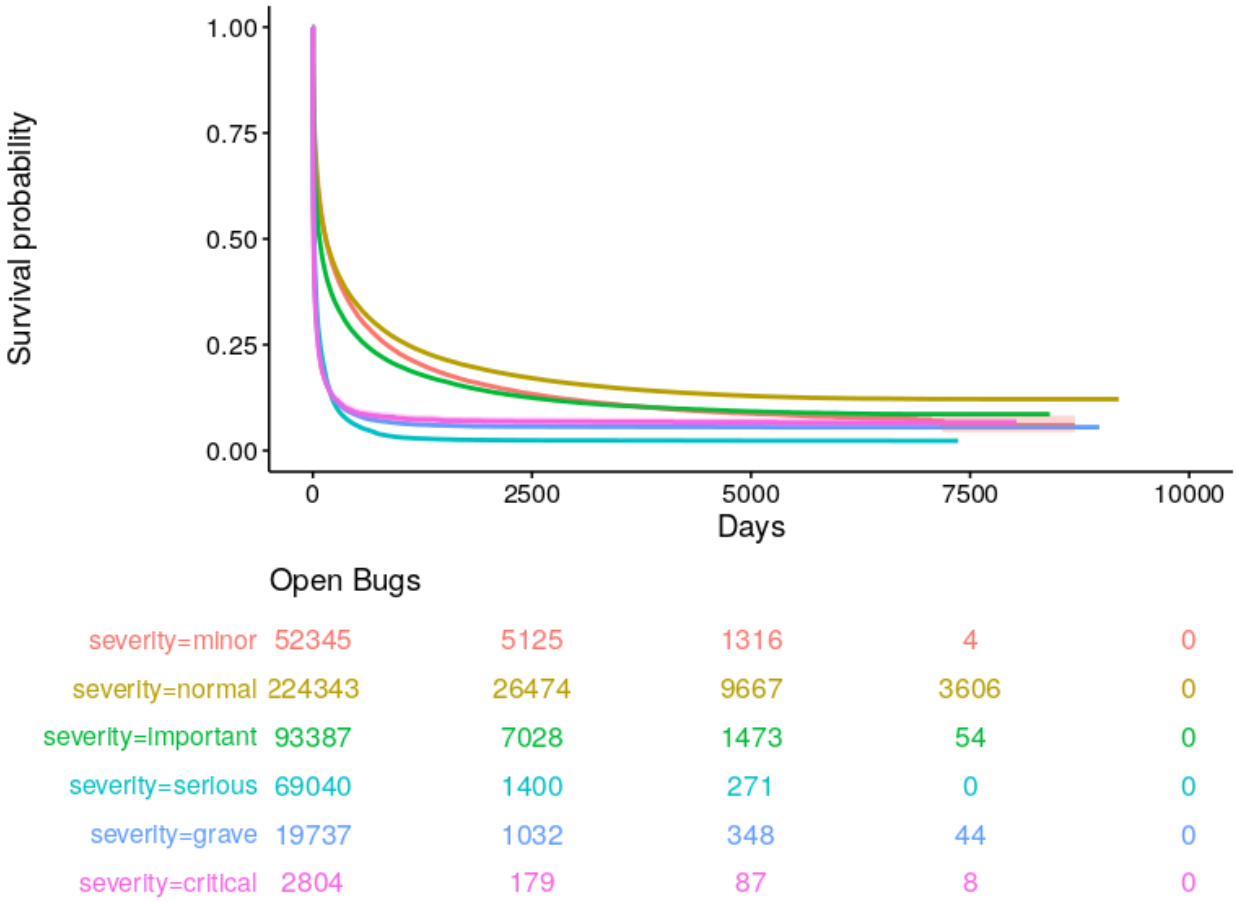


Figure 3.2: A Kaplan-Meier curve that shows the number of bugs of different severities that remain open over time.

This is because the goal of this work is to capture variation in resolution rate due to a range of reasons, which themselves may correspond to a level of risk.

Figure 3.2 shows non-parametric Kaplan-Meier survival curves for bugs of differing severities. These curves depict the probability of a given bug going from unresolved to resolved in days since it was filed. We observe in this plot that the curves are tightly clustered during the first few days after a bug is reported: about half of all bugs, regardless of severity, are solved shortly after they are reported. Bugs that are serious, grave, or critical are considered "release-critical" and must be fixed for the next "stable" release of Debian. Release-critical

bugs include security problems, bugs that make packages unusable, or licensing requirements that are incompatible with Debian.[4] We observe that the curves in Figure 3.2 cluster into two general shapes that correspond to release critical and non-release critical bugs, and that release-critical bugs are fixed more quickly.

Modeling time to resolution using Bayesian Survival Models

Measuring quality as *time to resolution* poses two analytic challenges. The first challenge is the fact that a substantial portion of bugs remain unresolved at the point of data collection. Because bugs languishing in an open state is precisely the type of risk we seek to measure, omitting bugs that are open at the time of data collection would underestimate how quickly bugs are resolved. Following recent work in software engineering by Abou Khalil et al. (2019), we incorporate data on unresolved bugs by modeling the process of bug closure directly using Cox proportional hazard models.[5]

A second challenge in measuring quality as *time to resolution* comes from the fact that the distribution of bugs across packages is highly unequal. Most of the packages in the dataset (14,604 of 21,902) have 10 or fewer bugs and more than one out of six (3,857 of 21,902) have only one bug reported. In response, we used Bayesian hierarchical models fit using Stan. The intuition behind this choice is that when a package has few bugs, the overall problem resolution rate across Debian is more informative than the package's small number of data points. In general, the Bayesian approach allows us to become progressively more confident in an estimate when more bugs have been filed against a package.

This approach finds support in the argument made by Ernst (2018) who elaborates the value of Bayesian hierarchical modeling as a method for carefully distinguishing local and global characteristics in studies of software repositories. This approach offers a "partial-

[4]https://www.debian.org/Bugs/Developer (Archived: https://perma.cc/MX3T-36PP)

[5]Cox models are a type of survival model developed originally in epidemiology to estimate how a behavior or treatment might prolong or shorten patients' lives while incorporating data from individuals for whom data is censored (i.e., individuals who are still alive at the conclusion of data collection). In this case, data is censored for any open bug.

pooling" model that allows us to both take advantage of the structure of the data (bugs are clustered within packages) and to update the prior assumption about expected resolution rates based on the new information that each bug supplies.

We incorporated a control for *severity* into a survival model of the following form where bugs are the unit of analysis and where λ reflects the hazard function capturing whether bug k in package j will be resolved at time t. We represented the severity for each bug as x_{jk} and usee z_j to reflect the log of the package-level random effect for package j. [6]

$$\lambda(t; x_{jk}; q_j) = \lambda_0(t) exp(\beta x_{jk} + q_j)$$

our measure of quality (q_j) is the package-level random effect of the posterior distributions in the survival models. We estimated this quantity using 4,000 independent draws from each package's posterior distribution using Stan. We took 95% credible intervals of these empirical distributions to reflect uncertainty. Estimates of q_j effects for all 21,902 packages are reported in the supplement Appendix A.

3.5.3 Step 3: Identify a Measure of Importance

The third step involves identifying a consistent measure of importance for every artifact in the collection. In FLOSS contexts, importance can be measured as attention on hosting sites, number of active users, intensity of use, dependency networks, or criticality of function (Eghbal, 2020; Stergiopoulos et al., 2016; Wiggins et al., 2009). We measured importance using data from from the Debian "Popularity Contest" or "Popcon" application. Popcon is an opt-in survey that shares anonymous data from volunteer systems back with Debian. Popcon data has been used in a range of previous studies in software research (Davies et al., 2010; Herraiz et al., 2011; Wiggins et al., 2009). Popcon is particularly applicable in this case because it is a signal which Debian itself has developed and deployed and because it is displayed in multiple locations in Debian's maintenance platforms. One important limitation

[6] We use notation for random effects survival models drawn from Sargent (1998).

of Popcon data is that despite the millions of systems running Debian, only a fraction have opted to report installation data.

This study includes data from 201,484 systems from a single-day snapshot on July 6, 2020. Popcon includes two measures: *inst* for *installation* (i.e. the presence of a package on a machine), which serves as the measure of importance for subsequent analysis (i_j, for each package j); and a measure called "vote" which attempts to capture if packages are being used. In this analysis, we used *installation*; Appendix A includes a version of the analysis conducted using "vote" as the measure of importance.

Unfortunately, because Popcon reports installation at the binary level, we cannot use this data to distinguish whether a single individual reported the installation of multiple binaries associated with a given source package. To avoid double-counting, we used the binary-source mappings described in §3.5.1 to set i_j to the largest install count among all binary packages associated with a source package j. As a result of this construction, the installation measure is necessarily conservative but indicates that a source package was installed at least i_j times.

3.5.4 Step 4: Select a Baseline Relationship

The fourth step in the conceptual framework involves comparing measures of quality and importance to some ideal baseline relationship. For the baseline, we take a non-parametric ranking approach that is similar to, but more granular than, the approach taken in Warncke-Wang et al.'s (2015) Wikipedia analysis which uses Wikipedia quality categories. The baseline definition of alignment is when the relative rankings of importance and quality are the same ($rq_i = rq_j$). We treated a ranking of 1 as describing the worst observed quality or lowest number of installations, while a rank of 21,902 represents the highest.

3.5.5 Step 5: Measure Deviation

The final step of the conceptual framework involves measuring deviation from the theoretical baseline. The measure of alignment is the "underproduction factor" (U_j) which we measured as the log of the ratio of rankings of importance and quality:

$$U_j = \log \frac{ri_j}{rq_j}$$

Given this construction, U_j will be zero when a package is fully aligned, negative if it is overproduced, and positive if it is underproduced. This approach means that the range of U_j is a function of repository size. In the dataset of 21,902 source packages, the range of $U_j = [-10, 10]$, where $U_j = 10$ for a theoretical maximally underproduced package where $ri_j = 21,902$ and $rq_j = 1$.

Although constructing U_j for a single value of q_j is straightforward, incorporating uncertainty in the measure of quality requires additional work. Because posterior draws in Stan are independent, we incorporate uncertainty in the measure of quality by computing U_j using the quality ranking from each of 4,000 posterior draws taken from the estimated random effect for each package j. U_j reported in the analysis reflect the 95% credible intervals from U_j computed separately for these draws. Because we only have a single measure of installation i_j, we do not attempt to incorporate uncertainty in this measure into the analysis.

3.6 Results from Experiments

In order to assess the conceptual model and to provide an empirical answer to the research question, we conduct two experiments. The first experiment describes results from the application of the method described in §3.5 and suggests that a minimum of 4,327 packages in Debian are underproduced. The second experiment validates the approach using an alternate measure of risk.

3.6.1 Experiment 1: Identifying Underproduced Software

This approach provides evidence that underproduction is widespread in Debian. Figure 3.3 shows 95% credible intervals (CIs) for all 21,902 packages. Packages whose 95% credible interval for U_j includes zero ($0 \in U_j$) are "aligned" and packages where both ends of the credible interval have the same sign as "overproduced" ($U_j < 0$) and "underproduced" ($U_j >$

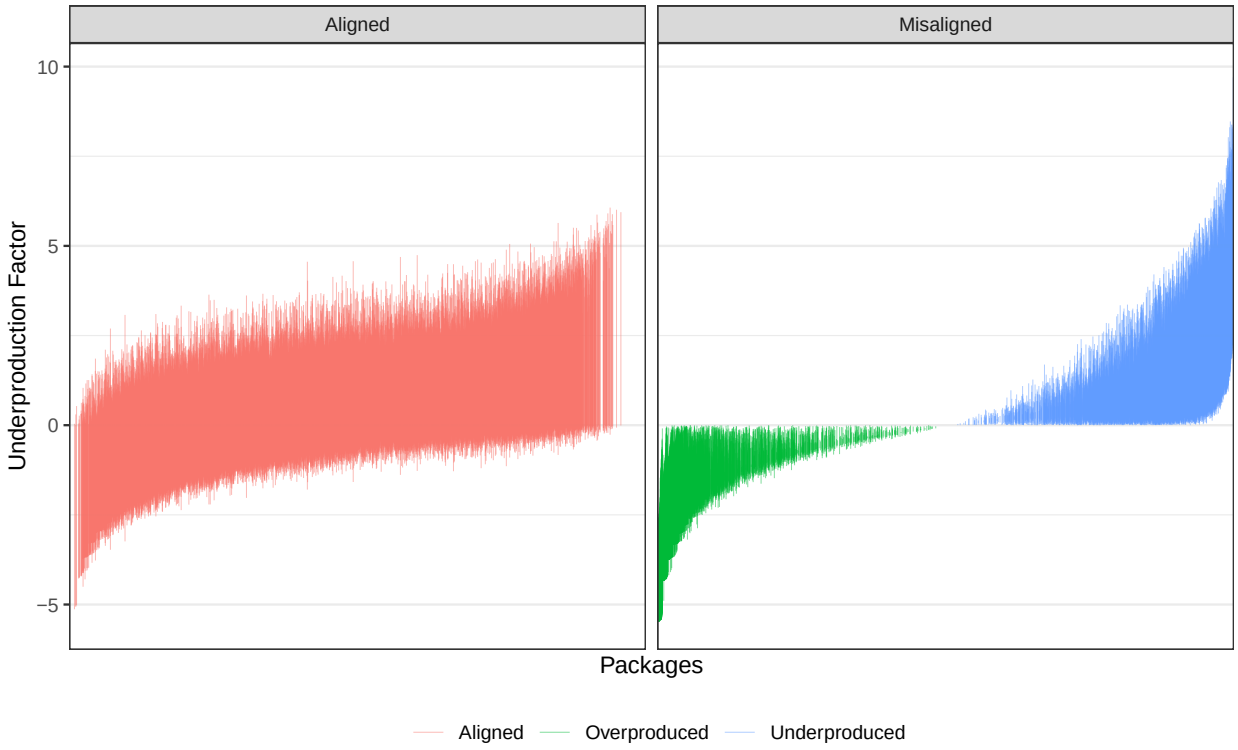


Figure 3.3: Credible intervals for U_j for every package in Debian. All packages whose CIs include zero are "aligned"; those whose CIs are entirely above 0 are labeled "underproduced;" those whose CIs are entirely below zero are labeled "overproduced."

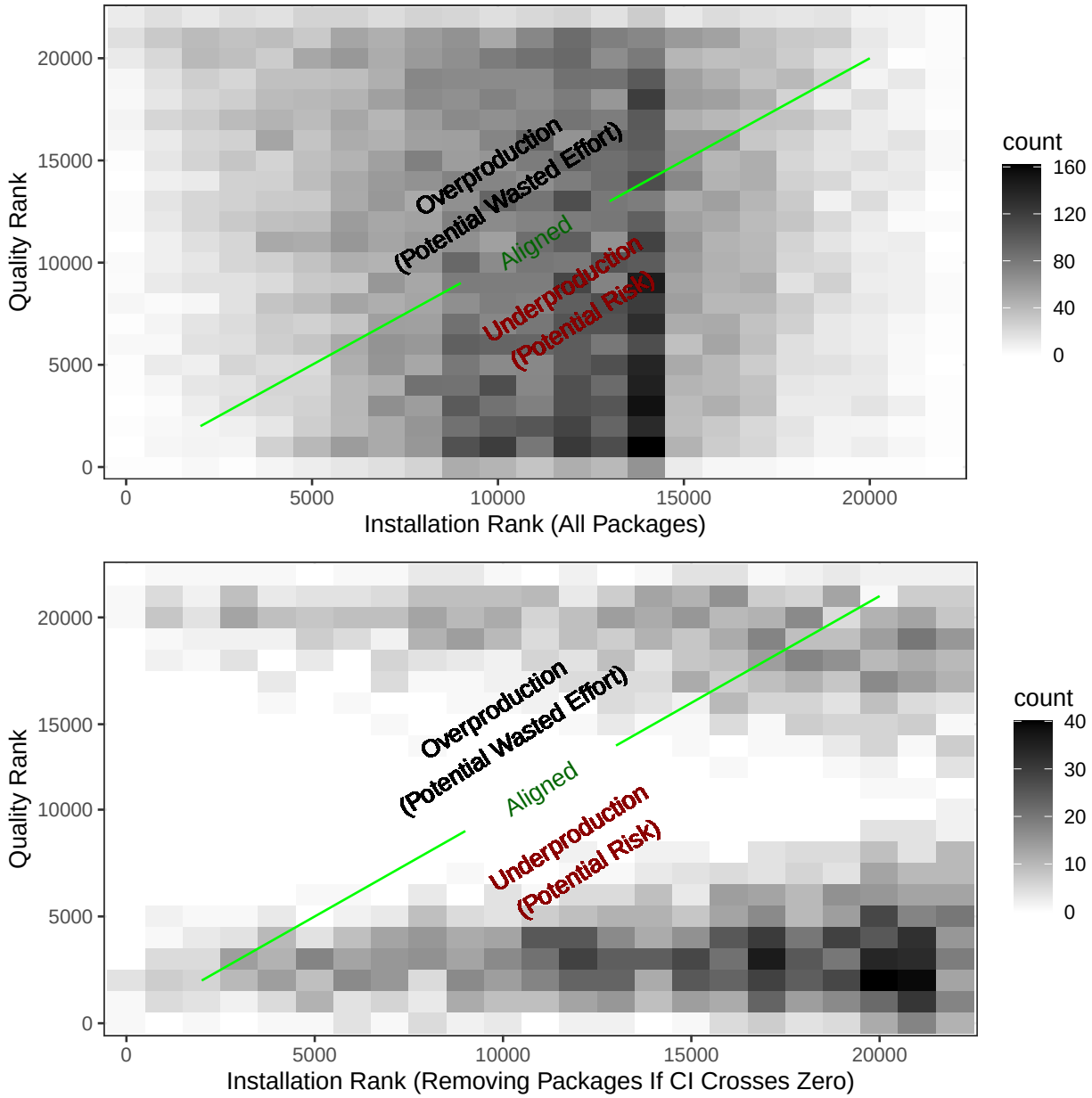


Figure 3.4: A heatmap of software alignment. Color intensity indicates the number of packages occupying a given ranking of quality and installation. Aligned packages appear along the lower-left to upper-right diagonal. The top heatmap includes all packages, while the bottom heatmap contains only those packages for which the 95% credible interval does not cross zero.

0). The wide credible intervals for many of the packages shown on the left panel of Figure 3.3 reflect the fact that many of the aligned packages may be misaligned packages with high variance. The noise in these measures of q_j and U_j is partially attributable to the fact that many packages have few bugs.

Figure 3.4 displays a heatmap visualization that shows the number of packages occupying a range of installation ranks (ri_j) along the x-axis and quality ranks (rq_j) along the y-axis in evenly spaced bins. In this way, this non-parametric data visualization seeks to reflect the basic intuition that goes into the construction of this measure of underproduction U_j. Because rq_j is a distribution, the figures display a ranking of the mean value of q_j. The top panel shows all packages in Debian and clearly shows data spread across the heatmap rather than clustered along the diagonal. The relative lack of density along diagonal in the top figure is evidence that misalignment in Debian is widespread. The relatively high density of packages in the lower right indicates that underproduction in Debian is common.

The lower panel of Figure 3.4 shows only packages for which the 95% credible interval excludes zero (i.e., the misaligned packages shown on the right panel of Figure 3.3). Removing aligned packages reveals much more density among underproduced packages relative to overproduced packages. Removing 'aligned' packages visibly hollows out the center of the heatmap filled with packages of moderate quality because many of these packages have wide CIs. However, although there are almost no packages of middling quality that we can confidently say are overproduced, there are many that we can confidently say are underproduced. Further, there is substantial density—hundreds of packages—in the extreme bottom right corner. This area contains packages that are almost maximally underproduced according to this measure. In Figure 3.5, we present a list of the packages displaying the highest levels of underproduction (mean U_j) with per-package boxplots drawn from the posterior distribution associated with each package.

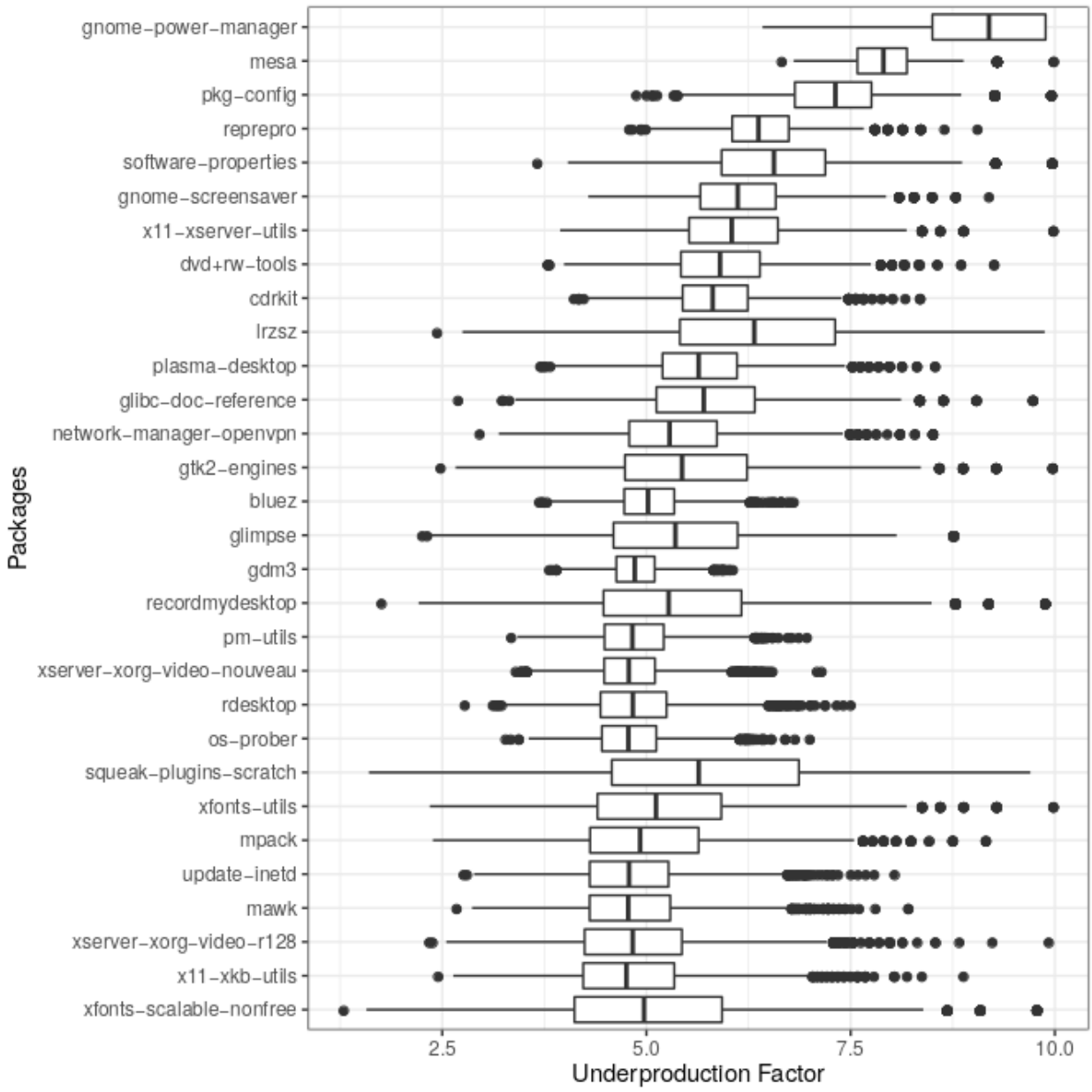


Figure 3.5: Packages displaying the highest mean levels of underproduction. Boxplots show the mean and interquartile range of the distributions of U_j and reflect uncertainty in the model of package-level quality.

Table 3.1: Predicting non-maintainer upload (NMU).

Intercept	-3.03^*
	$[-3.10; -2.95]$
Mean U_j	0.47^*
	$[0.40; 0.53]$
Num. obs.	21902

* 0 outside the confidence interval.

3.6.2 Experiment 2: Validation Using Alternate Indicator

To validate the application of this conceptual framework, we test whether the measure of underproduction can be used to predict community responses to risk. To do so, we collect data on the count of "non-maintainer uploads" (NMUs) which occur when any individual other than the denoted maintainer updates a version of a package in Debian.[7] Although some package maintainers may welcome NMUs, or even invite them, NMUs are widely understood as an indicator of risk in Debian. One of the criteria for "package salvage," that is, the taking over of maintainership without cooperation from the designated maintainer, is the presence of NMUs.[8] We hypothesize that any valid measure of underproduction in Debian will be positively associated with the number of NMUs that the package receives.

Given that the measure of NMUs is an overdispersed count, we conduct this analysis using a negative binomial regression framework. Results from the model are reported in Table 3.1. We find that U_j (underproduction factor) is a statistically significant predictor of NMU count, and that increases in underproduction factor correlate with increased numbers of NMUs. These model estimates suggest that there is a strong relationship between the measure of underproduction and NMU count ($\beta = 0.47$; CI $= [0.40, 0.53]$). To illustrate this

[7]https://wiki.debian.org/NonMaintainerUpload (Archived: https://perma.cc/TYN9-WQ79)

[8]https://wiki.debian.org/PackageSalvaging (Archived: https://perma.cc/NBF9-6QSK)

effect, consider two prototypical packages: a fully aligned package ($U_j = 0$) and one of the most underproduced packages in the distribution as depicted in Figure 3.5 ($U_j = 5$). The model predicts that, on average, a prototypical aligned package will be NMUed extremely rarely ($\widehat{\text{NMU}} = 0.048$; or less than 5% of packages will have a single NMU). On the other hand, it suggests that prototypical underproduced package will be NMUed as often as not ($\widehat{\text{NMU}} = 0.504$).

3.7 Threats to Validity

These experimental findings may be limited in their generalizability. Although Debian is widely used in software engineering research and the study of distributions has been identified as preferable to code repository hosting platforms (Spaeth et al., 2007), it is only a single empirical setting. Additionally, Debian is idiosyncratic in ways that might threaten the ability to generalize. For example, Debian's policy to only include FLOSS means we do not assess widely-installed packages with nonfree licenses or any tools that are not packaged in Debian. Additionally, although the sample captures a broad timespan of bug reports in Debian, it only reflects one snapshot in time for package installation. As a result, packages may have changed in importance over time in ways that this analysis does not capture. These experimental findings may not generalize to smaller, newer, more commercial, or more narrowly focused development communities.

With respect to internal validity, it is important to note that these results are only correlational. While prior research has implicated lack of maintenance as increasing the likelihood of failure (e.g., Coelho, Valente, Silva, & Shihab, 2018; Eghbal, 2016; Walden, 2020), we do not develop evidence to support a causal claim. We hope that further work will demonstrate whether underproduction is predictive of significant failure. The validity of this approach is also subject to threats related to construct validity. Underproduction is a concept borrowed from economics and involves a relationship between supply and demand. Although we have leveraged existing studies of underproduction in Wikipedia as part of this process of conceptualization, there remains space for further discussion about what should

constitute 'supply' and 'demand' in FLOSS. For example, we treat supply as quality but it might also be conceptualized as code quantity or developer effort. Should demand be a raw measure of consumption, as we have conceptualized it, or should we explore alternate approaches like central positions in software dependency networks?

In a related sense, this empirical work is subject to threats that stem from choices made in operationalization. For example, resolution time is an imperfect and partial measure of quality. Although we omitted bugs that package maintainers reject, certain packages might receive higher numbers of low quality or difficult-to-reproduce bug reports from less technical users, prolonging resolution time. The very high bar to filing a bug in Debian may mitigate this concern.[9] Additionally, resolution time is limited in that it emphasizes the quality-in-use perspective rather than directly taking up artifact concerns (e.g., is the code written in ways that make it efficient to maintain). Applying artifact-based metrics from software engineering to GNU/Linux distributions like Debian remains an active area of research in software engineering. We welcome future attempts to integrate alternate metrics into this framework.

The measure of installation is constrained to the systems whose administrators have volunteered data using Popcon. As with all opt-in surveys, this results in a non-random sample. Bias in this sample is possible because the installation of certain packages is possibly correlated with participation in the survey. It is also the case that importance and install base may be measured in different ways and the choice of metric may influence the results (Zerouali et al., 2019a). Although we consulted with Debian community members in designing and interpreting these experiments, there is no ground truth that we can use to ensure that we got it right.

This application of underproduction analysis is limited in that the *ex ante* baseline we selected to demonstrate the approach relies on rank ordering and can only identify *relative* underproduction within a group of software components. If this method were applied to a

[9]The onerous process of filing a bug report in Debian is described here: https://www.debian.org/Bugs/Reporting (Archived: https://perma.cc/RG3U-ZC7J).

collection of software components where all software was underproduced, it would be able to identify the worst of the batch but could not reveal a high degree of risk in general. Although this decision side-steps the need for parametric assumptions, an ordered ranking also means that we do not distinguish between consecutively ranked components.

Finally, we validated this approach using non-maintainer upload (NMU) count: the number of times a package is updated by someone other than its maintainer. However, this measure is not an entirely independent measure: packages with long resolution times and high user bases may also be more likely to draw outside contributions in the form of NMUs.

3.8 Discussion

The results suggest that underproduction is extremely widespread in Debian. The non-parametric survival analysis shown in Figure 3.2 suggests that Debian resolves most bugs quickly and that release-critical bugs in Debian are fixed much more quickly than non-release-critical bugs. The presence of substantial underproduction in widely-installed components of Debian exposes Debian's users to risk. We explore several implications of these findings in the sections below.

3.8.1 The Long Tail of GUI Underproduction

One striking feature of these results is the predominance of visual and desktop-oriented components among the most underproduced packages (see Figure 3.5). Of the 30 most underproduced packages in Debian, 12 are directly part of the XWindows, GNOME, or KDE desktop windowing systems. For example, the "worst" ranking package, GNOME Power Manager (*gnome-power-manager*) tracks power usage statistics, allows configuration of power preferences, screenlocking, screensavers, and alerts users to power events such as an unplugged AC adaptor. Seven additional packages in Figure 3.5 are also oriented to desktop uses. For example, the *pm-utils* will suspend or hibernate a computer, *network-manager-openvpn* manages network connectivity through VPNs, Ethernet, and WiFi, *mesa* is a 3D graphics library, *bluez* is part of the Linux Bluetooth stack, *cdrkit* and *dvd+rw-tools* are both

tools for creating CDs and DVDs, and *recordmydesktop* is a screen capture program. The finding of relative underproduction in these programs appears to be in line with the history of critiques of GNU/Linux with respect to desktop usability and visual tools (C. L. Paul, 2009). Although some of the reported issues in these GUI tools may be aesthetic, inspection reveals flaws reported at a range of severities including many very serious bugs.

A critique of the significance of this result might be that visual components are less likely to be used in business-critical circumstances. However, these packages have enormous install bases and are relied upon by many other packages. These results might simply reflect the difficulty of maintaining desktop-related packages. For example, maintaining *gnome-power-manager* includes complex integration work that spans from a wide range of low-level kernel features to high-level user-facing and usability issues. This pattern of underproduced GUI components suggests that although desktop GNU/Linux has made substantial progress, it remains a source of risk.

3.8.2 Implications for Software Engineering Research

Although the conceptual model and experiments demonstrate that underproduction can be measured, the detection and measurement of underproduction and the modeling of its predictors, causes, and remedies reflect a series of open challenges for the software engineering research community. More work is needed to further validate the conceptual framework and the statistical approach. We also hope that future work will extend this approach and implement this framework in other repositories of FLOSS.

3.8.3 Implications for Practice

FLOSS communities may find it useful to employ this technique to identify underproduced software in their repositories and to allocate resources and developer attention accordingly. For example, the Debian project has a volunteer QA team who might benefit from using this analysis to allocate its effort. Although underproduction is likely to be a particularly acute problem in FLOSS projects due to developer self-selection into tasks, it may also exist

in non-FLOSS contexts. We look forward to working with managers of software repositories in both FLOSS and non-FLOSS contexts to help them implement and take action based on measures of underproduction.

While FLOSS practitioners may be particularly worried about underproduction in their projects, the software infrastructure risk that results from underproduction in FLOSS is of broader concern. FLOSS acts as global digital infrastructure. Failures in that infrastructure ripple through supply chains and across sectors. Technology leaders may see opportunities to improve their own risk profiles by offering support to FLOSS components identified as relatively underproduced through the type of analysis we have described.

Finally, many studies have examined the effect of funding when firms participate in FLOSS. Mixed and often ineffective results might be improved if underproduction is taken into account when directing resources to FLOSS projects. Tradeoffs associated with the potentially demotivating effect of money may be mitigated if funds are targeted at high-impact areas with little existing volunteer labor. The influx of successful investment that followed the Heartbleed vulnerability (Walden, 2020) may offer concerned entities some hope that intervention in response to underproduction is possible and can be effective. The work seeks to help guide these types of interventions before events like Heartbleed.

3.9 Conclusion

This work makes three important contributions to software engineering research: we present a broad conceptual framework for identifying relative underproduction; we illustrate this approach using data from Debian; and we validate this method using a measure of response to risk. Results from these experiments revealed significant underproduction in the widely used Debian distribution and suggest that many of the most underproduced packages in Debian are desktop applications.

Flaws in widely used software components, regardless of their purpose, represent a source of risk to shared digital infrastructure. Even if a given bug does not result in system failure, it may provide an attack surface for intrusion or block upgrades of other vulnerable or failure-

prone components. Despite widespread dependence on FLOSS, the burden of maintenance continues to fall on small teams of volunteers selecting their own tasks. Without fresh investment of skilled and engaged participants, this public resource will remain at risk. As with Heartbleed, underproduction may not be recognized until it is too late. We hope that this work offers a step toward preventing these failures.

Online Supplement

Data and code for this paper are available in the Harvard Dataverse at the following URL: https://doi.org/10.7910/DVN/PUCD2P; the text of the online supplement is also reproduced in Appendix A.

Acknowledgement for Chapter 3

The authors gratefully acknowledge support from the Sloan Foundation through the Ford/Sloan Digital Infrastructure Initiative, Sloan Award 2018-11356. Wm Salt Hale of the Community Data Science Collective and Debian Developers Paul Wise and Don Armstrong provided valuable assistance in accessing and interpreting data. Rene Just at the University of Washington generously provided valuable insight and feedback. We are also grateful to the anonymous reviewers who pointed out opportunities for improvement. This work was conducted using the Hyak supercomputer at the University of Washington as well as research computing resources at Northwestern University.

Chapter 4

SOURCES OF UNDERPRODUCTION IN OPEN SOURCE SOFTWARE

In this chapter, I use results produced as part of demonstrating our new method in the previous chapter to test a series of predictions about how technical and social factors might be correlated with these findings. Much of this work was inspired by presenting the previous results to both academic and practitioner audiences.

I find that the passage of time is an important part of the story of underproduction, but not the entire picture—and that we should be more skeptical of the prevailing notion that underproduced packages are simply under-resourced, to be resolved simply by influxes of capital and engineering effort. That said, given the findings on the role of age in underproduction, it's clear that we need to think broadly about underproduction as a risk all projects may face in time (and not simply the province of the unlucky or ill-designed social production efforts).

The investigation of social factors also suggests the need to revisit assumptions about how communities and their structures interact with underproduced packages. In the final model we present, we find that increased resources in the form of contributors uploading new versions of a package are associated with more underproduction not less, and that higher eigenvector centrality (often interpreted as influence) is also associated with increased underproduction. The centrality measure was derived from a network where packages are connected to one another by means of their bugs having been worked on by the same people, suggesting underproduced packages are not languishing in an isolated or siloed part of the community, but rather part of the work of contributors who are central to the project. One interpretation of this is that core contributors are taking

on the problematic or struggling packages alongside other successful packages.

This chapter was published as: Champion, Kaylea and Hill, Benjamin Mako. (2024) "Sources of underproduction in open source software" *31st IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. I led all parts of this project. I have made minor typographical and formatting adjustments and removed a figure that was identical to one already included in Chapter 3.

4.1 Introduction

Open source software is frequently supported by teams of developers, system engineers, designers, and support specialists. While these teams are often organized as both firms, they increasingly take the form of networks of self-organized collaborators working together in a model called commons-based peer production (Benkler, 2002). Although the results of peer production are often innovative and influential (e.g., GNU/Linux, Apache, and Python), the tasks taken on by contributors in these efforts do not always align with tasks that are most needed by the software project's users or by the general public.

Of particular concern is software that is *underproduced*—i.e., low quality, but highly important. Underproduction has been shown to be widespread in open source software (Champion & Hill, 2021). Some underproduced software may be buried deep in the software supply chain, and vulnerabilities and flaws may not be noticed until they cause disruptions. How can we identify underproduced software and remediate risks before they cause major failures?

This paper builds on previous software engineering research focused on risk measurement to test a series of hypotheses on correlates and theorized causes of underproduction in open source software oriented to both material conditions (e.g., the programming language used) and social features (e.g., the number of maintainers). We open in §4.2 with a review of related work to build intuition around these hypotheses, describe the setting and methods in further detail in §4.3, and present our analysis in §4.4. We discuss the implications of our

results in §4.5, describing limitations around these results in §4.6 before concluding in §4.7.

4.2 Background

4.2.1 The Production of Free/Libre Open Source Software

People start and join free/libre open source software (FLOSS) projects for a wide range of reasons, often including intrinsic motivations (Eghbal, 2020; Hannebauer & Gruhn, 2016; Lakhani & Wolf, 2005). The work of these developers can be organized in numerous ways—from individual efforts with few if any other contributors, to casual handovers among whoever is willing to pitch in, to committed small-group collaborations, to networks of thousands of developers coordinating their work and making regular integrated releases (Christian & Vu, 2021; Crowston & Howison, 2006b). For example, the Apache Project today oversees a widely-used web server but was founded by a group of system administrators who had been informally trading fixes to an older, abandoned piece of software (Mockus et al., 2002). The Linux kernel was created by Linus Torvalds as a personal project (Benkler, 2002) before being shared with the world at no charge. The openness and flexibility of how work is organized do not guarantee participation. Volunteers may trickle in slowly, if at all, or be treated so poorly that they leave (Miller et al., 2022). Ultimately, software may come to rely heavily on a few individuals or even a single person. Leaders can burn out and may not have a pipeline of candidates to assume key roles (Miller et al., 2019; Tan & Zhou, 2022).

Although FLOSS's impact can be large, the process that creates these goods can be inefficient. Benkler observed that some of the largest and most well-known FLOSS projects function as commons-based peer production communities that rely on voluntary and self-organized labor (Benkler, 2006). Although a substantial portion of FLOSS is supported by firms (Germonprez et al., 2019), these firms do not generally assign tasks within FLOSS projects. Individuals—especially volunteers—tend to choose their own tasks. Unfortunately, the tasks individuals choose may not be those most needed by users. As a result, underproduction—when the quality of a good falls below its importance—can introduce an

important form of risk.

4.2.2 Alignment Between Supply and Demand in Open Source Software

In a 2021 paper that has influenced and inspired this work, Champion and Hill proposed a technique for measuring underproduction. Applied to any repository of comparable software packages, this method requires an ordinal measure of quality, an ordinal measure of importance, and a description of an optimal relationship between the two. In the example used by Champion and Hill, the proposed optimal relationship was a simple statement about relative rank: the most important packages ought to be the highest quality (illustrated in Figure 3.1). Champion and Hill illustrated their method using data extracted from Debian, finding that a "minimum of 4,327 packages in Debian are underproduced." However, a critical limitation of their work is that it did not examine potential sources, causes, or remedies for underproduction.

Where does underproduction come from within GNU/Linux distributions? Answering this question is difficult due to a lack of knowledge in the literature about the role and success of distributions in the software supply chain. However, considering that distributions's work involves high levels of expertise in writing, patching, installing, and integrating software, we can draw lessons from the software engineering literature about what factors could be associated with success and failure. Therefore, to develop hypotheses about the sources of underproduction, we consider both what is known about the most widely cited example of underproduction (the Heartbleed vulnerability in OpenSSL), as well about how underproduction might arise at the software project level, in general.

OpenSSL is open source software and the most widely used implementation of SSL.[1] The importance of OpenSSL is clear: global web traffic relies on SSL to encrypt traffic between web servers and browsers. In 2014, security researchers identified a vulnerability in OpenSSL that they nicknamed Heartbleed. At the time of its discovery, the vulnerability had been

[1] https://heartbleed.com

present for more than 2 years, and security researchers estimated that 24-55% of the most visited websites were vulnerable (Durumeric et al., 2014). Estimates of the costs to remediate Heartbleed ranged in the hundreds of millions of US dollars (Kerner, 2014). The aftermath of Heartbleed revealed that, despite its importance, OpenSSL had a range of quality challenges (Eghbal, 2016; Pagliery, 2014; Perlroth, 2014; Walden, 2020). The Heartbleed security is emblematic of underproduction because it involves extremely important software that was, at the time, low quality in critical respects.

Scholars have identified several potential reasons for Heartbleed that point to reasons for OpenSSL's underproduction. First, OpenSSL had a complex technical structure that was difficult to analyze by both tools and experts, suggesting a need for widespread refactoring and modernization (Carvalho et al., 2014; Walden, 2020; Wheeler, 2014a, 2014b). Second, OpenSSL is written in C, which, like C++ but unlike Java, lacks built-in detection of buffer over-reads—the bug that led to Heartbleed (Wheeler, 2014a, 2014b). Third, its architecture was substantially out of step with modern engineering standards (e.g., it implemented its own memory management) (Wheeler, 2014a). Bug reports related to the dangers of its memory architecture apparently were untriaged in the OpenSSL bug tracker for a substantial period prior to the release of the Heartbleed CVE.[2] Finally, there were a relatively small number of people involved with OpenSSL who were available to detect and respond to security vulnerabilities and very little funding devoted to its upkeep (Carvalho et al., 2014; Eghbal, 2016; Walden, 2020). Although scholars have presented these factors as causes or contributing factors to Heartbleed, the significance of Heartbleed is not simply in the details of how a buffer overread might arise, nor in how such an error might go undetected, but also in the broader problems of neglect and longstanding quality issues in OpenSSL that Heartbleed revealed.

Several of these factors suggest that underproduction is ultimately a technical problem to be prevented and solved. We might expect that modern languages, modern libraries, modern

[2]E.g., https://flak.tedunangst.com/post/analysis-of-openssl-freelist-reuse

architectures, and modern code analytic techniques could have detected the bug sooner or prevented the bug from being introduced in the first place. Although older packages written in older languages and according to older architectural ideas can be modernized, this is a substantial undertaking (Wheeler, 2014a, 2014b). Examining the problem of underproduction from a technical perspective suggests that some codebases are simply harder to maintain adequately and more failure prone. Technical causes of underproduction might include the language a piece of software is written in, code age, or some combination of these. Although languages and code can be refactored and modernized, the original language and architecture form at least an upper bound on how old the package can be. All things being equal, one might expect newer to be better as communities and individuals take the opportunity to learn from the past. To test these ideas, we propose three hypotheses: **(H1) underproduction is associated with older software**, and **(H2) underproduction is associated with the use of older programming languages**. Further, given improvements in engineering standards over time and our observation that OpenSSL was both an older package and written in an older language (C), we suggest **(H3) Underproduction associated with increased package age will be even stronger when the language is also old.**

Although age may be part of the explanation, some software is better maintained than other software of a similar age. There are numerous cases where old software continues to function as well or better than newer software. Focusing only on the passage of time would not account for the important role of maintainer resources. The lack of developers working on OpenSSL is frequently cited as part of the story of how Heartbleed happened (e.g., Eghbal, 2016; Pagliery, 2014; I. Paul, 2014; Walden, 2020). Proponents of open forms of collaboration point to Linus's law—the notion that "given enough eyeballs, all bugs are shallow" (Raymond, 1999). Presumably, a lack of eyeballs, therefore, leads to undiscovered vulnerabilities. In their study of the PyPi ecosystem, Valiev et al. (2018) found that a higher number of contributors was associated with both short-term and long-term project survival. A larger OpenSSL maintainer team might have prevented Heartbleed using defect detection tools or seeing the problem sooner. However, as Brooks (1995) famously argued, adding additional developer

resources can be detrimental to progress. In their study of GitHub, Joblin and Apel (2022) suggest that small clusters of collaborators with low turnover may be associated with more successful projects in comparison to projects where larger groups of people are committing to the same functions. Although we acknowledge the presence of competing explanations, our fourth hypothesis draws from the former perspective emphasizing the value of larger number of contributors and suggests that **(H4) underproduced software has fewer contributors.**

That said, attributing underproduction to simply the number of developers seems likely to be incomplete, given that FLOSS developers vary widely in such traits as knowledge of the codebase. Turnover in leadership may be harmful, as found in Joblin and Apel (2022). In his analysis of a software development firm, Mockus (2010) found that staff departures were associated with lower software quality as measured via code analysis and customer defect reports. Some open source projects rely heavily on a single individual, and the loss of the maintainer may spell the end of a project. Coelho and Valente (2017) surveyed maintainers of GitHub projects that were once popular, but have since been deprecated, and found that lack of maintainer interest and lack of maintainer time were some of the top reasons for project failure. Further, Coelho and Valente (2017) found that although maintainers tried to overcome failure by transitioning ownership to a team, recruiting a new maintainer, or bringing in new contributors, these approaches often did not revive the project.

On the other hand, some FLOSS projects are organized such that high levels of maintainer turnover are not detrimental. For example, in their study of five open source projects composed of many modules in an overarching framework (Angular.JS, Ansible, Jenkins, JQuery, and Rails), Foucault et al. (2015) found that these successful projects all had relatively high contributor turnover. Early work from Michlmayr and Hill (2003) observed that Debian tended to rely on single individuals. Robles et al. (2005) found that when a maintainer leaves the Debian project, others often adopt their packages. As a result of this process, Robles et al. (2005) found that the most important packages tend to be maintained by the most experienced maintainers. However, the adoption of a piece of software by others

does not mean that the software will ultimately be maintained at a quality level commensurate with its importance. Nassif and Robillard's (2017) study of eight open source projects (GIMP, Assimp, TrinityCore, Gitlab CE, Linux, Chromium, Kodi, and Apereo CAS) found that the departure of maintainers led to substantial knowledge loss. Further, Nassif and Robillard (2017) found that these departures led to parts of the project going unmaintained: those who continued to participate did not tend to take on maintenance of the parts written by those who had left. So, while there are reasons to think otherwise, we hypothesize that the loss of a maintainer will make it more likely that a package will be underproduced and that **(H5) underproduction is associated with maintainer turnover.**

FLOSS developers organize their work in multiple ways. They may work alone, form a loose collection of collaborators, or assemble into a team (Christian & Vu, 2021; Crowston & Howison, 2006b; Crowston et al., 2006; Robles et al., 2005). Although the team structure does not need to be large or complex to have an impact, the presence of a team suggests that a more stable collaboration has formed with a distinct identity (Studer, 2007). In their study of the PyPi ecosystem, Valiev et al. (2018) found that having a project hosted in an "organization" account, rather than an individual one, was associated with the project being 22% less likely to become dormant. Transitioning away from depending on a single maintainer was also one of the techniques Coelho and Valente (2017) found that maintainers of failing projects attempted, suggesting that maintainers themselves feel that teamwork may be helpful in preventing failure. Team approaches offer the potential for the transition of leadership, technical help, and social rewards such as encouragement and recognition. This perspective suggests that **(H6) packages maintained by a team are less likely to be underproduced than those maintained by individuals.**

Ways of organizing work go beyond the team that forms around a single package. Developers work together across multiple packages and form broad collaborative networks. Two software packages inside these networks can be thought of as connected if the same person has engaged with both. These types of connections may be valuable because someone who works on several packages may be well-placed to see how a bug in one package has implications for

another. Indeed, Joblin and Apel (2022) constructed collaboration networks from developers who commit to the same function and found that degree centrality strongly predicted success for the open source projects they examined on GitHub. One might expect that collaboration allows for greater resilience and productivity within the team supporting individual pieces of software because an individual who is highly connected to other collaborators may be able to access additional social and technical support (Bird et al., 2008). This availability of social connection may be highly valuable in driving project success. In their study of GitHub projects, Qiu et al. (2019) found that participants were more likely to persist in software projects with high potential for building social capital. A struggling package that is more central within a collaborative network may be more likely to be noticed, and the resources to solve its problems are more likely to be available. These perspectives suggest that **(H7) underproduced packages are associated with collaborators who are farther from the central influential core of collaborators.**

Finally, it may be the case that underproduced packages lack collaborators who are aware of what is happening across Debian as a whole—those people who might notice cross-package trends or have a more accurate sense of the baseline rates of problems and resolution such that they can recognize that a package is struggling. This suggests that **(H8) underproduction is associated with collaborators lacking visibility into what is going on elsewhere in the project—i.e., an absence of brokers.**

4.3 Methods

4.3.1 Empirical Setting

We build from Champion and Hill's method for detecting relative underproduction in software packages in the Debian GNU/Linux operating system distribution (Champion & Hill, 2021). Debian was founded in 1993 and has grown to serve a range of computing needs, especially with respect to server infrastructure. Debian also serves as the primary package source

for Ubuntu.[3] The Debian community has a history of democratic governance and volunteer participation in that individuals exclusively self-select into tasks and a long-established reputation for quality (Mateos-Garcia & Steinmueller, 2008; O'Neil, 2009; Sadowski et al., 2008).

An operating system distribution serves a vital integration role in the world of software development by bringing together, configuring, and testing thousands of packages under a common framework. One of the key ways that Debian members contribute to Debian is by serving as a package's *maintainer*. Although the term is used in several ways within Debian, we use the term to describe the person listed in a package's "Maintainer" field. This person (or group) will receive notifications of bugs and is responsible for the package.[4] In Debian, maintainers are not the only individuals who can update packages. Trusted community members (those granted "Debian Developer" status) can upload a Debian package in what is called a non-maintainer upload (NMU). Because they are a sign of problems, Champion and Hill used NMUs to validate their measure of underproduction. Many packages list a range of other people in package metadata who, while not the maintainer, can upload packages in what are not considered NMUs. We describe any person who uploads a version of a package outside of an NMU as an "uploader."

A Debian package maintainer is not necessarily the person who develops and maintains the software itself. Debian typically describes this latter person as the "upstream" maintainer. In this sense, Debian maintainers are the downstream recipients of problems that are located upstream in the software supply chain. However hard a Debian maintainer works, they may have little control over the challenging behavior of the software they are seeking to integrate with other packages. Of course, the responsibilities of the Debian maintainer may be made substantially more difficult if the software they are packaging is complex, has fragile dependencies, or if it not being maintained well (or at all) upstream. Although our study is of Debian, we examine only one supply chain segment. Some of the data we collect directly

[3]https://ubuntu.com/community/debian

[4]https://wiki.debian.org/Maintainers

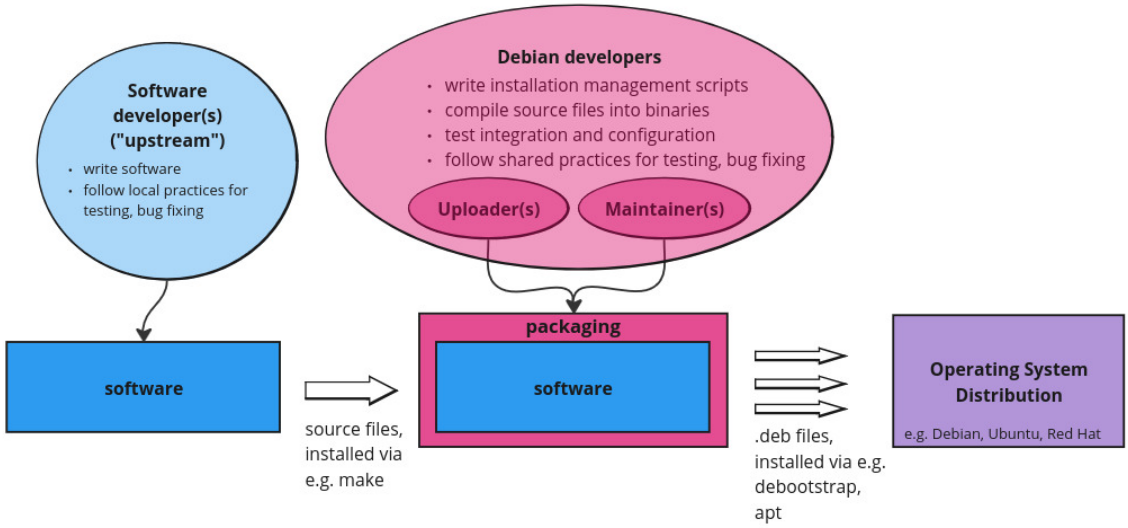


Figure 4.1: A piece of the free/libre open source software supply chain. Software is typically developed "upstream", and then numerous software programs are packaged and integrated by Debian developers before being distributed as part of an operating system or using package management tools. Users may also directly install software from source files or precompiled binaries without the benefit of a package manager (not shown).

reflects upstream conditions, which vary widely. Figure 4.1 illustrates a piece of this supply chain; upstream software developers produce software, which is packaged by operating system distribution communities like Debian.

4.3.2 Data

Debian has a history of making high-quality data available to the public in ways that have been useful for software engineering research (e.g., Caneill et al., 2016; Nussbaum & Zacchiroli, 2010). To test these hypotheses, we operationalize the concepts in each hypothesis using measures from this data (corresponding hypotheses are indicated in parentheses). We use the history of package uploads as recorded in the Ultimate Debian Database (Nussbaum & Zacchiroli, 2010) maintained by Debian, as well as package release notes for package age **(H1)**, the number of people contributing **(H4)**, the turnover in maintainership **(H5)**, and

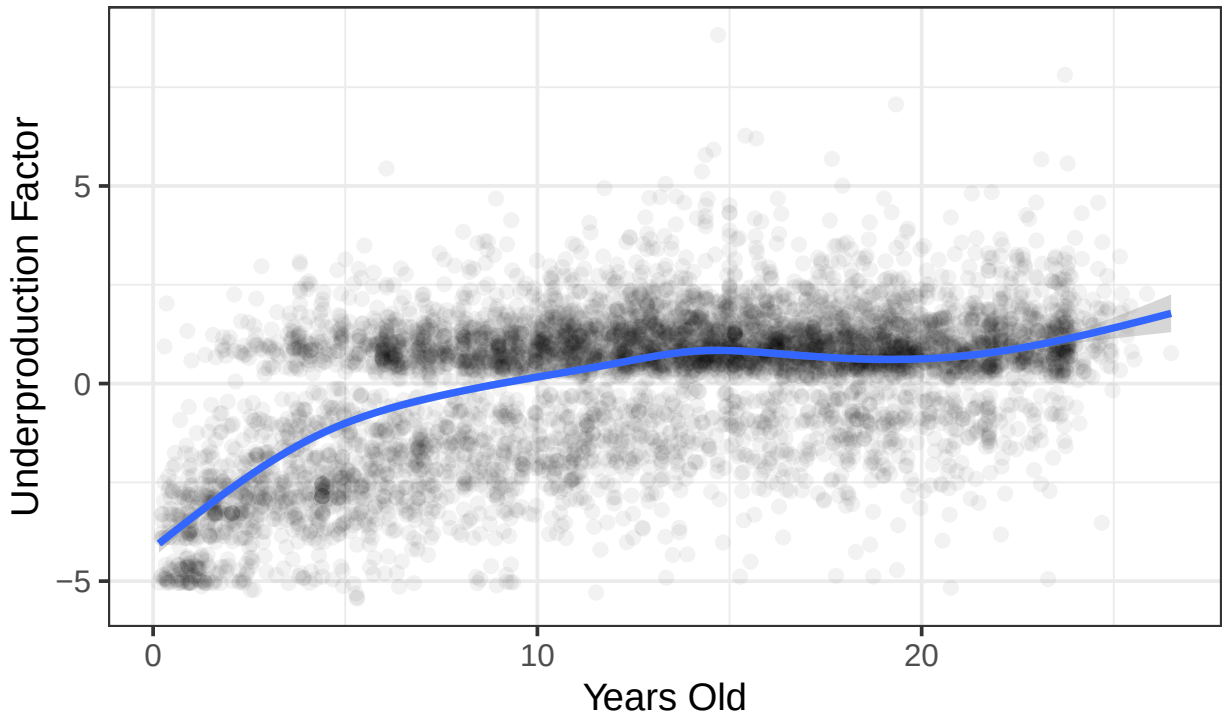


Figure 4.2: Visualizing package age based on when the package was added to Debian, with a generalized additive model (GAM) line to indicate a moving average.

the presence of a maintaining team **(H6)**. We collected package-level underproduction data from the dataset Champion and Hill published.[5] This dataset contains estimates for 6,551 packages (Champion & Hill, 2021). We merged these data with measures that we build from a series of other datasets. We identified the upstream programming language of each package (**H2**), and we used a set of tags in the UDD of the format "implemented-in::<language>". These tags are present for 2,383 of the 6,551 packages in our study and refer to 24 different languages. We also use the history of bug resolutions from the Debian Bug Tracking System (BTS) to build a collaboration network (**H7**, **H8**).

[5]https://doi.org/10.7910/DVN/PUCD2P

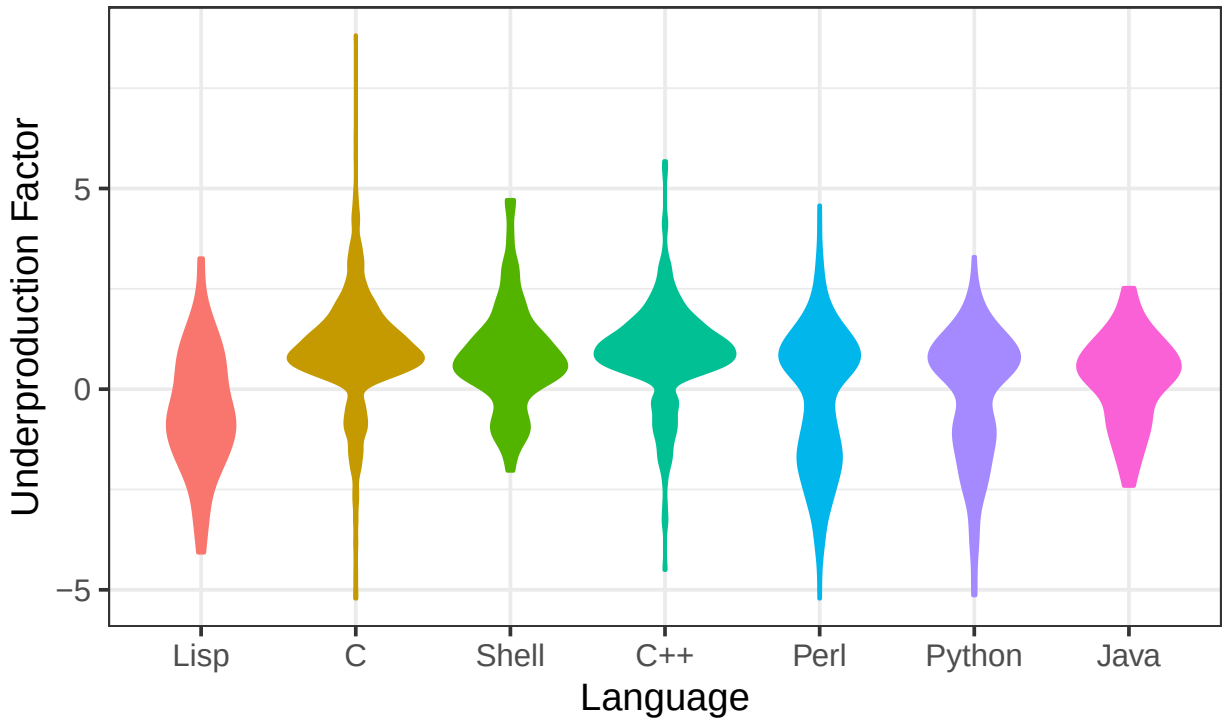


Figure 4.3: Violin plot of the data distribution broken down by the most commonly appearing languages. See Table 4.1 for models which test the relationship between language age and underproduction. This visualization contains data for 2,280 packages. On 135 occasions, the same package appears multiple times because it was consistently tagged as having been implemented in more than one language.

4.3.3 Measures

Our key outcome variable, *underproduction*, is drawn from the replication dataset published by Champion and Hill. Positive values indicate underproduction, with a higher underproduction factor associated with higher levels of underproduction; the measure is a ratio between quality rank and importance rank. We treat *underproduction* as a boolean value where true indicates that the underproduction factor, as measured in Champion and Hill, is positive.

In **H1**, we proposed that underproduction is associated with older software. We operationalize *package age* as the length of time a package has been present in Debian. To do so, we face the challenge of incomplete data because the tracking database was introduced several years into Debian's history. To prevent left-censoring, we take the earlier of two sources of information: the date that first appears in the package changelog file associated with each package (reading the last five lines in reverse to identify the latest date in the file and manually extracting the date in the 160 cases where the parser could not identify a valid date) or the oldest entry in the uploads database. We then subtract the introduction date from the data collection year (2020). As a result, older packages will have a higher *package age*.

For **H2** about the language age of packages, identifying the language in which a package is written required additional processing because packages are, in some cases, implemented in multiple programming languages and are tagged accordingly. Additionally, packages can change their programming language over time or be tagged incorrectly in ways that are later corrected. Therefore, we calculate *mean language age* per package. We constructed this measure first by measuring language proportion, taking the total proportion of releases tagged with a given language, omitting instances where a release was uploaded without any language tags. In those cases where a single release was uploaded multiple times with differences in tagging, we took the union of the set of languages listed. For example, if a package was uploaded three times, once tagged only as Perl, once tagged Perl and shell, and once tagged only shell, we treated that release as tagged Perl and shell. For a package i and

language j,

$$\text{Release Language Proportion}_{i,j} = \frac{\text{TaggedReleases}_{i,j}}{\text{TotalTaggedReleases}_i}$$

Hence our measure of language is per-package and per-language, and a continuous value from 0 to 1.

To calculate *mean language age*, we took the number of years before 2020 in which the language was introduced (the year of introduction was sourced from the English Wikipedia article about the language; larger numbers describe older programming languages) and calculate language age at the package level using the proportion of releases tagged with the language (packages can be tagged with multiple languages in a given release), multiplied by the age of the language in 2020. This measure of language age is admittedly very coarse: languages change over time, and new versions may have very different features than were available in older versions. However, we argue that language authors desire to maintain some level of continuity between versions and that the design decisions made early in a language tend to set the overall tenor for what comes later. Because there are 24 different languages represented, for a package i and language j:

$$\text{Mean Lang Age}_i = \frac{\sum_{j=1}^{24} \text{Release Lang Prop}_{i,j} * \text{YearsOld}_j}{\#\text{Total Languages Present}_i}$$

In **H4**, We proposed that underproduced packages will have fewer contributors. We measured the number of people contributing using *uploader count*, the number of people contributing to a package by counting the number of unique individuals who have uploaded new versions of the package.

In **H5**, we proposed that underproduced packages will have higher rates of maintainer turnover. We measured maintainership turnover by looking at the maintainer field of all contributions uploaded for a given package. It is important to recall that the maintainer of a package may or may not be the person doing the upload. We used a boolean value for turnover so that if a package has had more than one maintainer, *maintainer turnover* is true, otherwise false.

In **H6**, we hypothesize that packages maintained by teams will be less likely to be underproduced. Examples of teams are the Debian Games Team and the Debian Python Team. In Debian at the time of upload, the Debian archive system logs the identity of the uploader as well as the identity of the maintainer in the form of a name and email address pair. To detect whether a given upload occurred while the project was being maintained by a team, we examine the maintainer field of the log. If the maintainer is a mailing list (indicated by the mailing list email domain lists.alioth.debian.org), we record it as a team. We also identified three addresses representing the Debian-wide quality assurance teams. Because these teams are typically used a placeholder or temporary maintainers, we did not treat the presence of these QA groups in the maintainer field as indicative of being maintained by a team and omitted these uploads from our dataset.

In **H7** and **H8**, we hypothesize that underproduced packages will tend to be worked on by people in less central positions within the broader collaborative network in Debian. To build a network that considers both packages and contributors, we considered that contributors may have some social connection by having worked on bugs within the same package.

Using the package *igraph* (Csardi & Nepusz, 2006), we first generated a two-mode network from individuals doing bug work in packages. Two-mode networks are used when two types of nodes exist in a network analysis (Wasserman & Faust, 1994). In our case, one set of nodes represents individuals, and the other set represents packages. The network is formed by generating an undirected edge between people and packages when they work on bugs in the package and an undirected edge between people who have worked on bugs in the same package. This network included every bug in every package in Debian. We excluded from this network the internal-only automatic messages (e.g., "owner@bugs.debian.org") as these do not indicate actual contributors.

To obtain network measures at the level of the package, we then projected this two-mode network to a single-mode network such that packages are connected to other packages by means of the people who contribute to their bugs. Therefore, packages are ultimately connected by means of having contributors in common. This analysis yielded 78,955 con-

tributors across 18,399 packages. Not all of these packages were present in the dataset from Champion and Hill due to insufficient bug resolution data to generate an underproduction estimate. Therefore, although we use all 18,399 packages in generating our network measures, our inference is limited to the 6,551 in Champion and Hill. We use eigenvector centrality (which ranges from 0 to 1) to evaluate **H7** (proximity to the core). Eigenvector centrality is a weighted form of degree centrality: after assigning weights to nodes based on the number of connections to other nodes, the value of each connection is reweighted such that connections from high-degree nodes are worth more than those from low-degree nodes in a way that is similar to the Google PageRank algorithm. We used betweenness centrality to evaluate **H8** (cross-project visibility by means of brokerage). Betweenness centrality is the extent to which a given node lies on the shortest path between other nodes.

Name Canonification

Measuring the number of unique people contributing to a package in the form of package uploaders in **H4**, maintainer turnover in **H5**, as well as building the networks of bug collaborators in **H7** and **H8**, required identifying individuals. Doing this necessitated a process of name canonification. Bugs in Debian are represented as email messages, with addresses presented with a name part and an email part. Contributors sending messages with the system do not always use the same name and email address, and their name and address may change over time. A manual review of bug records revealed numerous examples where an individual submitted a bug with their work or personal address and then worked on the bug and resolved it using their Debian-specific address. As a result, email address was not sufficient as a primary key. Unfortunately, a person's name alone is also not a reliable primary key since some names and nicknames are relatively common globally. The risk is not only from conflating two Johns Smith, but also conflating Jack Johnson and Jennifer Juarez who have have both decided to set their name part of their email address to "JJ."

Given these concerns, we applied two heuristics as part of our canonification process. First, we reasoned that in all cases, someone using the same email address but a different

name was likely to be the same person (e.g., "J Doe <jdoe@example.com>" should be treated as the same person as "Jane Doe <jdoe@example.com>"). Further, within the same bug (but only within the same bug), we treated entries with the same name part but different email address part as also the same, e.g., "Jane Doe <jd@workaddress.com>" was treated as the same person as "Jane Doe <j.doe@homeaddress.com>". We also inspected this mapping manually to remove obvious dummy addresses and to resolve circular references. This process found 8,741 instances where the same person used more than one address for their Debian bug work and identified aliases associated with 6,438 unique individuals. We used these aliases to canonify uploaders, maintainers, and collaborators on bugs prior to analysis.

4.3.4 Analytic Plan

We used logistic regression to test each of our hypotheses, which calculates the log odds of an outcome from a vector of explanatory variables. Because the outcome variable in logistic regression is the log odds of an outcome, our models were all of the form:

$$\log\left(\frac{P_{\text{underprod}}}{1-P_{\text{underprod}}}\right) = \beta_{\mathbf{0}} + \beta\mathbf{X}$$

where **P** is the probability that a given package is underproduced, odds is defined as probability divided by 1 - probability, **X** describes a vector of variables from our dataset, and β describes the vector of fitted parameter estimates associated with our hypothesis tests.

One analytic challenge we faced was missing data in terms of package programming language (due to untagged packages) as well as missing network measures (due to isolates in the collaboration network, meaning we have no defined centrality measure for a package). In order to offer inference into the source of underproduction while managing the presence of confounders and missing data, we fit four models. M1 includes those predictors for which we have complete data (offering insight into **H1**, **H4**, **H5**, and **H6**). M2 omits only the language age predictors (**H1**, **H4-H8**). M3 omits only the network predictors (**H1-H6**). M4 is the full model (**H1-H8**) but, due to missing data, is estimated using only $\frac{1}{3}$ the observations used in M1. For each hypothesis, we assess the relationship between our predictors **X** and

underproduction and measure significance at the $\alpha = .05$ significance level. Full code and data for replicating our results are available via the Harvard Dataverse at https://doi.org/10.7910/DVN/N2HIRS.

4.3.5 Ethics

This study was conducted entirely using publicly available data published by the Debian community and does not involve any interaction or intervention with human subjects. This type of research using these data has been reviewed by the IRB at the authors' institution and has been determined not to be human subjects research. However, we recognize that this work removes observational data from its original context. Therefore, our publication does not include information that would identify Debian contributors.

4.4 Results

The results of our models are presented in Table 4.1.

4.4.1 H1, H2, and H3: Age

In **H1**, we proposed that underproduction would be associated with older software. Our results in Table 4.1 provide support for this hypothesis. This finding is consistent across all four models.

Our hypothesis in **H3** suggests the presence of an interaction between language age and package age, such that underproduction associated with older packages will be more extreme when the language it is written in is also old. However, our models contradict this hypothesis, and we find that while the main effect of being an older package and being written in an older language both tend to increase the probability that a package is underproduced, the interaction between these two independent variables has a negative coefficient. To interpret these coefficients, we visualized the marginal effect of package age for two language ages (25 years, corresponding to Java, and 48 years, corresponding to C), with results as seen in

	M1: no lang/network measures	M2: No language measures	M3: No network measures	M4: Full model
(Intercept)	−1.90*	−1.66*	−6.57*	−7.28*
	[−2.07; −1.73]	[−1.91; −1.41]	[−8.24; −4.89]	[−9.06; −5.50]
Package Age (years)	0.14*	0.08*	0.32*	0.34*
	[0.13; 0.15]	[0.06; 0.09]	[0.22; 0.43]	[0.23; 0.45]
Uploader Count	0.21*	0.13*	0.26*	0.18*
	[0.17; 0.24]	[0.09; 0.17]	[0.21; 0.32]	[0.12; 0.24]
Did maintainer change?	0.32*	0.35*	0.27*	0.22
	[0.19; 0.45]	[0.19; 0.51]	[0.03; 0.51]	[−0.03; 0.47]
Team proportion	0.17*	0.03	−0.48*	−0.20
	[0.02; 0.33]	[−0.17; 0.23]	[−0.79; −0.16]	[−0.54; 0.13]
Eigenvector Centrality		16.74*		18.91*
		[13.08; 20.41]		[14.18; 23.64]
Betweenness Centrality		−0.00		−0.00
		[−0.00; 0.00]		[−0.00; 0.00]
Mean Language Age			0.15*	0.16*
			[0.11; 0.19]	[0.12; 0.20]
Package Age : Mean Language Age			−0.01*	−0.01*
			[−0.01; −0.00]	[−0.01; −0.01]
AIC	6586.11	4373.82	2305.08	2088.71
BIC	6619.97	4418.64	2345.35	2140.37
Log Likelihood	−3288.05	−2179.91	−1145.54	−1035.35
Deviance	6576.11	4359.82	2291.08	2070.71
Num. obs.	6450	4459	2328	2299

* Null hypothesis value outside the confidence interval.

Table 4.1: These logistic regression models assess the extent to which underproduction is a function of a range of social and technical factors. Coefficients are untransformed log-odds estimates with a 95% confidence interval indicated in brackets. Note that the number of observations varies per model due to missing data.

Figure 4.4. Although the confidence intervals around the predictions are relatively wide, this result suggests that the effect on underproduction of being an older package is weaker when the language is also old.

4.4.2 H4: Contributor Count

In **H4**, we proposed that greater numbers of contributors would decrease the probability that a package is underproduced. Our results across all four models in Table 4.1 contradict this hypothesis. Instead, we find that additional uploaders are associated with an increase in odds that a package is underproduced. Our model M4 predicts that an additional uploader is associated with a 19.7% higher odds of being underproduced. In other words, increased uploaders are associated with an increased risk of underproduction.

4.4.3 H5: Maintainer Turnover

In **H5**, we proposed that maintainer turnover would be associated with a higher probability of underproduction. Although three of the four model results (M1-M3) in Table 4.1 provide support for this hypothesis, we observe that once we include collaboration network measures, language age, and the interaction of package and language age, this effect is no longer statistically significant. This suggests that having experienced maintainer turnover is not associated with higher odds of underproduction once language age and collaboration network measures are held constant.

4.4.4 H6: Organizing into Teams

In **H6**, we proposed that being maintained by a team would diminish the likelihood that a given package was underproduced. Model results presented in Table 4.1 provide little support for this claim. Instead, we found contradictory results in models M1–M3. With all covariates present in the smaller dataset in M4, we found that the impact of team proportion is not statistically significant. This suggests that, other factors being equal, the involvement of a

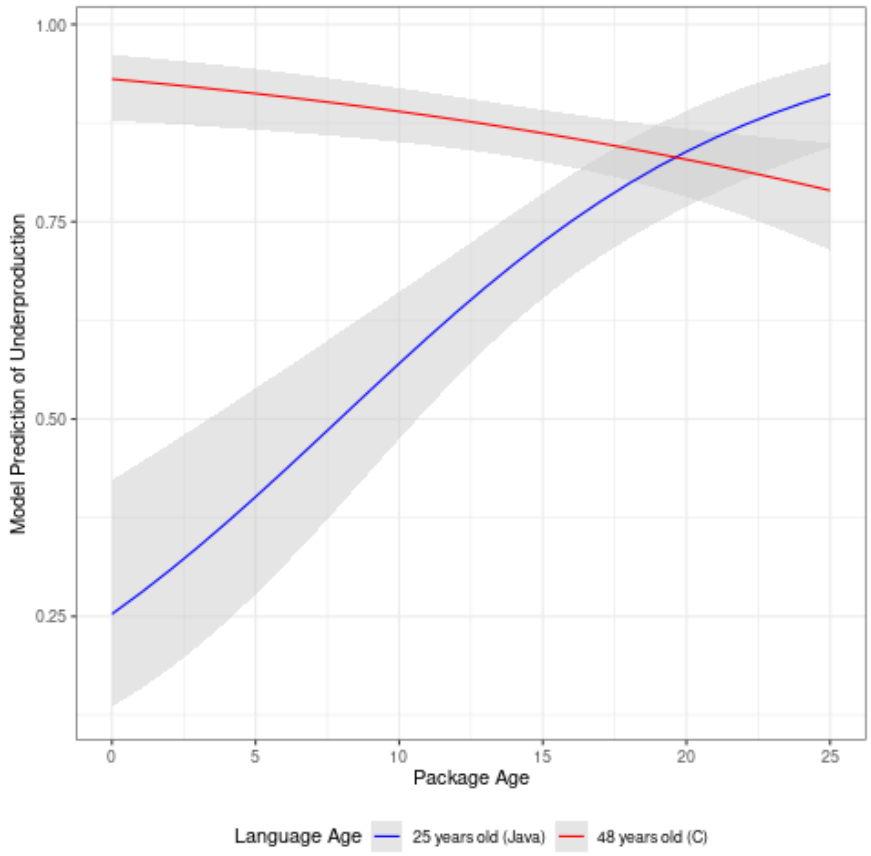


Figure 4.4: This visualization shows predicted underproduction probability from model M4 for two prototypical packages of different programming language ages where package age varies as shown along the x-axis. The package shown in blue is 25 years old, corresponding to a package written in a language as old as Java, while the package shown in red is 48 years old, corresponding to a package written in a language as old as C. The gray ribbon shows a 95% confidence interval around the prediction.

maintenance team does not impact the odds of a package becoming underproduced. Indeed, our results provide some evidence that the opposite may be true.

4.4.5 H7 and H8: Collaboration Networks

Finally, we examined the role of collaboration networks in underproduction. In **H7**, we tested whether underproduced software packages are associated with an absence of influence (eigenvector centrality). Our results for M2 and M4 presented in Table 4.1 contradicts our expectation in **H7**. Instead, we found that an increase in eigenvector centrality is associated with an increased likelihood of underproduction. In **H8**, we tested whether underproduced software packages are associated with an absence of brokerage (betweenness centrality). Our results for M2 and M4 presented in Table 4.1 contradict our expectation in **H8**. Instead, we found that all other factors being equal, the betweenness centrality of a package is not associated with an increase in the odds that the package is underproduced.

4.5 Discussion

4.5.1 The Role of Technology Choices in Underproduction

In **H1-H3**, we examined the relationship between older software, older languages, and underproduction. Our results suggest that one should think of software age in a nuanced way. All other things being equal, an older package or one written in an older language is more likely to be underproduced. However, having been written in an older language is associated with a weaker effect of package age. These results suggest that although software faces increased underproduction risk as it (and its language) grows older, thinking of software only in terms of language or age is insufficient. Indeed, for older software in older languages to be present in our dataset, it must have survived for decades. Recently written packages in newer languages are also less likely to be underproduced. Further, we observe that there is substantial variation within languages in Figure 4.3 and in the confidence interval width in Figure 4.4. Committed communities may be able to maintain the health of pieces of software

regardless of their age and language, but there are no guarantees.

From the perspective of software users, underproduction risk due to age is likewise challenging. Older software may have a range of benefits not captured in the direct maintenance of the software's quality: knowledge and trust from a history of use, availability of documentation, integration with other tools, embeddedness in a given process, or the existence of migration paths to an alternative. Change to a new service or paradigm has a cost even when the current solution is performing quite poorly. These factors may continue to elevate the importance of the software even after quality has fallen away substantially.

4.5.2 Organizing to Address or Prevent Underproduction

In **H4**, we studied the number of contributors by examining uploader quantity. In **H5**, we examined maintainer turnover. In **H6**, we examined the question of how uploaders and maintainers are organized. Contrary to our expectations, additional uploaders to a package were associated with increased odds of underproduction. Maintainer turnover, in and of itself, is not associated with underproduction. Nor is the presence of a maintenance team in the full model. These are challenging results for communities seeking to organize effort in ways that resist or prevent underproduction. It may be that identifying a level of modularity where a single uploader can sustain effort for the package's life is helpful (hence, the community should orient itself to retaining that uploader's commitment). Awareness of the key role of individual effort and the risk involved in a transition from "one" to "more than one" may allow these projects to think differently about how they approach scale and burnout prevention. Or, it may be the case that detection and remediation are more achievable than prevention.

Our results in **H7** and **H8** suggest that the people working on bugs in underproduced packages are influential in the network formed by bug commenters. This suggests that, rather than being isolated from other software, underproduced software is drawing from a highly central resource pool—one that is perhaps spread too thinly. However, this cross-sectional approach does not allow us to distinguish if the engagement with influential contributors predates the emergence of underproduction or followed after it. In sum, these findings

suggest that the best-case scenario for a piece of software is to be maintained by a dedicated individual who does not work on many other pieces of software.

4.5.3 Key Takeaways for Practitioners

Our study draws both from the distribution level and from upstream development communities, asking whether underproduction at the distribution level is attributable to technical factors such as the age of the package and the language in which the package is written or to how the distribution organizes effort. Although software developers and maintainers in distributions like Debian take on different roles in the supply chain, they have an important relationship. Software developers benefit from the additional testing, dependency management, visibility, and support that distributions provide. To the extent that their goal is to serve end users, making their package easy for distributions to maintain is in their best interest. For their part, distribution maintainers depend on upstream developers to make good quality software that can be easily installed and readily integrated with the other packages. Both groups have a role to play in preventing and addressing underproduction.

For distributions like Debian, our findings with respect to community structure should be particularly helpful. As described, the best-case scenario may be to support dedicated and focused individuals rather than push for simply "more eyeballs" or large volumes of new contributors pitching in casually. Although much of the literature on peer production communities emphasizes the power of these casual contributors as part of the long tail, distributions like Debian are an important counterexample.

For developer communities, our finding that underproduction seems to be an inevitable consequence of age and language age suggests that all projects need to confront the march of time. New projects should be careful about using older languages. However, the negative interaction term brings a sign of hope: longstanding projects may continue to pass the test of time.

4.6 Limitations

Our measure of underproduction is extracted entirely from prior results in Champion and Hill (2021). This underproduction measure used the mean resolution time of bugs as a measure of quality and usage as a measure of importance. Other measures of quality and importance could lead to different results. Furthermore, our results should be characterized within the context of how underproduction manifests itself at the distribution level. Although Debian is an important part of the software supply chain, it is only one link in the chain. Different links may be characterized by different concerns. This work may not generalize beyond the Debian context.

While this evidence suggests that packaging software is well-served by single individuals, some peer production activities are likely to be impossible without a team effort. Understanding the relative modularity of production tasks and the conditions that make it necessary to set aside the advantages of unitary leadership in favor of collaborative effort is a key area for future work.

Our measure of package age does not consider how long a package existed before it was added to Debian and is thus only a lower bound on the age of the software. Our assessment of language age only considers the year in which a given language emerged. Although this is an important part of the context of a given language and the paradigms under which it was designed, languages evolve, and code is rewritten and refreshed. Our use of language tagging in Debian omits variation in how important a given language may be to a package, and these tags are unlikely to be missing at random.

We have taken up four different perspectives on ways that people collaborate—maintainership, uploading, declaring a team, and working on bugs. However, each of these measures is relatively coarse and does not take the history of the package into account. This omits multiple forms of variation, which may be important, such as how the team functions. Further, this analysis's cross-sectional nature omits the maintenance structure's history. Teams and overworked individuals may adopt packages because they are underproduced. Or what begins as

a team effort may fall into disarray with an individual left picking up the pieces. We sought to limit the impact of confounders like these by including a full model with all predictors. However, because our collection of predictors is necessarily incomplete, our ability to infer the causes of underproduction is limited.

Additionally, we acknowledge that many of our predictors of underproduction are largely measures of quality—either of software or of software maintenance. We hope that by treating underproduction as an outcome, instead of quality directly, we are able to incorporate knowledge about importance to identify the causes and correlates of software that is less high quality *than it should be given its importance*. Doing so means that this analysis is substantively about risk, not only about identifying low-quality software.

A final limitation is the cross-sectional nature of this data and the correlational nature of the analysis. Although our hypotheses are framed in terms of causes, our results describe correlations in our data. Indeed, it is easy to imagine how underproduction could cause increases in some of our measures. For example, a package might have more maintainers over its life because it requires lots of work to fix bugs in a way that burns out maintainers or because the software is so important that it attracts lots of very capable people interested in helping. We have attempted to include a range of covariates to address this risk and have attempted to avoid causal language in our interpretation. That said, our results are best thought of as correlational evidence in support of causal theories.

Future work should seek to further unpack the factors that affect quality or importance to understand how these factors ultimately affect underproduction. Further, one should explore what social and structural factors might affect communities' underlying ability to ensure more or less alignment between quality and importance. Additional methods need to be developed to understand underproduction not only in a cross-sectional and cumulative manner, as in Champion and Hill (2021) and this work, but also longitudinally, to support prioritization and intervention.

4.7 Conclusion

Underproduction is partly a result of the simple passage of time. Older software, or software written in older programming languages, is at greater risk. This makes confronting underproduction risk a seemingly inevitable task for software in the longer term. One strategy to confront underproduction risk is to consider how best to organize maintenance effort. Although solitary contributors and teams may be viable, our results suggest that underproduction risk is associated with projects with higher resources. We found no evidence that maintainer turnover is associated with higher risk or that teams are associated with lower risk in the full model. The work of sustaining FLOSS is both an opportunity for individuals to make important personal contributions and for them to band together to build teams within larger communities. Although communities producing FLOSS distributions have little control over the age or language of the software they package, they have control over other things. Our work points to ways that FLOSS communities can use information on relative underproduction and its correlates to allocate resources and make difficult choices about when to retire or omit software when further intervention may not address the likely underlying causes of problems.

Acknowledgement for Chapter 4

The work would not have been possible without the generosity of the Debian community. We are indebted to these volunteers who, in addition to producing Free/Libre Open Source Software software, have also made their records available to the public. We also gratefully acknowledge support from the Sloan Foundation through the Ford/Sloan Digital Infrastructure Initiative, Sloan Award 2018-11356 as well as the National Science Foundation (Grant IIS-2045055). This work was conducted using the Hyak supercomputer at the University of Washington as well as research computing resources at Northwestern University.

Chapter 5

DETECTING RISK INSIDE PROJECTS USING UNDERPRODUCTION MEASURES

This chapter seeks to fill a methodological gap which was quickly apparent upon conclusion of the previous two chapters: the cross-sectional method developed in Chapter 3 offered insight across a given ecosystem and allowed for characterization of the traits associated with projects that are underproduced, as elaborated in Chapter 4. However, with an understanding that the risk of underproduction increases with time (and indeed this is a key finding in Chapter 4), I wanted a way to put time into the equation—surely these projects were not always underproduced, and surely some efforts can be made to bring them back into alignment.

Longitudinal perspectives allow for the testing of a different set of hypotheses in the future, including the introduction of causal analysis from strictly observational data (e.g., interrupted time series). I also wanted to step away from Debian slightly, and understand the traits of the goods they had to work with: after all, as a single link in the supply chain, Debian receives most of its software from elsewhere, and hence there is only so much Debian can do to combat underproduction.

Prior research into peer production of content on wikis has observed the existence of a "Rise And Decline" pattern (RAD)—projects often experience rapid initial growth followed by a decrease, with potential stability to follow (Halfaker et al., 2013; TeBlunthuis et al., 2018). I reasoned that peer production of a software project may follow a similar pattern: developers, particularly those with the freedom to choose their own tasks as they do in commons-based peer production, may initially develop something high-quality, or flock to a newly prominent package, helping to drive its rise as new features

are added quickly and bugs are resolved promptly. Eventually, this surge of energy and interest may decline as the available tasks narrow and the challenges of maintenance and technical debt emerge.

Although RAD in prior work was measured in terms of production activity levels, my sense was that we would also expect to see RAD patterns in the quality of an artifact in projects where the object was more bounded (a piece of free/libre open source software, rather than a collection of all knowledge broadly defined as in Wikipedia). We might expect a similar growth pattern for software importance: the use of a package may spread, then fall off somewhat as alternatives and gaps emerge, finally stabilizing or ultimately diminishing to the point of insignificance as it is supplanted by alternatives.

For a given software project, patterns of rise and decline may roughly proceed in parallel, with end-user interests and developer interest travelling together. However, when the RAD pattern for a package's importance diverges from the RAD pattern for its quality, such that developers are no longer maintaining packages that are still in active use, the package is underproduced and therefore a source of risk in its role as part of our shared digital infrastructure.

Just as collections of infrastructure can have underproduced members, so also a given package may also have versions that are underproduced: given a particular moment in their lifecycle, a given set of versions of a package may be poor quality relative to their importance. Early in the package's lifecycle, the package may be rarely used despite its quality; later, developer interest may wane while Internet-wide deployments continue to use the old package or lag behind in upgrading to a better one. This suggested that we might examine the synchrony of lifecycles to detect underproduction.

From the data I developed in Chapter 3 and analyzed for associated technical and social factors in Chapter 4, I selected three packages written in Python: the most underproduced package, the package with a median value for underproduction (a mildly underproduced package), and the least underproduced package. I examined the web

pages associated with each package to try to reconstruct its history, and I extracted some preliminary data about the package using the pylint static analyzer tool, used by developers to measure code quality, as well as the level of usage of the package in Debian. These case studies appear in Appendix B.

I led all parts of this project. This chapter is being revised for submission at the time of this writing: Kaylea Champion and Benjamin Mako Hill (2024). "Dynamic Underproduction Analysis: Quantifying Temporal Risk in Open Source Software."

5.1 Introduction

Decades of development effort by thousands if not millions of contributors has built the digital infrastructure on which modern computing relies: operating systems, programming languages, applications, databases, and more. However, the maintenance requirements for this digital infrastructure are vast. Understanding which software packages need additional effort and which can be deprioritized is a critical challenge facing industry, governments, and civil society groups. One strategy for tackling this challenge is underproduction analysis: assessing the extent to which software deviates from an ideal arrangement in which the quality of software is perfectly aligned with its importance.

The necessity of this work is underscored by recent efforts to understand the factors associated with underproduction. Previous work has found that underproduction risk is higher in older packages and in packages implemented in older programming languages (Champion & Hill, 2021). Just as our physical infrastructure must be maintained as it ages, so too must our digital infrastructure (Eghbal, 2016).

This chapter contributes a novel method of detecting software underproduction, which can be applied dynamically to any package without using any other packages as a baseline or point of comparison. In doing so, we answer the following research question: (RQ1) *How can we assess underproduction longitudinally, such that we can identify which packages are currently at the highest risk?* We also make an empirical contribution by applying this method

to a sample of packages written in Python. In doing so, we answer two additional research questions: (RQ2) *Which Python-language packages in Debian are currently underproduced and represent the greatest risk to Debian users?* and (RQ3) *For which packages is the degree of underproduction increasing?*

The chapter is structured as follows. We offer a review of related work in §5.2 to lay the conceptual groundwork for this new method for quantifying underproduction risk dynamically. We detail our new method in §5.3, then apply it with results as presented in §5.5. We discuss the implications of these results in §5.6, describing limitations in §5.7 before concluding in §5.8.

5.2 Background

5.2.1 Peer Produced Open Source Software

Substantial prior research has explored the phenomenon of open source software, and the commons-based peer production (CBPP) approach which has been used to build some of its most important examples. Key among these findings is the observation of self-organized developer communities with diverse motivations, a core-periphery structure, and self-selection of tasks (Benkler, 2002, 2006; Crowston et al., 2007; Crowston & Howison, 2006b; Crowston et al., 2006). Although the self-selection of tasks supports innovation and self-motivation from contributors, it also opens questions of the extent to which selected tasks align with what software users want and need most. Detecting when user behaviors diverge from developer behaviors in ways that place infrastructure at risk is the foundation of underproduction analysis of peer produced open source software. Prior work has found that underproduction is widespread in those goods developed through commons-based peer production, including knowledgebases like Wikipedia (Warncke-Wang et al., 2015) and open source software (Champion & Hill, 2021; Eghbal, 2020).

5.2.2 Underproduction Analysis

Champion and Hill (2021) introduce what they call "underproduction analysis" which they describe as based on the notion that we optimize effort against risk if we can align quality and importance. That is, in underproduction analysis, one takes up the position that given limited engineering effort, we would want the most important software to be the highest quality, are willing to tolerate low quality when importance is low, and are unconcerned if quality is high when importance is low, but we consider *low quality when importance is high* to be an important source of risk.

Despite its strengths, the method described in Champion and Hill (2021) has two substantial limitations: it is fundamentally a method for comparing a body or ecosystem of packages to each other and cannot speak to within package changes, and relatedly, it operates only in cross-sectional data. Therefore, this previous approach is difficult to use for experimental designs seeking to analyze current conditions or develop insight into causes; in turn, this limits our ability to intervene to prevent or remediate underproduction.

The key challenge to be overcome in addressing this shortcoming is the problem of selecting an appropriate baseline or critical value: in the absence of a context of comparable packages and given some level of quality in a single package, how important must that package be for us to consider it at risk? Or, given some level of importance, how much quality is too little? Our perspective is that we can use the theoretical position elaborated in Champion and Hill (2021) to resolve this question in a *within-package* way: just as we want the most important package to be the best quality, our expectation for a given package is that over its lifespan, we want it to be at its best when its importance is at its height. This preference for alignment of the two time periods–peak quality and peak importance–suggests taking a software evolution lifecycle perspective. To take a lifecycle perspective in a way that also supports quantitative techniques requires first describing software lifecycles numerically, and then identifying desirable and undesirable patterns within those lifecycles, which we now proceed to do.

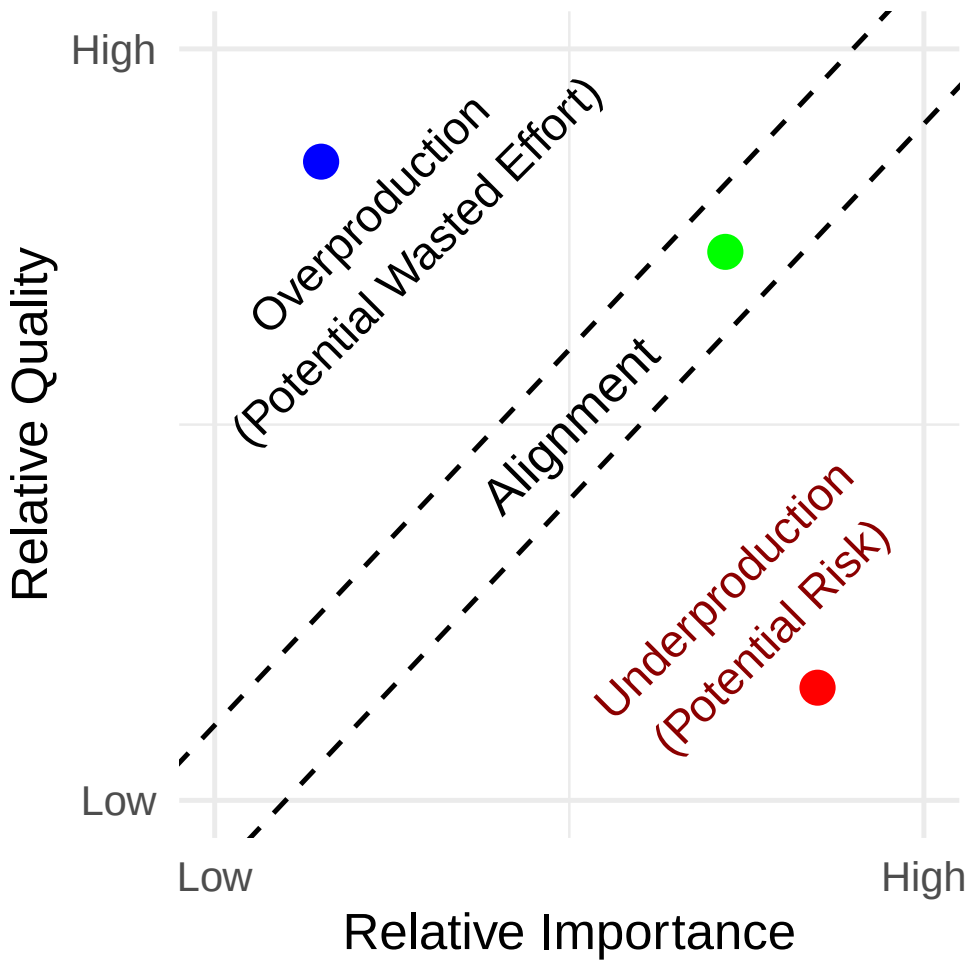


Figure 5.1: A conceptual diagram locating underproduction in open source software in relation to quality and importance, reproduced from Champion and Hill (2021) and annotated. In a cross-sectional analysis of underproduction, each dot would represent a different package–in this longitudinal approach, each dot instead represents the state of a package at a different point in time.

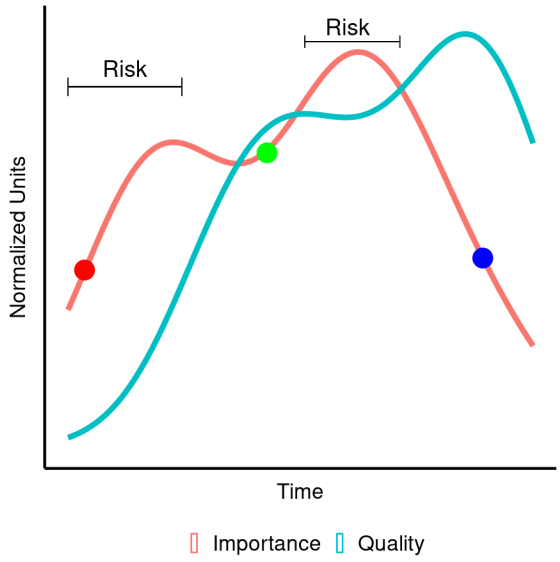


Figure 5.2: A conceptual diagram describing underproduction in open source software longitudinally for a simulated package's quality and importance trajectory. Underproduction analysis suggests that the package was at substantial risk at the period in its lifecycle where importance exceeded quality; given the drop in importance at the far ride side of the plot, current risk is low. We annotate the figure with dots corresponding to the same dots that appear in Figure 5.1.

Consider the conceptual diagram in Figure 5.1, derived from the one presented in Champion and Hill (2021). In cross-section, we might think of the red, green, and blue points as different packages. Of these, the red point represents the package at the greatest risk. The intuition for transforming this into a longitudinal view is that we might also consider these points to be the same package but at different time points. If a measurement taken today would place a package at the red dot, we would consider this a sign of higher risk. If today's measurement would place the package at the green or blue dot, we would consider this a sign of relatively low risk.

Figure 5.2 shows how this approach might appear over time. This conceptual diagram depicts the trajectory of a single package's importance and quality. We consider a package to be underproduced when quality is substantially below importance on a normalized scale

(red point); aligned when they are close together (green point), and overproduced if quality is substantially above importance (blue point). We might also weigh the slope of the line when making our assessment: given the positive slope of the quality line, we might be less concerned during the leftmost risk period, and more concerned during the rightmost risk period, particularly when quality is flat.

5.2.3 Relationship to Existing Methods

Our approach draws from two sources: first, the overall framework presented in Champion and Hill (2021), modified for the longitudinal context; and second, methodological perspectives drawing from the broader multidisciplinary literature on lifecycle analysis, particularly Raulo et al. (2023). In their method for evaluating macro-level patterns of change, Raulo et al. (2023) suggest a 6-part approach. Those parts, with the correspondence of our method (DUPA, to be presented next), are: (1) Observe the world, (2) decide units of analysis, (3) decide measures of change (DUPA1:DUPA3), (4) visualize patterns (DUPA4), (5) quantify aspects of patterns of change (DUPA5), and (6) compare patterns across different fields.

5.3 The Dynamic Underproduction Analysis Method

To answer *RQ1*, we articulate a new method: Dynamic Underproduction Analysis (DUPA). Given observations from prior work, we propose the following key steps will allow for the evaluation of dynamic underproduction, that is, identification of whether a package is in a state of underproduction at a given moment.

Step 1: Identify a longitudinal measure of importance. It must be possible to record this measure consistently over time.

Step 2: Identify a longitudinal measure of quality. It must likewise be possible to record this measure consistently over time.

Step 3: Make units comparable.

Step 4: Propose preferable temporal relationships between the two measures. These preferable relationships might be expressed in differences, magnitudes, moments, slopes, or direction. Some allowance might be made for lag if desired.

Step 5: Extract relevant comparisons of the curves defined by these measures and interpret based on the proposal in Step 4.

This method is a generalized form of the argument we have made in § 5.2 (that we optimize against risk if periods of greatest importance coincide with periods of highest quality), and is intended to support analysis of underproduction across a wide range of contexts.

5.4 Methods

5.4.1 Empirical Setting

We make our observations in the context of software packages distributed through the Debian Linux operating system (Spaeth et al., 2007), focusing on those tagged by their maintainers as having been written in Python. Debian is a popular and influential distribution,[1] and the source of packages for multiple other distributions including Ubuntu.[2] Python is one of the most widely-used development languages worldwide (Krill, 2022).

5.4.2 Data and Measures

We use data made public by Debian and the data from Champion and Hill (2024) as the basis for our analysis to build a sample of projects Debian developers have tagged as having been written in Python. For our measure of *importance*, we use results of Debian's opt-in survey tool Popularity Contest or "popcon."[3] Data from popcon is available longitudinally and available via API query.

---

[1] https://w3techs.com/

[2] https://ubuntu.com/community/governance/debian

[3] https://popcon.debian.org

The software metrics community has developed numerous measures to assess software quality. Many broad measures suffer from validation concerns and the lack of a useful baseline (e.g., How complex is 'too' complex? Is the volume of security alerts a sign of better testing and detection or a sign of low standards and flawed architecture?), or ultimately represent trade-offs among facets of quality (e.g., efficiency versus simplicity). One avenue for addressing this challenge is to make use of static analysis tools that compare code against language-specific engineering rules and practices. This approach focuses on the perspective of engineers rather than users. While it omits the highly variable contexts in which software is used, it has the advantage of holding the engineers to the same standards that they have generated and supports the collection of consistent measurements.

For our measure of quality of this sample of software packages written in Python, we use the output of running the Pylint static analysis tool.[4] Pylint identifies errors, code smells, and divergence from the PEP8 style guidelines,[5] outputting a combined quality score. Oliveira et al. (2022) found that Pylint has broad acceptance within the Python developer community as a way to detect code quality problems. We obtain every version in Debian's archive for each of the 201 packages in our dataset for a total of 4355 versions.

For our time metric *age in Debian*, we first identify a "birthday"—the first recorded date of the package being uploaded to Debian, or the first recorded usage of the package, whichever comes first. We then calculate *age in Debian* for each entry in *importance* and *quality* by taking the difference between the birthday and the date on which usage was recorded or the date on which the package was changed, respectively.

5.4.3 Analytical Plan

To answer *RQ2* and *RQ3*, we apply DUPA to a dataset step by step, structuring our presentation of the methods around the steps of a DUPA analysis.

---

[4]https://pylint.readthedocs.io

[5]https://peps.python.org/pep-0008/

Step 1: For our measure of *importance*, we use the output of the Popularity Contest ('popcon') opt-in survey tool.

Step 2: For our measure of *quality*, we use the Pylint static analysis tool.

Step 3: We used min-max normalization within each project, such that each project is only compared to its own maximum and minimum values: for project i, the quality at time t is Q_{it},

$$Q_{it} = \frac{\text{pylint score}_{it} - \min(\text{pylint score})_i}{(\max(\text{pylint score})_i - \min(\text{pylint score})_i}$$

and importance at time t is I_{it},

$$I_{it} = \frac{\text{installation count}_{it} - \min(\text{installation count})_i}{(\max(\text{installation count})_i - \min(\text{installation count})_i}$$

.

Each value Q and I is on a scale of 0 to 1, where 1 represents the very best it achieved during the observed period, and 0 the worst, each project being gauged relative only to itself. We use last observation carried forward (locf) interpolation for quality on the argument that statically measured quality is the same until a new release changes it.

Step 4: This portion of DUPA requires an essentially normative argument: we propose two conditions for an ideal relationship between quality and importance. First, to answer *RQ2*, we propose that periods of high quality and high importance should coincide—a project's best days should occur when public need is greatest. Second, to answer *RQ3*, we propose that the trajectory of each measure also matters. Specifically, while importance is growing, quality should also be growing (i.e., if the importance slope is positive, the quality slope should also be positive.) Our argument for this condition is that with increased importance comes new requirements from users. So long as importance is growing, the space of possible deployment contexts and interactions is increasing—leading to not only new ideas, but new bugs. Meanwhile, the attack surface for cybersecurity vulnerabilities in the package is likewise increasing. Suppose quality has flattened out or is beginning to decline despite continued growth in importance. In that case, we can reasonably suspect that the package

developers at that point are less able to respond to new needs represented by the increase in important.

Step 5: In this phase, we combine the prior decisions and data to generate measures of underproduction. For our answer to *RQ2*, identifying current underproduction, we use the most recently collected values of quality and importance.

We measure underproduction as being the following case:

$$I(t) > Q(t) \qquad \text{(Risk for RQ2)}$$

For our answer to *RQ3*, we need to calculate trajectories and compare slopes. When applied to real-world conditions, we anticipate that measures are noisy and individual packages may exhibit local minima/maxima. Assuming the order of the polynomial may obscure important dynamics, while examining only current values risks overfitting. Instead, we use a non-parametric approach that allows curves to be defined empirically by extracting our values of interest as predictions from a LOESS (LOcal regrESSion) function calculated from the data (R Core Team, 2024). The noise sensitivity of LOESS curves can be calibrated by setting a value of α or *span* to indicate the desired degree of smoothing. We use an α of 0.05 and round all predicted values to 4 digits on the argument that any differences of less than 0.01% are more likely due to noise imposed by the smoother rather than trends.

For the LOESS curve, such that if we take t as a particular value of time, then $Q(t)$ is some function Q predicting quality given time and $I(t)$ is some function I predicting importance given time. We take the maximum value of t from our data, and then predict values of quality and importance at that value of t.

We can extend the previous notation to express the risk condition as conditions placed on the derivatives of the function evaluated at that time t:

$$\frac{dI}{dt} > 0 \;\&\; \frac{dQ}{dt} <= 0 \qquad \text{(Risk for RQ3)}$$

5.4.4 Ethics

This study was conducted entirely using publicly available data published by open source contributors and communities, and it does not involve any interaction or intervention involving human subjects. We completed a self-certification worksheet developed by our institution's IRB and determined that this work is not human subjects research. No individually identifiable information about contributors appears in our replication materials. We have made our code and data available.[6]

5.5 Results

5.5.1 Underproduced Python Packages

To answer *RQ2*, we examined the quality and importance scores from the most recent date of analysis. We also evaluated the LOESS curves for each package at the most recent observation to evaluate *RQ3*. The results are visualized in Figure 5.3—packages in particular need of remediation because quality is trending negative while importance is trending positive are marked in blue.

In all, our analysis identified 34 packages from our set of 201 (16%) that were underproduced, as of the last observation, because their importance exceeded their quality (RQ2). The underproduced packages are listed in Table 5.1 along with our measures of quality (Q) and importance (I).

We also identified 2 packages where the most recent predicted slope for quality is negative while the predicted slope for importance is positive (RQ3). We observe that both of the packages we found with our *RQ3* criteria (i.e., the alignment of the package is getting worse) were assessed to be underproduced already. We consider these the most concerning examples of underproduction in the dataset. The detailed trajectories of these packages appear in Figure 5.4.

We observe that the two projects predicted to be at higher and increasing risk (*ufw*

[6]https://dataverse.harvard.edu/privateurl.xhtml?token=96134f4f-7443-4a9b-a9c8-728658c42c4b

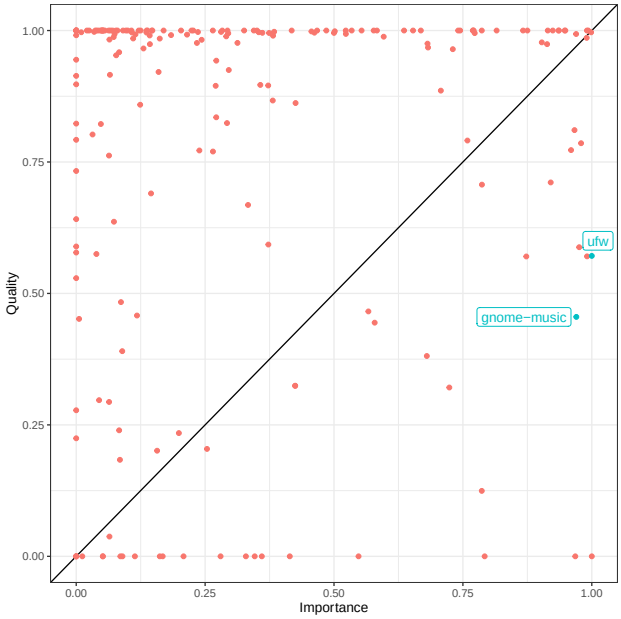


Figure 5.3: Most recent values of quality and importance for each package in our dataset. The packages in blue are particularly concerning because their LOESS-predicted quality slope is negative while the LOESS-predicted importance slope is positive.

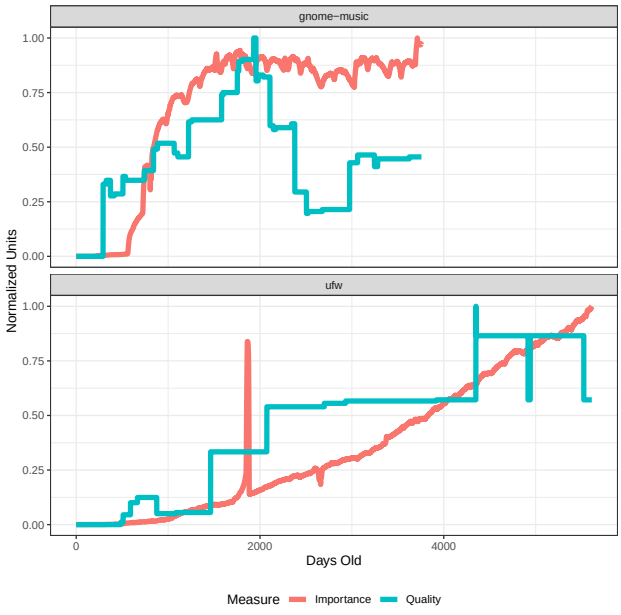


Figure 5.4: The predicted trajectories for quality and importance for our dataset's most concerning examples of underproduction.

and *gnome-music*) are quite different and may present very different challenges: *ufw* is "uncomplicated firewall," a tool for managing the Linux networking systems *iptable*,[7] while *gnome-music* is an end-user application for playing music.[8]

Package	Importance	Quality
ansible	0.9979	0.5887
apt-listchanges	1.0000	0.9861
bup	1.0000	0.5708
bzr-loom	0.0810	0.0000
calibre	0.7974	0.0000
childsplay	0.1292	0.0000
dput	0.6759	0.4509
font-manager	0.9642	0.7726
galternatives	0.9861	0.8106
gnome-music	1.0000	0.4544
klaus	0.5655	0.4444
lazygal	0.5773	0.4676
lfm	0.8442	0.1245
libuser	0.9255	0.7051
lintian-brush	0.7304	0.3230
meld	0.7957	0.7909
nicotine	0.2020	0.0000
packagekit	0.9794	0.0019
pgloader	0.9859	0.7858
picard	0.8810	0.6546
pycode-browser	0.5322	0.0000
python-axiom	0.0115	0.0000
python-crypto	0.3307	0.0000
python-django-pyscss	0.0586	0.0000
python-funcsigs	0.1677	0.0000
python-glance-store	0.0556	0.0000
python-ldap	0.4189	0.0000
python-scipy	0.1649	0.0000
pywps	0.4049	0.0627
qemu	0.4485	0.2525
swauth	0.3600	0.0000
tryton-modules-analytic-invoice	0.2608	0.2043
ufw	0.9959	0.4774
virt-manager	0.9973	0.9966

Table 5.1: Underproduced packages from our analysis.

[7]https://wiki.debian.org/UncomplicatedFirewall(ufw)

[8]https://packages.debian.org/bookworm/gnome-music

5.6 Discussion

5.6.1 Interpreting Underproduction (and Overproduction) from Multiple Perspectives

Underproduction analysis takes up an infrastructure risk perspective, with implications for development teams, open source foundations and program offices, civil society organizations, trade organizations, and regulators. However, users are also key stakeholders. Individuals, acting both privately and on behalf of their organizations as information technologists, must decide whether a given level of risk suits their purposes or the needs of their organization. Adopting the package at the timepoint represented by the red dot (underproduced) in Figure 5.1 might be less than prudent, while adopting it at the green point might appear reasonable.

However, this analysis only examines two of the three conditions. Research to date has focused on the problems of underproduction and the desirability of alignment. Less attention has been paid to overproduction—other than to characterize it as potential wasted effort. But whose effort is wasted in this case? In this case, the developer. To the extent developer effort is sufficiently fungible to be redirected elsewhere, we might intervene to nudge different choices. But what should we understand about overproduction from the user perspective?

We suggest that adopting the package as a new user at the blue point in Figure 5.1—overproduction—might also be risky if there are viable alternatives. This might seem counter-intuitive—how could software be "too good"? While there may be hidden gems among the overproduced packages, very highly overproduced packages (e.g., those in the top left corner of Figure 5.3) may also indicate a problem for some end users. We argue that the fact that excellent quality is not associated with high importance in some cases but is instead associated with the opposite, should give new adopters pause, especially for deployment contexts with a high cost of change (e.g., embedded systems)—is there some unobserved reason this package has not been more widely adopted?

It is worth entertaining the idea that the absence of a regulating force keeping importance and quality in alignment is itself reason for concern—whether the package is underproduced or overproduced. Shrinking importance despite high quality suggests something unsatisfying

about what the package delivers or absent factor in the quality analysis—perhaps due to some superior alternative, the absence of critical features, the presence of flaws that are not detectable through static analysis, or the package has some challenging dependencies. Technically correct software may still not be fit for purpose. Additional metrics should accompany underproduction analysis as a decision tool, and further elaboration on how importance and quality are regulated with respect to one another is an important area of future work.

5.6.2 Implications for Research

DUPA is anchored in traditions of both social science and computer science. Econometrics offers a rich range of computational techniques which might be applied to trends of software quality and importance. Our preliminary findings in this space only begin to sketch the range of opportunities for translating social scientific approaches and econometric measures to the cause of software lifecycle analysis. The availability of a longitudinal approach supports causal modeling techniques and additional measures. We imagine further examining characteristics like second derivatives for signs of slowing momentum and repurposing measures more commonly used to assess economic phenomena (such as market bubbles, leading indicators, trailing indicators, and so on).

Software development communities and their lifecycles likewise offer a wide range of settings for examining social scientific questions about organizing and collective behavior. One such line of investigation is the provisioning of public goods—of which open source software, being non-excludable and non-rivalrous, is one. Our approach is compatible with the "production function" notion described in Marwell and Oliver's (1993) work on the nature of critical mass and suggests further investigation into traits they describe as relevant to the achieving of critical mass, including group size, group resources, the presence of a core group, and the shape of the production function itself (that is, the relationship between marginal contributions of resources and marginal progress toward the provisioning of the public good.)

5.6.3 Implications for Practice

Taken at their narrowest, our empirical results identify several packages for which the quality and importance curves' relative values and relative slope suggest risk. Users of these packages may want to review these results to minimize their exposure, and developers may want to adjust their practices and dependencies with this result in mind.

More broadly, our results illustrate a lifecycle perspective on software quality and importance as a way to quantify risk. Although our illustration is grounded in projects written in Python, adapting our general framework to other environments may generate insights for any community examining the maintainability of our shared digital infrastructure.

5.7 Limitations

In the process of developing our method, we faced several potential complications. The first was developing a basis of comparison—identifying what circumstances we would classify as desirable versus undesirable. Ultimately, we concluded that a normative assumption was unavoidable: our own judgment was necessary in identifying a relationship between quality and importance. We describe our reasoning, but other arguments are possible.

A further limitation is imposed by the divergence in software update behaviors from the timing of software releases. Although we compare importance and quality synchronously, only some people update their software to a new version immediately. Many of the recorded users of a given package will still be using the previous version. Therefore, the quality of a package at the time of release is a measure of the quality that is available at that time, not necessarily the quality being experienced by contemporary users. This limits how we can interpret our results and argues for caution in evaluating short-term trends.

5.8 Conclusion

This chapter has proposed a method for studying software development lifecycles from a social and technical perspective, demonstrating a novel method and empirical results sug-

gesting packages at risk. Understanding where a given project is in its lifecycle relative to both quality and importance allows us to make informed decisions about where more development effort is needed and the risk we might expose ourselves to if we adopt it. We face a growing challenge: managing and sustaining digital infrastructure developed over the past two or more decades is difficult and become more so. Robust analysis of our current state, the causes of underproduction, and how best to prevent it are our best defense against the risk posed by low-quality but highly important software.

Acknowledgement for Chapter 5

The work would not have been possible without the continuing generosity of the open source community. We also gratefully acknowledge support from the Sloan Foundation through the Ford/Sloan Digital Infrastructure Initiative, Sloan Award 2018-11356 as well as the National Science Foundation (Grant IIS-2045055). This work was conducted using both the Hyak supercomputer at the University of Washington and research computing resources at Northwestern University.

Chapter 6

CONCLUDING REFLECTIONS

This dissertation has included empirical findings identifying underproduced infrastructure and organizational and technical features associated with underproduction, methods for identifying underproduction risk within an ecosystem and a longitudinal method for identifying underproduction risk within a single project. I will conclude by offering a preliminary elaboration of a theory of social production failure, and offer implications for three key constituencies: (1) researchers in communication, social computing, and computer science, (2) practitioners within commons-based peer production communities, and (3) civil society groups concerned with information public goods and digital infrastructure.

As described in the introductory Chapter 1, this dissertation dives in to the complex and contingent realm that is the maintaining and sustaining of digital infrastructure, seeking to make this kind of work, and this empirical setting, more legible to a broader audience. From the brief description I offered in Chapter 1, I hope that the self-organizing, self-directing, and ever-emergent work processes of the wide range of people who cooperate to sustain digital infrastructure are evident and visible. In Chapter 2, I offer a perhaps less melodic but more rigorous account of the key theories that constitute the framework in which I have been working: collective action and commons-based peer production to generate communication public goods—and a key challenge to those communication public goods: social production failures. I choose as my focus a particular form of social production failure: underproduction. I also describe a particularly salient setting for underproduction: digital infrastructure.

I explore answers for four research questions. The first question, *How can we measure underproduction in digital infrastructure?* I answer by describing a five-part method, and I demonstrate that method in Chapter 3, finding that underproduction is widespread in digital

infrastructure. I use the data from this analysis to answer my second research question, *How are social and technical factors associated with underproduction?* I conduct a series of hypothesis tests about some social and technical factors associated with underproduction, finding in Chapter 4 that underproduction risk increases with aging packages and aging software languages, and that contrary to my proposal, increased resources are associated with increased risk and that these packages are not maintained in peripheral parts of the ecosystem but rather occupy a central position. In Chapter 5, to answer my third research question, *How can we extend measures of underproduction longitudinally?*, I build from this work a series of case studies (see Appx. B) and propose an extended method suitable for longitudinal analysis. I also present a series of empirical results from the application of this method. In this concluding chapter, I engage in some theory-building and synthesis work to answer my fourth research question, *What can underproduction tell us about social production failure?* I hope that these findings with be useful to a range of audiences.

6.1 A Theory of Social Production Failure

> **Social production failure:** when a mutually beneficial social production activity fails to occur, launch, meet its goals, thrive, be sustained, complete, or gracefully end.

The preceding chapters have sought to make clear the extent to which underproduction represents a failure—both a failure in the sense of a market failure (because what fails is a beneficial exchange, and some regulating factor that should have—or could have—brought supply and demand into alignment has not done so), and a social failure (because what has failed is in some ways a social exchange, e.g., mutually beneficial task selection, or rounds of generalized reciprocity, in keeping with the account of diverse motivations as summarized in Chapter 2).

A social production failure as I have defined it is the union of these two definitions, and occurs when some mutually beneficial social production activity fails to occur or is not sustained. This chapter offers a brief summary of theories of market failure and social failure

as previously described in Chapter 2, now with an eye to the evidence I have developed my proposal that we should concern ourselves with social production failures as a class of collective action challenges, of which underproduction as I have described it represents only one manifestation. I then offer a series of propositions derived from theories of market failure and social failure taken in conversation with the empirical work developed so far as part of that elaboration of social production failure. I close by outlining a research agenda for investigation of social production failures, describing what I see as the most pressing questions and challenges.

6.1.1 Connections to Market Failure

Bator (1958) is credited as having developed the modern definition of market failure (Marciano & Medema, 2015), which Bator describes as "the failure of a more or less idealized system of price-market institutions to sustain 'desirable' activities or to estop 'undesirable' activities." Bator (1958) considers these activities to be both production and consumption, and characterizes desirability in terms of a maximal utility function given some set of preferences. Market failures can occur in the case of public goods, potential transactions where parties act with imperfect information, and exchanges with substantial externalities, and intervention (as by governments) may be necessary to prevent underproduction (Thomas, 2015a, 2015b). In her history of theories of market failure and public goods, Johnson (2015) describes general agreement that market failure in the case of public goods has distinct qualities, and in particular that, as defined by "the market failure arose from the combination of the very features that defined a public good — nonrivalry and nonexcludability—rather than from the mispricing that characterizes externalities" (p. 17). In the assessments of market failure and appropriate response, there has emerged two distinct lines of argument about how best to respond to these market failures, with one line emphasizing the importance of government intervention (with prototypical arguments being made by, e.g., Paul Samuelson) and the other emphasizing other forms of institutional arrangements as offering solutions (e.g., as argued by Ronald Coase, James Buchanan, and Elinor Ostrom) (Furton & Martin,

2019; Johnson, 2015).

This line of argument is consistent with what I have described in the case of digital infrastructure software produced through commons-based peer production. Digital infrastructure software is a public good in that it is non-excludable and non-rivalrous, and it is underproduced as I have shown in Ch. 3-5: not always produced to the quality that is demanded. As a public good, digital infrastructure software is also subject to externalities and imperfect information. Externalities—in which uninvolved people are beneficiaries or victims of an exchange by others—include all the people and firms who benefit from digital infrastructure without contributing to its upkeep. Imperfect information—a lack of information about supply and demand—is also an ongoing challenge because it is often impossible to directly measure the size of the demand for a given product, since it is distributed and redistributed without limits and the marginal cost of production is essentially zero. As described in the limitations of my empirical project work in Ch. 3-5, robust usage metrics are scant, and a single download transaction might represent anywhere from zero to vast numbers of uses.

6.1.2 Connections to Social Failure

Piskorski (2014) describes social failure in terms closely analogous to the definition of a market failure as described in economics. A social failure is when a mutually beneficial social exchange does not occur. This perspective treats social failure as a shared problem, explicitly rooted in mutuality and exchange. Piskorski describes these failures as arising because of interaction costs making the exchange difficult or risky: perhaps it is complex, costly, or requires overcoming stigma or breaking social norms. He describes technology as reducing these interaction costs, e.g., dating sites make it faster to search for potential partners without risking a face to face rejection, professional networking sites allow people to keep their colleagues and friends-of-friends informed of their qualifications without the overt embarrassment that could result if they were known to be searching for a job while still employed, and social networking sites can make it easier to casually keep up with other people's personal news without having to ask naive or intrusive questions. In each case,

technology enables social exchange (like connection-forming and searching) to occur and allows for mutually beneficial exchange with lower social cost and risk. In the case of digital infrastructure, we might think of the mutually beneficial exchange as being between software producers and software consumers, but the medium of exchange is diffuse and complex, in keeping with the complex motives of producers and the limited signals available from consumers as described in Ch. 2. That said, many of the social rewards for producers (e.g., sense of satisfaction at meeting needs successfully, prestige from working on something important) are contingent on the ongoing existence of consumers, or on the persistence of other producers (e.g., experience of collaboration, opportunity to learn).

Piskorski is not the only social scientist to find the term "social failure" appealing. Millar (2020) offers the "social failure in technology" as occurring when the design of a technology fails to conform to existing social norms or does not encourage necessary social norms, leading to the technology being rejected by, or is harmful to, society; consequently, the benefit of the artifact is not realized. This framing is close to the definition offered by Piskorski (2014), however Millar (2020) emphasizes a relationship between design decisions and generalized acceptance of an artifact, while my work focuses on the ongoing and dynamic relationship between production and consumption, as well as the ways that consumers may themselves become producers.

However, the term also has been used in a different way entirely—some psychology literature describes people as social failures (e.g. those who are unsuccessful in social situations or are generally unliked), or moments of social failure in the course of someone's life; in this line of inquiry, the focus is on how to treat the deficiencies of an individual (e.g., as in Zentall & Beike, 2012).

6.1.3 Propositions on Social Production Failure

Social production failure as I have sketched it is built on three bodies of theory: market failure in public goods, social failure in collective contexts and mutual exchanges, and observations grounded in commons-based peer production as a particularly salient form of social

production. Building from market failure, I consider the notion that public goods in general are subject to underproduction, and the proposal that some regulating force needs to be imposed to bring supply and demand into better alignment, as well as the observation about the problems of externalities and imperfect information. I also note that Piskorski's (2014) notion of social failure focuses on challenges of information and questions of social norms. I combine the implications of these theories together with my empirical findings and methodoliogical development so far to offer a series of propositions on social production failure.

Proposition 1: Communication problems drive social production failure, that is, social production failures are more likely to occur when communication between producers and consumers is poor. Both market failures and social failures implicate communication problems—incomplete information, inability to communicate, or high communication costs create suboptimal outcomes in all kinds of exchanges. Therefore it seems natural to likewise explore communication failures in the case of social production failures.

Proposition 2: Social production failures are more common when maintenance costs are high relative to creation costs. One of the material conditions of information good production is that some goods require substantial ongoing maintenance, perhaps such that they cannot ever be gauged complete, while others may achieve a degree of completion. As the ongoing effort of sustaining the good rises relative to the efforts of initial production, I would predict higher likelihood of failure. This in part follows from the dynamics I introduced in §2.6, where I described collective action goals as including not only laws and bridges but also the creation and maintenance of systems like governance or digital infrastructure. To the extent that high effort is required to sustain the good, the collective action problem must be solved again and again, and any of those attempts could fail, even if due only to random chance.

This proposition is consistent with findings I presented in Chapter 4 that underproduction levels are higher with older technologies and older languages. This also accords with what we might expect to see based on other related lines of work. Eghbal (2020) describes how

maintenance work may not be as engaging to developers as compared to creating something new. Further, some of these dynamics may be a feature of the nature of software engineering independently of how the production is organized: cost estimation approaches in commercial software engineering suggest that maintenance constitute from 40% to as much as 90% of the total cost (Davis, 2009; Koskinen et al., 2003).

Proposition 3: Social production failures are a failure in organizing; that is, once resources are beyond a minimal level, failures are associated with how work is shared and how resources are organized and governed rather than simply a function of resource pool size. With this proposition, I offer a potential line of explanation for a puzzle emerging from the work presented so far. In the example of Heartbleed that motivated the work in Chapter 3, analysts pointed to the small number of resources maintaining OpenSSL as an important factor in the vulnerability. Further, we should acknowledge that some software is sufficiently large that a single person or even small group cannot sustain it. Yet my findings in Chapter 4 suggest that more resources, in the form of more maintainers and uploaders, are associated with higher levels of underproduction. My relatively flat and coarse measure of teamwork (having maintainership associated with a mailing list rather than an individual) was not statistically significant in models which included language age or network measures—and in the model where team maintenance was statistically significant, increased team maintenance was associated with higher risk, not lower. How should these results be explained?

Part of this unclear picture is undoubtedly due to heterogeneity in software itself. Debian packages are different kinds of artifacts than programming libraries, small applications, or large platforms. Contributions to these varying artifacts vary in terms of size (single line, hundreds of lines) and impact (change in wording, change in algorithm, entire new feature) as well as maintainability considerations such as complexity, architecture, and coupling between files (e.g., how feasible is it to contribute to one part of the code without understanding all of the code?).

In his description of commons-based peer production, Benkler (2006) commented on the value of the tools used in peer production which diminish the cost of collaboration, such as

minimizing the effort required to combine even very small contributions together. Keeping barriers to contribution low is understood to promote participation in online communities of all kinds, including peer production communities (Kraut et al., 2012). Taken together, these factors suggest that there is much to learn about optimal paths that diminish the risk of social production failures. This proposition is further suggested by the findings in Appendix F, published as Gaughan et al. (2024). This work extends the findings from Chapter 3 by examining organizing and governance: hierarchy, formality of project management, and level of power-sharing. We found that formal processes were not statistically significant in association with underproduction, however increased degrees of power-sharing (such that those who contribute to the package also have the power to approve contributions) was associated with lower risk, while formal hierarchies were associated with higher risk. Left unresolved by the work to date is the question of causality (e.g. are these structures predictors or even causes of underproduction, or are they attempted adaptations to address it?). I hope my longitudinal methods development in Chapter 5 will help to resolve this gap in future work.

6.2 A Research Agenda for Investigating Social Production Failure

The relationships between social production failure and such factors as communication problems (**Proposition 1**), task structure (**Proposition 2**), and community structure (**Proposition 3**), as well as such factors as technical decisions, and project and contributor lifecycle remain complex and the trade-offs need further unpacking. However, some social and technical decisions may make underproduction more likely, either as they make sustaining the good more difficult, either materially/technically (**Proposition 2**) or socially (**Proposition 1, 3**). I suggest the following three avenues for future work.

Future work should identify factors serving to regulate the relationship between quality and importance. The economic sciences are an illustration of how much insight can be built from simple and sharp observations about mechanisms; I suggest that in part our challenge in the successful management of public goods like free software is to

build out similarly robust series of techniques for understanding the dynamics of non-market exchange. Of particular interest here is how we might better sustain public goods created through commons-based peer production without sacrificing the innovation and efficiency gain made possible by self-selection of tasks and free distribution.

Future work should map the full lifecycle of social production, including both the building and sustaining of goods as well as the dismantling and successful termination of these goods. Substantial questions remain as to how peer production projects succeed and the factors that support them. However, more is known about how these projects live than how they die. Much of my work (and work in related subfields such as community design and software maintenance) has thus far been oriented to creating, building, sustaining, and maintaining—the generation of public goods and the extending of their useful lifespan. However, it seems increasingly necessary that we also think about what it means to bring these public goods to an end—to replace old ideas with newer and better paradigms, to take advantage of new discoveries, to incorporate different expectations and needs.

As we take stock of how the last three decades of free/libre open source software have evolved, we also need to contend with the time-bounded nature of humans themselves—the people who know a codebase best, whose understanding of it grew as it grew, are themselves growing older. How do we manage these kinds of transitions with minimal harm, allowing new ideas and new leaders to emerge? How do we know when we should stop maintaining and start retiring these public goods? How can we design for these transitions to occur? How might some of the effort involved in creating software be carried forward while other pieces are allowed to retire? For example, core business logic and workflows of even very old software might persist much longer than the implementation of the routines it invokes. A long-developed software platform is likely much easier to replace if we understand not only what it does but why it was designed that way.

Future work should develop and extend evidence for community and governing structures. Along with evolving a more robust understanding of the basic mechanisms

of social production and social production failure, as well as a broad perspective on the rise and fall of projects, there is also a tremendous opportunity to develop interventions and articulate specific and actionable advice to guide the work of practitioners building projects. What steps can a project founder take today to give their project a better chance of surviving in good health for a longer period of time? Organizing work is in some sense overhead—the cost we pay in time or energy to line up resources in ways that seem most optimal so that we can accomplish our actual goal. Given the many ways in which effort might be organized and the many processes, documents, and practices we might invest in creating and adopting, which are most supportive of a thriving project? A project drawing from contributor passion and free time can ill afford to spend human capital on efforts that are unlikely to improve the project's health or success.

Each of these avenues represent urgent areas of need for the sustaining of digital infrastructure, including its ability to continue to serve the public's daily communication, trade, labor, education, and entertainment, and draw from different lines of research. Although underproduction of digital infrastructure software is an example of social production failure, the lack of a price-like regulator in non-market contexts suggests that other kinds of social production failure are possible, in a broad array of contexts, including knowledge, science, and education. I hope that these challenges inspire the selection of tasks for further investigation across the academy.

6.3 Implications for Research

The development of a cross-sectional, ecosystem-level method (Ch. 3) and a longitudinal, project-level method (Ch. 5) for underproduction analysis have multiple implications for scholars of communication, including those who examine society, structure, and organizing. First, these approaches support explicit empirical analysis of questions of collective action, community structure, and the effective generation and management of public goods. Identifying concrete risk conditions allows for the analysis of factors associated with these conditions, including causal analysis. The field of economics has used straightforward observations of

supply and demand to build out a robust collection of theories, measures, and implications. I hope my work can likewise support for the continued development of communication sciences as well. With new methods come not only new empirical results, but also the opportunity to test theories and ideas about how best to organize our efforts to sustain public goods. In Gaughan et al. (2024) (Appendix F, I have evidence of some progress on this front, as we found that underproduction was not associated with formal engineering processes but rather the sharing of power among contributors.

The methods I present in Ch. 3 and 5 are high-level and flexible, adaptable to multiple contexts, measures, and communities. The work I present in Appendix A illustrates this explicitly as I use a different measure of importance and find similar results as demonstrated in Ch. 3. My work presented in Appendix C takes the concept of underproduction and applies it to the case of Wikipedia. Although the goods being produced in Wikipedia are somewhat different from digital infrastructure, being knowledge rather than software, both serve as communication public goods, and both are produced through commons-based peer production. This work suggests that contributors who persist in Wikipedia tend to increasingly choose underproduced articles as areas to contribute, and that this trend occurs whether they are contributing with an account or not.

The implication of this finding for this dissertation is three-fold. First, it demonstrates the use of the underproduction analysis approach in another context. Second, it offers an empirical result which suggests that one approach to countering underproduction is to take greater care in retaining contributors—although the hypotheses were tested in the context of Wikipedia, a long line of research connects Wikipedia and free/libre open source software as having similar social production dynamics. Third, this demonstrates more generally the value of the theoretical approach I am taking: by thinking about concepts such as underproduction and communication public goods, we can build insights in one context to potentially transfer to another context (where such an analysis might be substantially more difficult to conduct). Further, as described in the results I have presented so far, I hope this work has demonstrated that there are many unanswered yet vital questions about how best to sustain our digital

infrastructure—a concern which is deeply communicational but also cuts across disciplinary boundaries.

6.4 Implications for Regulators and Civil Society Leaders

My finding that underproduction is widespread is of great concern to both governmental and non-governmental organizations, including institutions concerned with cybersecurity, defense, economics, and civil society. By coupling this finding with a quantification technique, this work also suggests a strategy for developing policy prescriptions, prioritizing responses, and assessing the impacts of interventions.

The empirical results in Chapters 3-5 suggest some preliminary priorities: not only specific packages in need of some intervention, but also the potential shape of these interventions. The findings in Chapter 4 suggest that intervening to prevent or counter underproduction is not simply a matter of resources—individuals and small teams can do very well. By conducting this analysis in the context of Debian, I chose a particular point in the software supply chain. More resources for Debian would surely be welcome, however Debian is only one step in a series of dependencies—no matter how skilled Debian maintainers are or how well-resourced they might become, their power to counter underproduction is limited because they are part of a far broader and more complex system of systems. They test and integrate software which is largely created elsewhere, which again relies on software created in still other places and by other teams, and all of it is written in languages that are again developed by others. Tracing and taking stock of these social relationships and catalyzing change is a general societal effort, and one in which leadership is needed at multiple levels.

6.5 Implications for Practitioners

The implications of this research are valuable for practitioners as well. The methods I describe make use of measures of quality and importance I selected, with a normative relationship I specified, however the overall approach does not require the same measures—instead, a production community can select measures that make sense for their own goals and environ-

ment. Identifying these measures may itself be a valuable activity for these communities as they work to articulate what and how they want to measure importance and quality, as well as what relationship they are seeking to establish among them. The finding in Ch. 4 that underproduction risk is higher in older packages and in software written in older languages suggests that the problem of underproduction adds further evidence to more general observations about the need to maintain, update, and replace legacy systems. We must confront the aging of three decades worth of creativity and hard work.

The observation of underproduction in GUI components is not necessarily an indicator of grave threats to digital infrastructure—although such components provide a potential attack surface, they are perhaps less a source of risk in such locations as headless cloud environments. However, I observe that the trevails of the GNU/Linux desktop are a long-running source of wry commentary in technical communities—which is to say, although quantification is useful, we already have substantial knowledge about neglected, struggling projects. The results I present may put new priorities on the list, but we should also consider how best to take the knowledge communities already have—including the knowledge encoded as humor, untouchable or "third-rail" topics, superstitions, open secrets, and rough heuristics—and scrutinize it, clarify it, test it, and ultimately include it in both risk assessments and in community organizing research.

Finally, I observe that all of this work is built on publicly available data generated by digital infrastructure development communities. The finding of underproduction in Debian was substantially supported by the fact that Debian has a long history of collecting consistent data, and the findings of underproduction in software written in Python made use of pylint as a community-developed quality measurement tool using community-developed criteria for code quality. Public code and public data are vital for public-serving research. Underproduction is a global problem that can only be tackled through direct engagement between research and practice, and the willingness to make efforts to bridge that divide.

BIBLIOGRAPHY

Abou Khalil, Z., Constantinou, E., Mens, T., Duchien, L., & Quinton, C. (2019). A longitudinal analysis of bug handling across Eclipse releases. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 1–12. https://doi.org/10.1109/ICSME.2019.00010

Ackermann, R. (2023). The future of open source is still very much in flux. *MIT Technology Review*. Retrieved January 30, 2024, from https://www.technologyreview.com/2023/08/17/1077498/future-open-source/

Adams, J., & Brückner, H. (2015). Wikipedia, sociology, and the promise and pitfalls of Big Data. *Big Data & Society*, *2*(2), 2053951715614332. https://doi.org/10.1177/2053951715614332

Adewumi, A., Misra, S., Omoregbe, N., Crawford, B., & Soto, R. (2016). A systematic literature review of open source software quality assessment models. *SpringerPlus*, *5*(1), 1–13. https://doi.org/10.1186/s40064-016-3612-4

Aksulu, A., & Wade, M. (2010). A comprehensive review and synthesis of open source research. *Journal of the Association for Information Systems*, *11*(11/12), 576–656.

Anthony, D. L., Smith, S. W., & Williamson, T. (2009). Reputation and reliability in collective goods: The case of the online encyclopedia Wikipedia. *Rationality and Society*, *21*(3), 283–306. https://doi.org/10.1177/1043463109336804

Arazy, O., & Gellatly, I. R. (2013). *Corporate wikis: The effects of owners' motivation and behavior on group members' engagement* (SSRN Scholarly Paper). http://papers.ssrn.com/abstract=2270644

Ardichvili, A. (2008). Learning and knowledge sharing in virtual communities of practice: Motivators, barriers, and enablers. *Advances in Developing Human Resources*, *10*(4), 541–554. https://doi.org/10.1177/1523422308319536

Asay, M. (2019). The real number of open source developers. *InfoWorld*. https://www.infoworld.com/article/3452881/the-real-number-of-open-source-developers.html

Avelino, G., Constantinou, E., Valente, M. T., & Serebrenik, A. (2019). On the abandonment and survival of open source projects: An empirical investigation. *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 1–12. https://doi.org/10.1109/ESEM.2019.8870181

Avieson, B. (2022). Editors, sources and the 'go back' button: Wikipedia's framework for beating misinformation. *First Monday*. https://doi.org/10.5210/fm.v27i11.12754

Bailey, J., Budgen, D., Turner, M., Kitchenham, B., Brereton, P., & Linkman, S. (2007). Evidence relating to object-oriented software design: A survey. *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 482–484. https://doi.org/10.1109/ESEM.2007.58

Bator, F. M. (1958). The anatomy of market failure. *The Quarterly Journal of Economics*, *72*(3), 351. https://doi.org/10.2307/1882231

Belenzon, S., & Schankerman, M. (2015). Motivation and sorting of human capital in open innovation. *Strategic Management Journal*, *36*(6), 795–820. https://doi.org/10.1002/smj.2284

Benkler, Y. (2002). Coase's Penguin, or, Linux and "The Nature of the Firm". *The Yale Law Journal*, *112*(3), 369. https://doi.org/10.2307/1562247

Benkler, Y. (2006). *The Wealth of Networks: How Social Production Transforms Markets and Freedom*. Yale University Press.

Benkler, Y. (2017). Peer production, the commons, and the future of the firm. *Strategic Organization*, *15*(2), 264–274. https://doi.org/10.1177/1476127016652606

Benkler, Y., Shaw, A., & Hill, B. M. (2015). Peer production: A form of collective intelligence. In T. W. Malone & M. S. Bernstein (Eds.), *Handbook of Collective Intelligence* (pp. 175–204). MIT Press.

Bennett, J. C., Bohoris, G. A., Aspinwall, E. M., & Hall, R. C. (1996). Risk analysis techniques and their application to software development. *European Journal of Operational Research*, *95*(3), 467–475. https://doi.org/10.1016/S0377-2217(96)00171-3

Berdou, E. (2011). *Organization in Open Source Communities: At the Crossroads of the Gift and Market Economies*. Routledge.

Bird, C., Pattison, D., D'Souza, R., Filkov, V., & Devanbu, P. (2008). Latent social structure in open source projects. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 24–35. https://doi.org/10.1145/1453101.1453107

Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative Research in Psychology*, *3*(2), 77–101. https://doi.org/10.1191/1478088706qp063oa

Brooks, F. P. (1995). *The Mythical Man-month: Essays on Software Engineering*. Addison-Wesley Pub. Co.

Bryant, S. L., Forte, A., & Bruckman, A. (2005). Becoming Wikipedian: Transformation of participation in a collaborative online encyclopedia. *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work*, 1–10. https://doi.org/10.1145/1099203.1099205

Budhathoki, N. R., & Haythornthwaite, C. (2013). Motivation for open collaboration: Crowd and community models and the case of OpenStreetMap. *American Behavioral Scientist*, *57*(5), 548–575. https://doi.org/10.1177/0002764212469364

Caneill, M., Germán, D. M., & Zacchiroli, S. (2016). The Debsources Dataset: Two decades of free and open source software. https://doi.org/10.5281/zenodo.61089

Carvalho, M., DeMott, J., Ford, R., & Wheeler, D. A. (2014). Heartbleed 101. *IEEE Security & Privacy*, *12*(4), 63–67. https://doi.org/10.1109/MSP.2014.66

Catolino, G., Palomba, F., & Tamburri, D. A. (2019). The secret life of software communities: What we know and what we don't know. *Proceedings of the 18th Belgium-Netherlands Software Evolution Workshop,*

Champion, K., & Hill, B. M. (2021). Underproduction: An approach for measuring risk in open source software. *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 388–399. https://doi.org/10.1109/SANER50967.2021.00043

Champion, K., & Hill, B. M. (2024). Sources of underproduction in open source software. *The IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. https://doi.org/https://doi.org/10.48550/arXiv.2401.11281

Champion, K., McDonald, N., Bankes, S., Zhang, J., Greenstadt, R., Forte, A., & Hill, B. M. (2019). A forensic qualitative analysis of contributions to Wikipedia from anonymity seeking users. *Proceedings of the ACM on Human-Computer Interaction*, *3*(CSCW), 1–26. https://doi.org/10.1145/3359155

Chatterjee, A., Guizani, M., Stevens, C., Emard, J., May, M. E., Burnett, M., & Ahmed, I. (2021). AID: An automated detector for gender-inclusivity bugs in OSS project pages. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 1423–1435. https://doi.org/10.1109/ICSE43902.2021.00128

Chen, K., & Wagner, D. (2007). Large-scale analysis of format string vulnerabilities in Debian Linux. *Proceedings of the 2007 workshop on Programming languages and analysis for security - PLAS '07*, 75. https://doi.org/10.1145/1255329.1255344

Chidamber, S., & Kemerer, C. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493. https://doi.org/10.1109/32.295895

Choudhary, S., Bogart, C., Rose, C., & Herbsleb, J. (2020). Using productive collaboration bursts to analyze open source collaboration effectiveness. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 400–410. https://doi.org/10.1109/SANER48275.2020.9054852

Christian, J., & Vu, A. N. (2021). Task-based structures in open source software: Revisiting the onion model. *R&D Management*, *51*(1), 87–100. https://doi.org/10.1111/radm.12428

Cinelli, C., Forney, A., & Pearl, J. (2022). A crash course in good and bad controls. *Sociological Methods & Research*, 004912412210995. https://doi.org/10.1177/00491241221099552

CISA Open Source Software Security Roadmap. (2023). Retrieved October 30, 2023, from https://www.cisa.gov/resources-tools/resources/cisa-open-source-software-security-roadmap

Claes, M., Mens, T., Di Cosmo, R., & Vouillon, J. (2015). A historical analysis of Debian package incompatibilities. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 212–223. https://doi.org/10.1109/MSR.2015.27

Coelho, J., & Valente, M. T. (2017). Why modern open source projects fail. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 186–196. https://doi.org/10.1145/3106237.3106246

Coelho, J., Valente, M. T., Silva, L. L., & Hora, A. (2018). Why we engage in FLOSS: Answers from core developers. *CHASE'18*.

Coelho, J., Valente, M. T., Silva, L. L., & Shihab, E. (2018). Identifying unmaintained projects in GitHub. *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 1–10. https://doi.org/10.1145/3239235.3240501

Cordy, J. R., & Roy, C. K. (2011). DebCheck: Efficient checking for open source code clones in software systems. *2011 IEEE 19th International Conference on Program Comprehension*, 217–218. https://doi.org/10.1109/ICPC.2011.27

Crowston, K., Wei, K., Li, Q., & Howison, J. (2007). Self-organization of teams in free/libre open source software development. *Information and Software Technology Journal: Special issue on Understanding the Social Side of Software Engineering*, *49*(6), 564–575.

Crowston, K. (2005). A structurational perspective on leadership in Free/Libre Open Source Software teams.

Crowston, K., Annabi, H., Howison, J., & Masango, C. (2004). Effective work practices for software engineering: Free/libre open source software development. *Proceedings of the 2004 ACM Workshop on Interdisciplinary Software Engineering Research*, 18–26. https://doi.org/10.1145/1029997.1030003

Crowston, K., & Howison, J. (2006a). Assessing the health of open source communities. *IEEE Computer*, *39*(5), 89–91. https://doi.org/10.1109/MC.2006.152

Crowston, K., & Howison, J. (2006b). Hierarchy and centralization in free and open source software team communications. *Knowledge, Technology & Policy*, *18*(4), 65–85. https://doi.org/10.1007/s12130-006-1004-8

Crowston, K., Wei, K., Howison, J., & Wiggins, A. (2012). Free/libre open-source software development: What we know and what we do not know. *ACM Computing Surveys*, *44*(2), 1–35. https://doi.org/10.1145/2089125.2089127

Crowston, K., Wei, K., Li, Q., & Howison, J. (2006). Core and periphery in free/libre and open source software team communications. *Proceedings of the 39th Annual Hawaii International Conference on System Sciences, 2006. HICSS '06*, *6*, 118a. https://doi.org/10.1109/HICSS.2006.101

Cruzes, D. S., & Dybå, T. (2010). Synthesizing evidence in software engineering research. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '10*, 1. https://doi.org/10.1145/1852786.1852788

Csardi, G., & Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal*, *Complex Systems*. https://igraph.org

Cunningham, W. (1993). The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, *4*(2), 29–30. https://doi.org/10.1145/157710.157715

Dabbish, L., Stuart, C., Tsay, J., & Herbsleb, J. (2012). Social coding in GitHub: Transparency and collaboration in an open software repository. *Proceedings of the ACM*

2012 conference on Computer Supported Cooperative Work - CSCW '12, 1277. https://doi.org/10.1145/2145204.2145396

Davies, J., Hanyu Zhang, Nussbaum, L., & German, D. M. (2010). Perspectives on bugs in the Debian bug tracking system. *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 86–89. https://doi.org/10.1109/MSR.2010.5463288

Davis, B. (Ed.). (2009). *97 Things Every Project Manager Should Know: Collective Wisdom From the Experts*. O'Reilly.

De Farias, M. A. F., Colaço, M., Mendonça, M., Novais, R., Da Silva Carvalho, L. P., & Spínola, R. O. (2016). A systematic mapping study on mining software repositories. *Proceedings of the ACM Symposium on Applied Computing*, *04-08-April-2016*, 1472–1479. https://doi.org/10.1145/2851613.2851786

De Stefano, M., Iannone, E., Pecorelli, F., & Tamburri, D. A. (2022). Impacts of software community patterns on process and product: An empirical study. *Science of Computer Programming*, *214*, 102731. https://doi.org/10.1016/j.scico.2021.102731

de Laat, P. B. (2007). Governance of open source software: State of the art. *Journal of Management & Governance*, *11*(2), 165–177. https://doi.org/10.1007/s10997-007-9022-9

Djuric, D. (2015). Penetrating the omerta of predatory publishing: The Romanian connection. *Science and Engineering Ethics*, *21*(1), 183–202. https://doi.org/10.1007/s11948-014-9521-4

Dueñas, S., Cosentino, V., Robles, G., & Gonzalez-Barahona, J. M. (2018). Perceval: Software project data at your will. *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 1–4. https://doi.org/10.1145/3183440.3183475

Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., & Halderman, J. A. (2014). The matter of Heartbleed. *Proceedings of the 2014 Conference on Internet Measurement*, 475–488. https://doi.org/10.1145/2663716.2663755

Eghbal, N. (2016). Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure.

Eghbal, N. (2020). *Working In Public: The Making and Maintenance of Open Source Software*. Stripe Press.

Ernst, N. A. (2018). Bayesian hierarchical modelling for tailoring metric thresholds. *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*, 587–591. https://doi.org/10.1145/3196398.3196443

Evans, S., Mabey, J., & Mandiberg, M. (2015). Editing for equality: The outcomes of the Art+Feminism Wikipedia Edit-a-thons. *Art Documentation: Journal of the Art Libraries Society of North America*, *34*(2), 194–203. https://doi.org/10.1086/683380

Falcao, F., Barbosa, C., Fonseca, B., Garcia, A., Ribeiro, M., & Gheyi, R. (2020). On relating technical, social factors, and the introduction of bugs. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 378–388. https://doi.org/10.1109/SANER48275.2020.9054824

Farič, N., & Potts, H. W. (2014). Motivations for contributing to health-related articles on Wikipedia: An interview study. *Journal of Medical Internet Research*, *16*(12), e260. https://doi.org/10.2196/jmir.3569

Fernandez Del Carpio, A., & Angarita, L. B. (2018). Techniques based on data science for software processes: A systematic literature review. *18th International Conference on Software Process Improvement and Capability Determination, SPICE 2018, October 9, 2018 - October 10, 2018*, *918*, 16–30. https://doi.org/10.1007/978-3-030-00623-5_2

Finley, K. (2019). For open source, it's all about GitHub now. *Wired*. Retrieved November 20, 2023, from https://www.wired.com/story/open-source-all-about-github-now/

Ford, H., Pensa, I., Devouard, F., Pucciarelli, M., & Botturi, L. (2018). Beyond notification: Filling gaps in peer production projects. *New Media & Society*, *20*(10), 3799–3817. https://doi.org/10.1177/1461444818760870

Forte, A., Andalibi, N., & Greenstadt, R. (2017). Privacy, anonymity, and perceived risk in open collaboration: A study of Tor users and Wikipedians. *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, 1800–1811. https://doi.org/10.1145/2998181.2998273

Foucault, M., Palyart, M., Blanc, X., Murphy, G. C., & Falleri, J.-R. (2015). Impact of developer turnover on quality in open-source software. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 829–841. https://doi.org/10.1145/2786805.2786870

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Fulk, J., Flanagin, A. J., Kalman, M. E., Monge, P. R., & Ryan, T. (1996). Connective and communal public goods in interactive communication systems. *Communication Theory*, *6*(1), 60–87. https://doi.org/10.1111/j.1468-2885.1996.tb00120.x

Furton, G., & Martin, A. (2019). Beyond market failure and government failure. *Public Choice*, *178*(1-2), 197–216. https://doi.org/10.1007/s11127-018-0623-4

Galindo Duarte, J. A., Benavides Cuevas, D. F., & Segura Rueda, S. (2010). Debian packages repositories as software product line models: Towards automated analysis. *First International Workshop on Automated Configuration and Tailoring of Applications*.

Gallagher, K., Fioravanti, M., & Kozaitis, S. (2019). Teaching software maintenance. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 353–362. https://doi.org/10.1109/ICSME.2019.00054

Gaughan, M., Champion, K., & Hwang, S. (2024). Engineering formality and software risk in Debian Python packages. *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

Geiger, R. S., & Halfaker, A. (2017). Operationalizing conflict and cooperation between automated software agents in Wikipedia: A replication and expansion of 'Even good bots fight'. *Proceedings of the ACM on Human-Computer Interaction*, *1*(CSCW), 1–33. https://doi.org/10.1145/3134684

Germonprez, M., Lipps, J., & Goggins, S. (2019). The rising tide: Open source's steady transformation. *First Monday*, *24*(8). https://doi.org/10.5210/fm.v24i8.9297

Gonzalez-Barahona, J. M., Robles, G., Herraiz, I., & Ortega, F. (2014). Studying the laws of software evolution in a long-lived FLOSS project: Studying the Laws of Software

Evolution. *Journal of Software: Evolution and Process*, *26*(7), 589–612. https://doi.org/10.1002/smr.1615

Gorbatai, A. D. (2011a). *Aligning collective production with demand: Evidence from Wikipedia* (Working Paper). http://faculty.chicagobooth.edu/workshops/orgs-markets/past/pdf/gorbatai.pdf

Gorbatai, A. D. (2011b). Exploring underproduction in Wikipedia. *Proceedings of the 7th International Symposium on Wikis and Open Collaboration*, 205. https://doi.org/10.1145/2038558.2038595

Gousios, G., Pinzger, M., & Deursen, A. v. (2014). An exploratory study of the pull-based software development model. *Proceedings of the 36th International Conference on Software Engineering*, 345–355. https://doi.org/10.1145/2568225.2568260

Gritzalis, D., Iseppi, G., Mylonas, A., & Stavrou, V. (2018). Exiting the Risk Assessment Maze: A Meta-Survey. *ACM Computing Surveys*, *51*(1), 1–30. https://doi.org/10.1145/3145905

Halfaker, A. (2017). Interpolating quality dynamics in Wikipedia and demonstrating the Keilana Effect, 1–9. https://doi.org/10.1145/3125433.3125475

Halfaker, A., Geiger, R. S., Morgan, J. T., & Riedl, J. (2013). The rise and decline of an open collaboration system: How Wikipedia's reaction to popularity is causing its decline. *American Behavioral Scientist*, *57*(5), 664–688. https://doi.org/10.1177/0002764212469365

Halfaker, A., Morgan, J., Sarabadani, A., & Wight, A. (2016). *ORES: Facilitating re-mediation of Wikipedia's socio-technical problems* (Working Paper). Wikimedia Research.

Halstead, M. H. (1979). *Elements of Software Science* (3. pr). North Holland.

Hannebauer, C., & Gruhn, V. (2016). Motivation of newcomers to FLOSS projects. *Proceedings of the 12th International Symposium on Open Collaboration*, 1–10. https://doi.org/10.1145/2957792.2957793

Healy, K., & Schussman, A. (2003). *The ecology of open-source software development.*

Herfort, B., Lautenbach, S., Porto de Albuquerque, J., Anderson, J., & Zipf, A. (2023). A spatio-temporal analysis investigating completeness and inequalities of global urban building data in OpenStreetMap. *Nature Communications*, *14*(1), 3985. https://doi.org/10.1038/s41467-023-39698-6

Herraiz, I., Shihab, E., Nguyen, T. H., & Hassan, A. E. (2011). Impact of installation counts on perceived quality: A case study on Debian. *2011 18th Working Conference on Reverse Engineering*, 219–228. https://doi.org/10.1109/WCRE.2011.34

Hilderbrand, C., Perdriau, C., Letaw, L., Emard, J., Steine-Hanson, Z., Burnett, M., & Sarma, A. (2020). Engineering gender-inclusivity into software: Ten teams' tales from the trenches. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 433–444. https://doi.org/10.1145/3377811.3380371

Hill, B. M., & Shaw, A. (2013). The Wikipedia gender gap revisited: Characterizing survey response bias with propensity score estimation (A. Sánchez, Ed.). *PLoS ONE*, *8*(6), e65782. https://doi.org/10.1371/journal.pone.0065782

Hill, B. M., & Shaw, A. (2014). Consider the redirect: A missing dimension of Wikipedia research. *Proceedings of The International Symposium on Open Collaboration*, 28:1–28:4. https://doi.org/10.1145/2641580.2641616

Hill, B. M., & Shaw, A. (2015). Page protection: Another missing dimension of Wikipedia research. *Proceedings of the 11th International Symposium on Open Collaboration*, 15:1–15:4. https://doi.org/10.1145/2788993.2789846

Hill, B. M., & Shaw, A. (2021). The hidden costs of requiring accounts: Quasi-experimental evidence From peer production. *Communication Research*, *48*(6), 771–795. https://doi.org/10.1177/0093650220910345

Hindle, A., Herraiz, I., Shihab, E., & Zhen Ming Jiang. (2010). Mining Challenge 2010: FreeBSD, GNOME Desktop and Debian/Ubuntu. *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 82–85. https://doi.org/10.1109/MSR.2010.5463350

Holtzblatt, L. J., Damianos, L. E., & Weiss, D. (2010). Factors impeding wiki use in the enterprise: A case study. *CHI '10 Extended Abstracts on Human Factors in Computing Systems*, 4661–4676. https://doi.org/10.1145/1753846.1754208

Hope, A. (2020). Millions of vulnerable systems unpatched for severe bugs including BlueKeep and Heartbleed despite NSA warnings. *CPO Magazine*. Retrieved November 11, 2021, from https://www.cpomagazine.com/cyber-security/millions-of-vulnerable-systems-unpatched-for-severe-bugs-including-bluekeep-and-heartbleed-despite-nsa-warnings/

Houtti, M., Johnson, I., Cepeda, J., Khandelwal, S., Bhatnagar, A., & Terveen, L. (2022). "We need a woman in music": Exploring Wikipedia's values on article priority. *Proceedings of the ACM on Human-Computer Interaction*, *6*(CSCW2), 1–28. https://doi.org/10.1145/3555156

Houtti, M., Johnson, I., & Terveen, L. (2023). Leveraging recommender systems to reduce content gaps on peer production platforms [arXiv:2307.08669 [cs]]. Retrieved January 30, 2024, from http://arxiv.org/abs/2307.08669

Hwang, S., & Shaw, A. (2022). Rules and rule-making in the five largest Wikipedias. *Proceedings of the International AAAI Conference on Web and Social Media*, *16*, 347–357. https://doi.org/10.1609/icwsm.v16i1.19297

Jiang, Y., Adams, B., & German, D. M. (2013). Will my patch make it? And how fast? Case study on the Linux kernel [ISSN: 2160-1860]. *2013 10th Working Conference on Mining Software Repositories (MSR)*, 101–110. https://doi.org/10.1109/MSR.2013.6624016

Joblin, M., & Apel, S. (2022). How do successful and failed projects differ? A socio-technical analysis. *ACM Transactions on Software Engineering and Methodology*, *31*(4), 1–24. https://doi.org/10.1145/3504003

Johnson, M. (2015). Public goods, market failure, and voluntary exchange. *History of Political Economy*, *47*, 174–198. https://doi.org/10.1215/00182702-3130499

Juristo, N., Vegas, S., & Gomez G., O. S. (2010). Replication, reproduction and re-analysis: Three ways for verifying experimental findings. *Proceedings 1st Int. Workshop on Replication in Empirical Software Eng. Research (RESER 2010)*.

Karczewska, A. (2023). GLAM WikiProjects as a form of organization of cooperation between Wikipedia and cultural institutions. *European Conference on Knowledge Management*, *24*(1), 619–627. https://doi.org/10.34190/eckm.24.1.1374

KernelCare. (2020). Heartbleed still found in the wild: Did you know that you may be vulnerable? Retrieved November 11, 2021, from https://linuxize.com/post/heartbleed-still-found-in-the-wild/

Kerner, S. M. (2014). Heartbleed SSL flaw's true cost will take time to tally. *eWEEK*. Retrieved October 9, 2022, from https://www.eweek.com/security/heartbleed-ssl-flaw-s-true-cost-will-take-time-to-tally/

Kim, S., & Whitehead, E. J. (2006). How long did it take to fix bugs? *Proceedings of the 2006 International Workshop on Mining Software Repositories - MSR '06*, 173. https://doi.org/10.1145/1137983.1138027

Kitchenham, B., Brereton, O. P., Budgen, D., Turner, M., Bailey, J., & Linkman, S. (2009). Systematic literature reviews in software engineering: A systematic literature review. *Information and Software Technology*, *51*(1), 7–15. https://doi.org/10.1016/j.infsof.2008.09.009

Kitchenham, B., Dyba, T., & Jorgensen, M. (2004). Evidence-based software engineering. *Proceedings, 26th International Conference on Software Engineering*, 273–281. https://doi.org/10.1109/ICSE.2004.1317449

Konieczny, P. (2012). Wikis and Wikipedia as a teaching tool: Five years later. *First Monday*. https://doi.org/10.5210/fm.v0i0.3583

Koskinen, J., Lahtonen, H., & Tilus, T. (2003). *Software maintenance cost estimation and modernization support* (tech. rep.). Information Technology Research Institute.

Kraut, R. E., Resnick, P., & Kiesler, S. (2012). *Building successful online communities: Evidence-based social design*. MIT Press.

Krill, P. (2022). Python popularity still soaring. *InfoWorld*. Retrieved November 26, 2023, from https://www.infoworld.com/article/3669232/python-popularity-still-soaring.html

Krill, P. (2023). Report finds few open source projects actively maintained. *InfoWorld*. Retrieved October 30, 2023, from https://www.infoworld.com/article/3708630/report-finds-few-open-source-projects-actively-maintained.html

Krishnamurthy, S., Ou, S., & Tripathi, A. K. (2014). Acceptance of monetary rewards in open source software development. *Research Policy*, *43*(4), 632–644. https://doi.org/10.1016/j.respol.2013.10.007

Kuznetsov, S. (2006). Motivations of contributors to Wikipedia. *ACM SIGCAS Computers and Society*, *36*. https://doi.org/10.1145/1215942.1215943

Lakhani, K. R., & Wolf, B. (2005). Why hackers do what they do: Understanding motivation and effort in free/open source software projects. In J. Feller, B. Fitzgerald, S. A. Hissam, & K. R. Lakhani (Eds.), *Perspectives on Free and Open Source Software* (pp. 3–22). MIT Press.

Langrock, I., & González-Bailón, S. (2022). The gender divide in Wikipedia: Quantifying and assessing the impact of two feminist interventions. *Journal of Communication*. https://doi.org/10.1093/joc/jqac004

Lave, J., & Wenger, E. (1991). *Situated Learning: Legitimate Peripheral Participation*. Cambridge University Press.

Le, D. M., Link, D., Shahbazian, A., & Medvidovic, N. (2018). An empirical study of architectural decay in open-source software. *2018 IEEE International Conference on Software Architecture (ICSA)*, 176–17609. https://doi.org/10.1109/ICSA.2018.00027

Lee, C. P., Dourish, P., & Mark, G. (2006). The human infrastructure of cyberinfrastructure. *Proceedings of the 2006 20th anniversary conference on Computer Supported Cooperative Work - CSCW '06*, 483. https://doi.org/10.1145/1180875.1180950

Log4Shell 10 days later: Enterprises halfway through patching. (2021). *Wiz.IO*. Retrieved October 6, 2023, from https://www.wiz.io/blog/10-days-later-enterprises-halfway-through-patching-log4shell

Mahmood, Z., Bowes, D., Hall, T., Lane, P. C., & Petrić, J. (2018). Reproducibility and replicability of software defect prediction studies. *Information and Software Technology*, *99*, 148–163. https://doi.org/10.1016/j.infsof.2018.02.003

March, L., & Dasgupta, S. (2020). Wikipedia edit-a-thons as sites of public pedagogy. *Proceedings of the ACM on Human-Computer Interaction*, *100:1–100:26*(CSCW2). https://doi.org/10.1145/3415171

Marciano, A., & Medema, S. G. (2015). Market failure in context: Introduction. *History of Political Economy*, *47*(1), 1–19. https://doi.org/10.1215/00182702-3130415

Marwell, G., & Oliver, P. (1993). *The Critical Mass in Collective Action: A Micro-social Theory*. Cambridge University Press.

Mateos-Garcia, J., & Steinmueller, W. E. (2008). The institutions of open source software: Examining the Debian community. *Information Economics and Policy*, *20*(4), 333–344. https://doi.org/10.1016/j.infoecopol.2008.06.001

Mauerer, W., Joblin, M., Tamburri, D. A., Paradis, C., Kazman, R., & Apel, S. (2022). In search of socio-technical congruence: A large-scale longitudinal study. *IEEE Transactions on Software Engineering*, *48*(8), 3159–3184. https://doi.org/10.1109/TSE.2021.3082074

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

McDowell, Z. J., & Vetter, M. A. (2022). Wikipedia as open educational practice: Experiential learning, critical information literacy, and social justice. *Social Media + Society*, *8*(1). https://doi.org/10.1177/20563051221078224

McMahon, C., Johnson, I. L., & Hecht, B. J. (2017). The substantial interdependence of Wikipedia and Google: A case study on the relationship between peer production

communities and information technologies. *International AAAI Conference on Web and Social Media (ICWSM 2017)*, 142–151.

Meidan, A., García-García, J. A., Ramos, I., & Escalona, M. J. (2018). Measuring software process: A systematic mapping study. *ACM Computing Surveys*, *51*(3), 1–32. https://doi.org/10.1145/3186888

Meng, B., & Wu, F. (2013). COMMONS/COMMODITY: Peer production caught in the Web of the commercial market. *Information, Communication & Society*, *16*(1), 125–145. https://doi.org/10.1080/1369118X.2012.675347

Meyer, J. W., & Rowan, B. (1977). Institutionalized organizations: Formal structure as myth and ceremony. *American Journal of Sociology*, *83*(2), 340–363. Retrieved October 9, 2008, from http://www.jstor.org/stable/2778293

Michlmayr, M., & Hill, B. M. (2003). Quality and the reliance on individuals in free software projects. *3rd Workshop on Open Source Software Engineering, ICSE*.

Michlmayr, M., & Senyard, A. (2006). A statistical analysis of defects in Debian and strategies for improving quality in free software projects. *The Economics of Open Source Software Development* (pp. 131–148). Elsevier. Retrieved July 18, 2020, from https://linkinghub.elsevier.com/retrieve/pii/B9780444527691500068

Millar, J. (2020). Social failure modes in technology and the ethics of AI: An engineering perspective. In M. D. Dubber, F. Pasquale, & S. Das (Eds.), *The Oxford Handbook of Ethics of AI* (pp. 441–461). Oxford University Press. https://doi.org/10.1093/oxfordhb/9780190067397.013.28

Miller, C., Cohen, S., Klug, D., Vasilescu, B., & Kastner, C. (2022). "Did you miss my comment or what?": Understanding toxicity in open source discussions. *Proceedings of the 44th International Conference on Software Engineering*, 710–722. https://doi.org/10.1145/3510003.3510111

Miller, C., Widder, D. G., Kastner, C., & Vasilescu, B. (2019). Why do people give up FLOSSing? A study of contributor disengagement in open source. In F. Bordeleau,

A. Sillitti, P. Meirelles, & V. Lenarduzzi (Eds.), *Open Source Systems* (pp. 116–129). Springer International Publishing. https://doi.org/10.1007/978-3-030-20883-7_11

Mockus, A. (2010). Organizational volatility and its effects on software defects. *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 117–126. https://doi.org/10.1145/1882291.1882311

Mockus, A., Fielding, R. T., & Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *11*(3), 309–346. https://doi.org/10.1145/567793.567795

Monge, P. R., Fulk, J., Kalman, M. E., Flanagin, A. J., Parnassa, C., & Rumsey, S. (1998). Production of collective action in alliance-based interorganizational communication and information systems. *Organization Science*, *9*(3), 411–433. https://doi.org/10.1287/orsc.9.3.411

Morgan, J. T., & Halfaker, A. (2018). Evaluating the impact of the Wikipedia Teahouse on newcomer socialization and retention. *Proceedings of the 14th International Symposium on Open Collaboration*, 20:1–20:7. https://doi.org/10.1145/3233391.3233544

Nassif, M., & Robillard, M. P. (2017). Revisiting turnover-induced knowledge loss in software projects. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 261–272. https://doi.org/10.1109/ICSME.2017.64

Natella, R., Cotroneo, D., & Madeira, H. S. (2016). Assessing dependability with software fault injection: A survey. *ACM Computing Surveys*, *48*(3), 1–55. https://doi.org/10.1145/2841425

Nguyen, R., & Holt, R. (2012). Life and death of software packages: An evolutionary study of Debian. *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, 192–204.

Nussbaum, L., & Zacchiroli, S. (2010). The Ultimate Debian Database: Consolidating bazaar metadata for quality assurance and data mining. *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 52–61. https://doi.org/10.1109/MSR.2010.5463277

Oganisian, A., & Roy, J. A. (2021). A practical introduction to Bayesian estimation of causal effects: Parametric and nonparametric approaches. *Statistics in Medicine*, *40*(2), 518–551. https://doi.org/10.1002/sim.8761

Okoli, C. (2015). A guide to conducting a standalone systematic literature review. *Communications of the Association for Information Systems*, *37*. https://doi.org/10.17705/1CAIS.03743

Oliveira, N., Ribeiro, M., Bonifacio, R., Gheyi, R., Wiese, I., & Fonseca, B. (2022). Lint-based warnings in Python code: Frequency, awareness and refactoring. *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 208–218. https://doi.org/10.1109/SCAM55253.2022.00030

Olson, M. (1965). *The Logic of Collective Action: Public Goods and the Theory of Groups*. Harvard University Press.

O'Neil, M. (2009). *Cyberchiefs: Autonomy and Authority in Online Tribes*. Pluto Press; Palgrave Macmillan.

Onoue, S., Hata, H., & Matsumoto, K. (2014). Software population pyramids: The current and the future of OSS development communities. *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 1–4. https://doi.org/10.1145/2652524.2652565

Oreg, S., & Nov, O. (2008). Exploring motivations for contributing to open source initiatives: The roles of contribution context and personal values. *Computers in Human Behavior*, *24*(5), 2055–2073. https://doi.org/10.1016/j.chb.2007.09.007

Ostrom, E. (2015). *Governing the Commons: The Evolution of Institutions for Collective Action*. Cambridge University Press. https://doi.org/10.1017/CBO9781316423936

Pagliery, J. (2014). Your Internet security relies on a handful of volunteers. *CNNMoney*. Retrieved November 11, 2021, from https://money.cnn.com/2014/04/18/technology/security/heartbleed-volunteers/

Panciera, K., Halfaker, A., & Terveen, L. (2009). Wikipedians are born, not made: A study of power editors on Wikipedia. *Proceedings of the ACM 2009 International conference on Supporting Group Work*, 51–60. https://doi.org/10.1145/1531674.1531682

Pasha, M., Qaiser, G., & Pasha, U. (2018). A critical analysis of software risk management techniques in large scale systems. *IEEE Access*, *6*, 12412–12424. https://doi.org/10.1109/ACCESS.2018.2805862

Pati, J., & Shukla, K. K. (2014). A comparison of ARIMA, neural network and a hybrid technique for Debian bug number prediction. *2014 International Conference on Computer and Communication Technology (ICCCT)*, 47–53. https://doi.org/10.1109/ICCCT.2014.7001468

Paul, C. L. (2009). A survey of usability practices in free/libre/open source software. In C. Boldyreff, K. Crowston, B. Lundell, & A. I. Wasserman (Eds.), *Open Source Ecosystems: Diverse Communities Interacting* (pp. 264–273). Springer Berlin Heidelberg. Retrieved September 11, 2020, from http://link.springer.com/10.1007/978-3-642-02032-2_23

Paul, I. (2014). In Heartbleed's wake, tech titans launch fund for crucial open-source projects. *PCWorld*. Retrieved November 11, 2021, from https://www.pcworld.com/article/438906

Perlroth, N. (2014). Heartbleed highlights a contradiction in the web. *The New York Times*. Retrieved November 11, 2021, from https://www.nytimes.com/2014/04/19/technology/heartbleed-highlights-a-contradiction-in-the-web.html

Piskorski, M. J. (2014). *A Social Strategy: How We Profit from Social Media*. Princeton University Press.

Preece, J., & Shneiderman, B. (2009). The reader-to-leader framework: Motivating technology-mediated social participation. *AIS Transactions on Human-Computer Interaction*, *1*(1), 13–32.

Qiu, H. S., Lieb, A., Chou, J., Carneal, M., Mok, J., Amspoker, E., Vasilescu, B., & Dabbish, L. (2023). Climate Coach: A dashboard for open-source maintainers to overview

community dynamics. *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 1–18. https://doi.org/10.1145/3544548.3581317

Qiu, H. S., Nolte, A., Brown, A., Serebrenik, A., & Vasilescu, B. (2019). Going farther together: The impact of social capital on sustained participation in open source. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 688–699. https://doi.org/10.1109/ICSE.2019.00078

Qiu, H. S., Zhao, Z. H., Yu, T. K., Wang, J., Ma, A., Fang, H., Dabbish, L., & Vasilescu, B. (2023). Gender representation among contributors to open-source infrastructure : An analysis of 20 package manager ecosystems. *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, 180–187. https://doi.org/10.1109/ICSE-SEIS58686.2023.00025

R Core Team. (2024). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. https://www.R-project.org/

Rafaeli, S., & Ariel, Y. (2008). Online motivational factors: Incentives for participation and contribution in Wikipedia. In A. Barak (Ed.), *Psychological Aspects of Cyberspace: Theory, Research, Applications*. Cambridge University Press.

Raulo, A., Rojas, A., Kröger, B., Laaksonen, A., Orta, C. L., Nurmio, S., Peltoniemi, M., Lahti, L., & Žliobaitė, I. (2023). What are patterns of rise and decline? *Royal Society Open Science*, *10*(11), 230052. https://doi.org/10.1098/rsos.230052

Ray, B., Posnett, D., Filkov, V., & Devanbu, P. (2014). A large scale study of programming languages and code quality in GitHub. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, 155–165. https://doi.org/10.1145/2635868.2635922

Raymond, E. S. (1999). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary* (T. O'Reilly, Ed.). O'Reilly; Associates.

Restivo, M., & van de Rijt, A. (2014). No praise without effort: Experimental evidence on how rewards affect Wikipedia's contributor community. *Information, Communication & Society*, *17*(4), 1–12. https://doi.org/10.1080/1369118X.2014.888459

Robles, G., Gonzalez-Barahona, J. M., & Michlmayr, M. (2005). Evolution of volunteer participation in libre software projects. *Proceedings of the First International Conference on Open Source Systems*.

Robles, G., González-Barahona, J. M., Izquierdo-Cortazar, D., & Herraiz, I. (2009). Tools for the study of the usual data sources found in libre software projects. *International Journal of Open Source Software and Processes*, *1*(1), 24–45. https://doi.org/10.4018/jossp.2009010102

Ronchieri, E., & Canaparo, M. (2018). Metrics for software reliability: A systematic mapping study. *Journal of Integrated Design & Process Science*, *22*(2), 5–25. https://doi.org/10.3233/JID180008

Ruiz, C., & Robinson, W. N. (2011). Measuring open source quality: A literature review. *International Journal of Open Source Software and Processes*, *3*(3), 48–65. https://doi.org/10.4018/jossp.2011070104

Sadowski, B. M., Sadowski-Rasters, G., & Duysters, G. (2008). Transition of governance in a mature open software source community: Evidence from the Debian case. *Information Economics and Policy*, *20*(4), 323–332. https://doi.org/10.1016/j.infoecopol.2008.05.001

Safadi, H., Johnson, S. L., & Faraj, S. (2021). Who contributes knowledge? Core-periphery tension in online innovation communities. *Organization Science*, *32*(3), 752–775. https://doi.org/10.1287/orsc.2020.1364

Santos, J. A. M., Rocha-Junior, J. B., Lins Prates, L. C., do Nascimento, R. S., Freitas, M. F., & de Mendonca, M. G. (2018). A systematic review on the code smell effect. *Journal of Systems and Software*, *144*, 450–477. https://doi.org/10.1016/j.jss.2018.07.035

Sargent, D. J. (1998). A general framework for random effects survival analysis in the Cox proportional hazards setting. *Biometrics*, *54*(4), 1486. https://doi.org/10.2307/2533673

Sarkar, S., Waldman-Brown, A., & Clegg, S. (2023). A digital ecosystem as an institutional field: Curated peer production as a response to institutional voids revealed by COVID -19. *R&D Management*, *53*(4), 695–708. https://doi.org/10.1111/radm.12555

Schmahl, K. G., Viering, T. J., Makrodimitris, S., Naseri Jahfari, A., Tax, D., & Loog, M. (2020). Is Wikipedia succeeding in reducing gender bias? Assessing changes in gender bias in Wikipedia using word embeddings. *Proceedings of the Fourth Workshop on Natural Language Processing and Computational Social Science*, 94–103. https://doi.org/10.18653/v1/2020.nlpcss-1.11

Schweik, C. M., & English, R. C. (2012). *Internet Success: A Study of Open-Source Software Commons*. MIT Press.

Schweik, C. M., English, R. C., Kitsing, M., & Haire, S. (2008). Brooks' Versus Linus' Law: An empirical test of open source projects. *Proceedings of the 2008 International Conference on Digital Government Research*, 423–424.

Shah, S. K. (2006). Motivation, governance, and the viability of hybrid forms in open source software development. *Management Science*, *52*(7), 1000–1014. https://doi.org/10.1287/mnsc.1060.0553

Shaw, A., & Hill, B. M. (2014). Laboratories of oligarchy? How the iron law extends to peer production. *Journal of Communication*, *64*(2), 215–238. https://doi.org/10.1111/jcom.12082

Shirky, C. (2010). *Cognitive Surplus: Creativity and Generosity in a Connected Age*.

Silva, J. O., Wiese, I., German, D. M., Treude, C., Gerosa, M. A., & Steinmacher, I. (2020). Google summer of code: Student motivations and contributions. *Journal of Systems and Software*, *162*, 110487. https://doi.org/10.1016/j.jss.2019.110487

Smith, V., Devane, D., Begley, C. M., & Clarke, M. (2011). Methodology in conducting a systematic review of systematic reviews of healthcare interventions. *BMC Medical Research Methodology*, *11*(1), 15. https://doi.org/10.1186/1471-2288-11-15

Sobrinho, E. V. d. P., De Lucia, A., & Maia, M. d. A. (2018). A systematic literature review on bad smells. 5 W's: Which, when, what, who, where. *IEEE Transactions on Software Engineering*. https://doi.org/10.1109/TSE.2018.2880977

Sonatype. (2023). *9th Annual State of the Software Supply Chain* (tech. rep.).

Spaeth, S., Stuermer, M., Haefliger, S., & von Krogh, G. (2007). Sampling in open source software development: The case for using the Debian GNU/Linux distribution. *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. https://doi.org/10.1109/HICSS.2007.471

Spaeth, S., von Krogh, G., Stuermer, M., & Haefliger, S. (2008). *A lightweight model of component reuse: A study of software packages in Debian GNU/Linux* (Working Paper). https://ssrn.com/abstract=1153968

Stallman, R. M. (2002). *Free software, free society: Selected essays of Richard M. Stallman* (J. Gay, Ed.). http://www.gnu.org/doc/Press-use/fsfs3-hardcover.pdf

Steinmacher, I., Graciotto Silva, M. A., Gerosa, M. A., & Redmiles, D. F. (2015). A systematic literature review on the barriers faced by newcomers to open source software projects. *Information and Software Technology*, *59*, 67–85. https://doi.org/10.1016/j.infsof.2014.11.001

Stergiopoulos, G., Kotzanikolaou, P., Theocharidou, M., Lykou, G., & Gritzalis, D. (2016). Time-based critical infrastructure dependency analysis for large-scale and cross-sectoral failures. *International Journal of Critical Infrastructure Protection*, *12*, 46–60. https://doi.org/10.1016/j.ijcip.2015.12.002

Studer, M. (2007). Community structure, individual participation and the social construction of merit. In J. Feller, B. Fitzgerald, W. Scacchi, & A. Sillitti (Eds.), *Open Source Development, Adoption and Innovation* (pp. 161–172). Springer US. http://link.springer.com/chapter/10.1007/978-0-387-72486-7_13

Syeed, M. M. M., Hammouda, I., & Systä, T. (2014). Prediction models and techniques for Open Source Software projects: A systematic literature review. *International Journal*

of Open Source Software and Processes, *5*(2), 1–39. https://doi.org/10.4018/ijossp.2014040101

Tamburri, D. A., Lago, P., & Vliet, H. v. (2013). Organizational social structures for software engineering. *ACM Computing Surveys*, *46*(1), 3:1–3:35. https://doi.org/10.1145/2522968.2522971

Tamburri, D. A. A., Palomba, F., & Kazman, R. (2019). Exploring community smells in Open-Source: An automated approach. *IEEE Transactions on Software Engineering*, 1–1. https://doi.org/10.1109/TSE.2019.2901490

Tan, X., & Zhou, M. (2022). Scaling open source software communities: Challenges and practices of decentralization. *IEEE Software*, *39*(1), 70–75. https://doi.org/10.1109/MS.2020.3025959

Tan, X., Zhou, M., & Sun, Z. (2020). A first look at good first issues on GitHub. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 398–409. https://doi.org/10.1145/3368089.3409746

Team, S. D. (2023). RStan: The R interface to Stan. Retrieved June 9, 2023, from https://mc-stan.org/

TeBlunthuis, N., Shaw, A., & Hill, B. M. (2018). Revisiting "The rise and decline" in a population of peer production projects. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 355:1–355:7. https://doi.org/10.1145/3173574.3173929

Textor, J., Van Der Zander, B., Gilthorpe, M. S., Liśkiewicz, M., & Ellison, G. T. (2017). Robust causal inference using directed acyclic graphs: The R package 'dagitty'. *International Journal of Epidemiology*, dyw341. https://doi.org/10.1093/ije/dyw341

Thebault-Spieker, J., Hecht, B., & Terveen, L. (2018). Geographic Biases are 'Born, not Made': Exploring contributors' spatiotemporal behavior in OpenStreetMap, 71–82. https://doi.org/10.1145/3148330.3148350

Thomas, R. (2015a). Market Efficiency. *Worldmark Global Business and Economy Issues*. Gale.

Thomas, R. (2015b). Market Failure. *Worldmark Global Business and Economy Issues*. Gale.

Tourani, P., Adams, B., & Serebrenik, A. (2017). Code of conduct in open source projects. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 24–33. https://doi.org/10.1109/SANER.2017.7884606

Tracy, P. E., & Carkin, D. M. (2014). Adjusting for design effects in disproportionate stratified sampling designs through weighting. *Crime & Delinquency*, *60*(2), 306–325. https://doi.org/10.1177/0011128714522114

Tripodi, F. (2021). Ms. Categorized: Gender, notability, and inequality on Wikipedia. *New Media & Society*. https://doi.org/10.1177/14614448211023772

Valiev, M., Vasilescu, B., & Herbsleb, J. (2018). Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 644–655. https://doi.org/10.1145/3236024.3236062

van Meijel, J. (2021). *On the Relations Between Community Patterns and Smells in Open-Source: A Taxonomic and Empirical Analysis* (Master's thesis). Eindhoven University of Technology. Eindhoven, NL.

Villegas, P., Munoz, M. A., & Bonachela, J. A. (2020). Evolution in the Debian GNU/Linux software network: Analogies and differences with gene regulatory networks. *Journal of The Royal Society Interface*, *17*(163), 20190845. https://doi.org/10.1098/rsif.2019.0845

Walden, J. (2020). The impact of a major security event on an open source project: The case of OpenSSL. *17th International Conference on Mining Software Repositories*. https://doi.org/10.1145/3379597.3387465

Wang, L., Li, R., Zhu, J., Bai, G., & Wang, H. (2021). A large-scale empirical study of COVID-19 themed GitHub repositories. *2021 IEEE 45th Annual Computers, Soft-*

ware, and Applications Conference (COMPSAC), 914–923. https://doi.org/10.1109/COMPSAC51774.2021.00124

Warncke-Wang, M., Ranjan, V., Terveen, L., & Hecht, B. (2015). Misalignment between supply and demand of quality content in peer production communities. *Proceedings of the Ninth International AAAI Conference on Web and Social Media*, 493–502.

Wasserman, S., & Faust, K. (1994). *Social Network Analysis: Methods and Applications*. Cambridge University Press. https://doi.org/10.1017/CBO9780511815478

Wheeler, D. A. (2014a). How to prevent the next Heartbleed. Retrieved November 11, 2021, from https://dwheeler.com/essays/heartbleed.html

Wheeler, D. A. (2014b). Preventing Heartbleed. *Computer*, *47*(8), 80–83. https://doi.org/10.1109/MC.2014.217

Wiggins, A., Howison, J., & Crowston, K. (2009). Heartbeat: Measuring active user base and potential user interest in FLOSS projects. In C. Boldyreff, K. Crowston, B. Lundell, & A. I. Wasserman (Eds.), *Open Source Ecosystems: Diverse Communities Interacting* (pp. 94–104). Springer Berlin Heidelberg. Retrieved October 8, 2018, from http://link.springer.com/10.1007/978-3-642-02032-2_10

Wilfahrt, M., & Michelitch, K. (2022). Improving open-source information on African politics, one student at a time. *PS: Political Science & Politics*, *55*(2), 450–455. https://doi.org/10.1017/S1049096521001219

Willer, R. (2009). A status theory of collective action. *Advances in Group Processes* (pp. 133–163). Emerald Group Publishing.

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. *Proceedings of EASE '14: The 18th International Conference on Evaluation and Assessment in Software Engineering*. https://doi.org/10.1145/2601248.2601268

Wong, W. E., Qi, Y., & Cooper, K. (2005). Source code-based software risk assessing. *Proceedings of the 2005 ACM symposium on Applied computing - SAC '05*, 1485. https://doi.org/10.1145/1066677.1067014

Wu, C.-G., Gerlach, J. H., & Young, C. E. (2007). An empirical analysis of open source software developers' motivations and continuance intentions. *Information & Management*, *44*(3), 253–262. https://doi.org/10.1016/j.im.2006.12.006

Xing, J., & Vetter, M. (2020). Editing for equity: Understanding instructor motivations for integrating cross-disciplinary Wikipedia assignments. *First Monday*. https://doi.org/10.5210/fm.v25i6.10575

Xu, B., & Li, D. (2015). An empirical study of the motivations for content contribution and community participation in Wikipedia. *Information & management*, *52*(3), 275–286.

Yang, H.-L., & Lai, C.-Y. (2010). Motivations of Wikipedia content contributors. *Computers in Human Behavior*, *26*(6), 1377–1383. https://doi.org/10.1016/j.chb.2010.04.011

Yang, N., Ferreira, I., Serebrenik, A., & Adams, B. (2022). Why do projects join the Apache Software Foundation? *Proceedings of the 2022 ACM/IEEE 44th International Conference on Software Engineering: Software Engineering in Society*, 161–171. https://doi.org/10.1145/3510458.3513006

Yin, L., Chakraborti, M., Yan, Y., Schweik, C., Frey, S., & Filkov, V. (2022). Open source software sustainability: Combining institutional analysis and socio-technical networks. *Proceedings of the ACM on Human-Computer Interaction*, *6*(CSCW2), 1–23. https://doi.org/10.1145/3555129

Yin, L., Chen, Z., Xuan, Q., & Filkov, V. (2021). Sustainability forecasting for Apache incubator projects. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1056–1067. https://doi.org/10.1145/3468264.3468563

Yin, X., & Zajac, E. J. (2004). The strategy/governance structure fit relationship: Theory and evidence in franchising arrangements. *Strategic Management Journal*, *25*(4), 365–383. https://doi.org/10.1002/smj.389

Zampetti, F., Serebrenik, A., & Di Penta, M. (2020). Automatically learning patterns for self-admitted technical debt removal. *2020 IEEE 27th International Conference on*

Software Analysis, Evolution and Reengineering (SANER), 355–366. https://doi.org/10.1109/SANER48275.2020.9054868

Zentall, S. S., & Beike, S. M. (2012). Achievement and social goals of younger and older elementary students: Response to academic and social failure. *Learning Disability Quarterly*, *35*(1), 39–53. https://doi.org/10.1177/0731948711429009

Zerouali, A., Mens, T., Robles, G., & Gonzalez-Barahona, J. M. (2019a). On the diversity of software package popularity metrics: An empirical study of npm. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 589–593. https://doi.org/10.1109/SANER.2019.8667997

Zerouali, A., Mens, T., Robles, G., & Gonzalez-Barahona, J. M. (2019b). On the relation between outdated Docker containers, severity vulnerabilities, and bugs. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 491–501. https://doi.org/10.1109/SANER.2019.8668013

Zhang, Y., Wang, H., Wu, Y., Hu, D., & Wang, T. (2020). GitHub's milestone tool: A mixed-methods analysis on its use. *Journal of Software: Evolution and Process*, *32*(4), e2229. https://doi.org/10.1002/smr.2229

Appendix A

METHODS NOTES FROM CHAPTER 3

This appendix was published as the online supplement to the paper reproduced as Chapter 3. I have made minor typographical and formatting adjustments.

Using an Alternate Measure of Importance

In this appendix, we illustrate our approach using an alternate measure of importance. In §3.3, we described underproduction measurement in FLOSS as a five-step method that includes the identification of a measure of importance. The method we describe is not dependent on a specific choice of measure; however, in our illustration of the measure in §3.5.3 we offer an example of *installation count* as reported by the Debian project's Popcon tool.

Installation count is not the only possible measure of importance; in fact, it is not even the only measure of importance that can be derived from Popcon. Popcon also reports a measure called *vote*. Popcon documentation explains that "A computer 'vote's for a package if according to the data provided in the report, a program provided or depending on the package was used less than thirty days ago."[1] However, one limitation of this measure is that the calculation of vote uses a comparison of a file's *atime* (access time) and *ctime* (creation time).[2]

A file's atime updates when the file is read or run, including not only direct usage, but also interactions occurring as part of some other process, e.g. some system backups or updates of the locate database. In addition, files or filesystems can be set not to update atime. Libraries acting as compile-time dependencies would not register updates to their

[1] https://popcon.debian.org/FAQ (Archived: https://perma.cc/2VXC-47YD)

[2] https://popcon.debian.org/README (Archived: https://perma.cc/4S3D-H2RU)

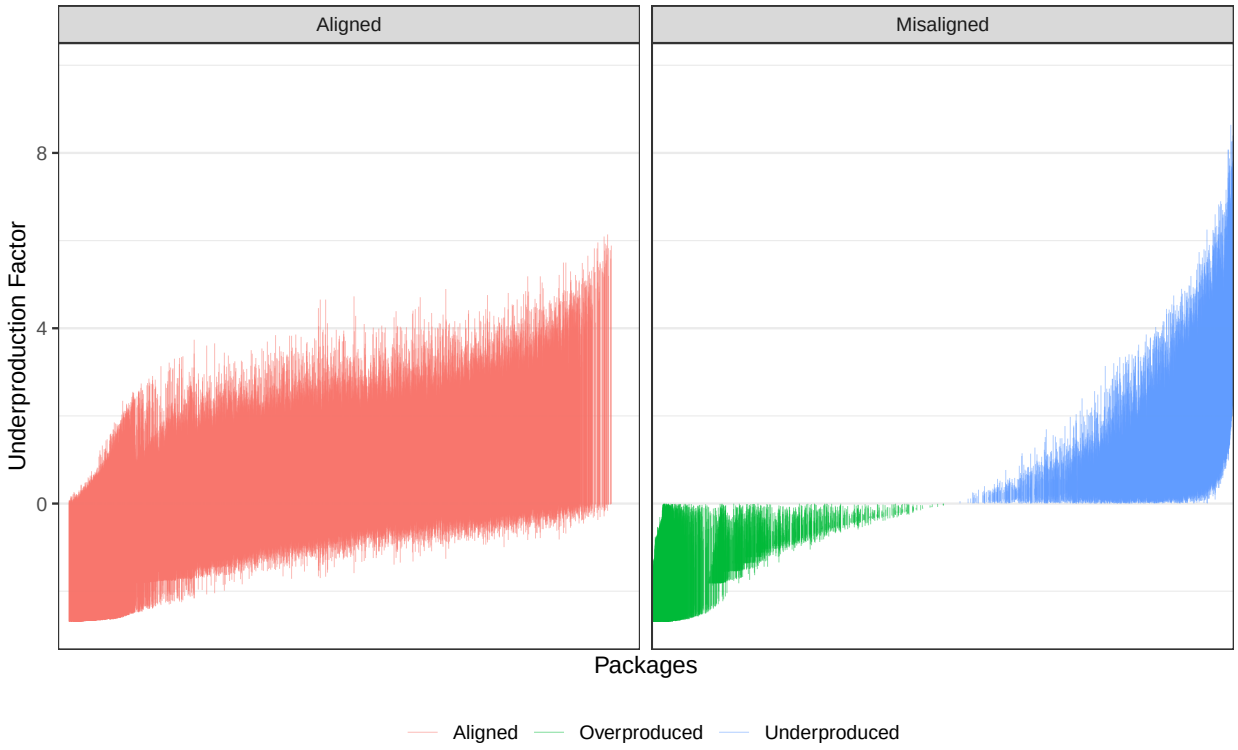


Figure A.1: Credible intervals for U_j for every package in Debian. We treat all packages whose CIs include zero as aligned; those whose CIs are entirely above 0 are labeled 'underproduced;' those whose CIs are entirely below zero are labeled 'overproduced.'

atime when the application depending upon them is run. Hence although the intention is for vote to indicate usage, it has the disadvantage that it relies on atime—and atime updates do not always indicate usage, and a lack of atime updates likewise do not always indicate non-usage.

That said, the 30 day observation span mitigates and Debian's dependency analysis although limited to dependencies of which Debian is aware both serve to mitigate this limitation: vote as a measure is widely used in the Debian community alongside installation and offers another measure of package importance.

To illustrate the results of applying our method using an alternate measure of quality, we followed the same process described in §3.5, substituting *vote* for *inst* in §3.5.3 as appropriate. The credible interval plot in Figure A.1 can therefore be compared with Figure 3.3. The heatmap in Figure A.3 can be compared with the heatmap in Figure 3.4 and the boxplot in Figure A.2 to the boxplots in Figure 3.5.

We observe that the results of this analysis are similar—underproduction by this measure is likewise widespread, and in fact, in Figure A.2, the same 11 packages appear at the top (i.e., worst underproduction problem) in both lists. However, the ordering beyond the 11th most badly underproduced is slightly shuffled. We also observe that the package "glibc-doc-reference" and "xfonts-scalable-nonfree" drop out of the 30-worst list when usage ("vote") is the measure of importance instead of installation, and "xserver-xorg-video-vesa" and "gpm" are added.

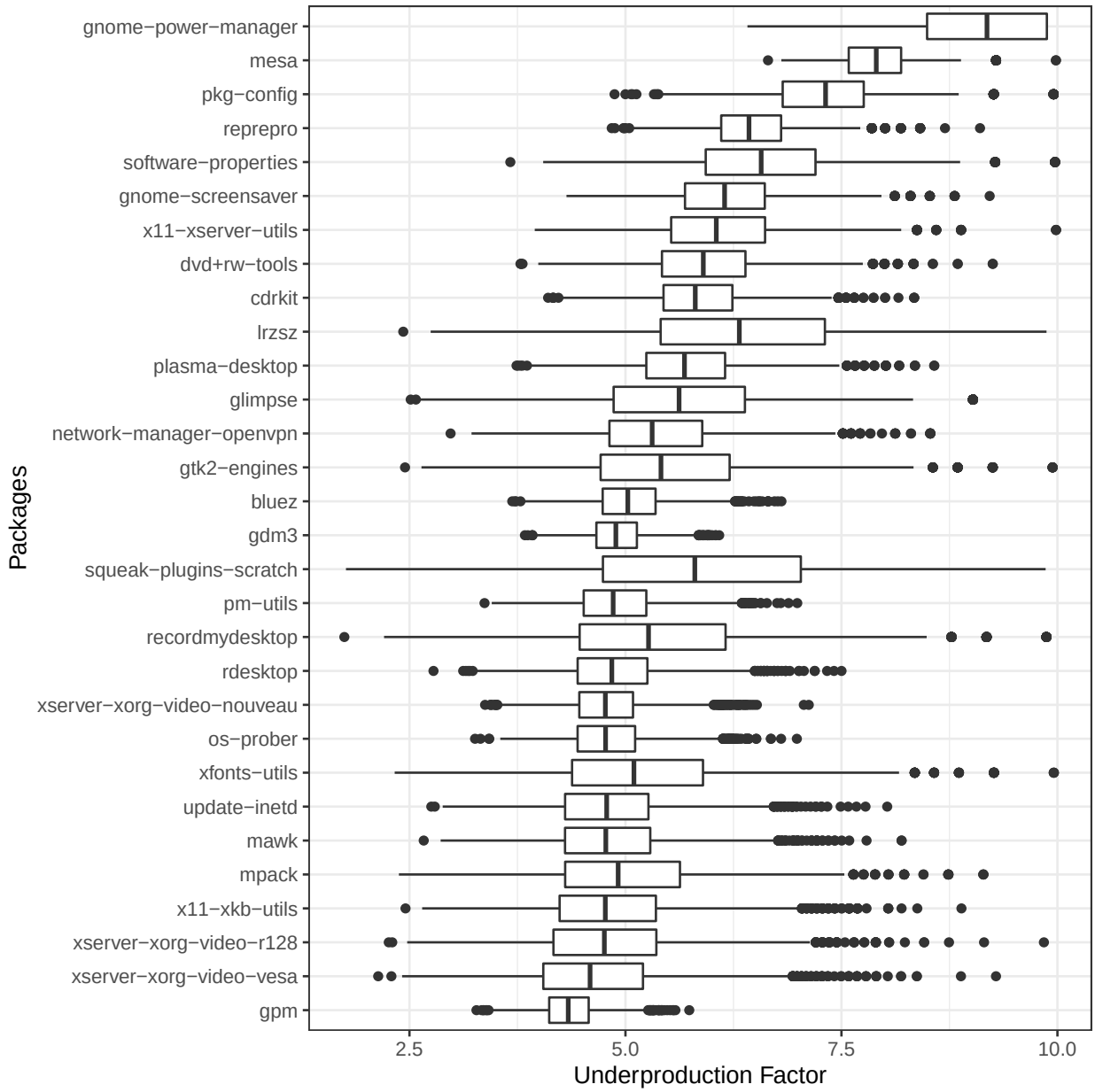


Figure A.2: Packages displaying the highest mean levels of underproduction using "vote" as a measure of importance. Boxplots show the mean and interquartile range of our distributions of U_j and reflect uncertainty in our model of package-level quality.

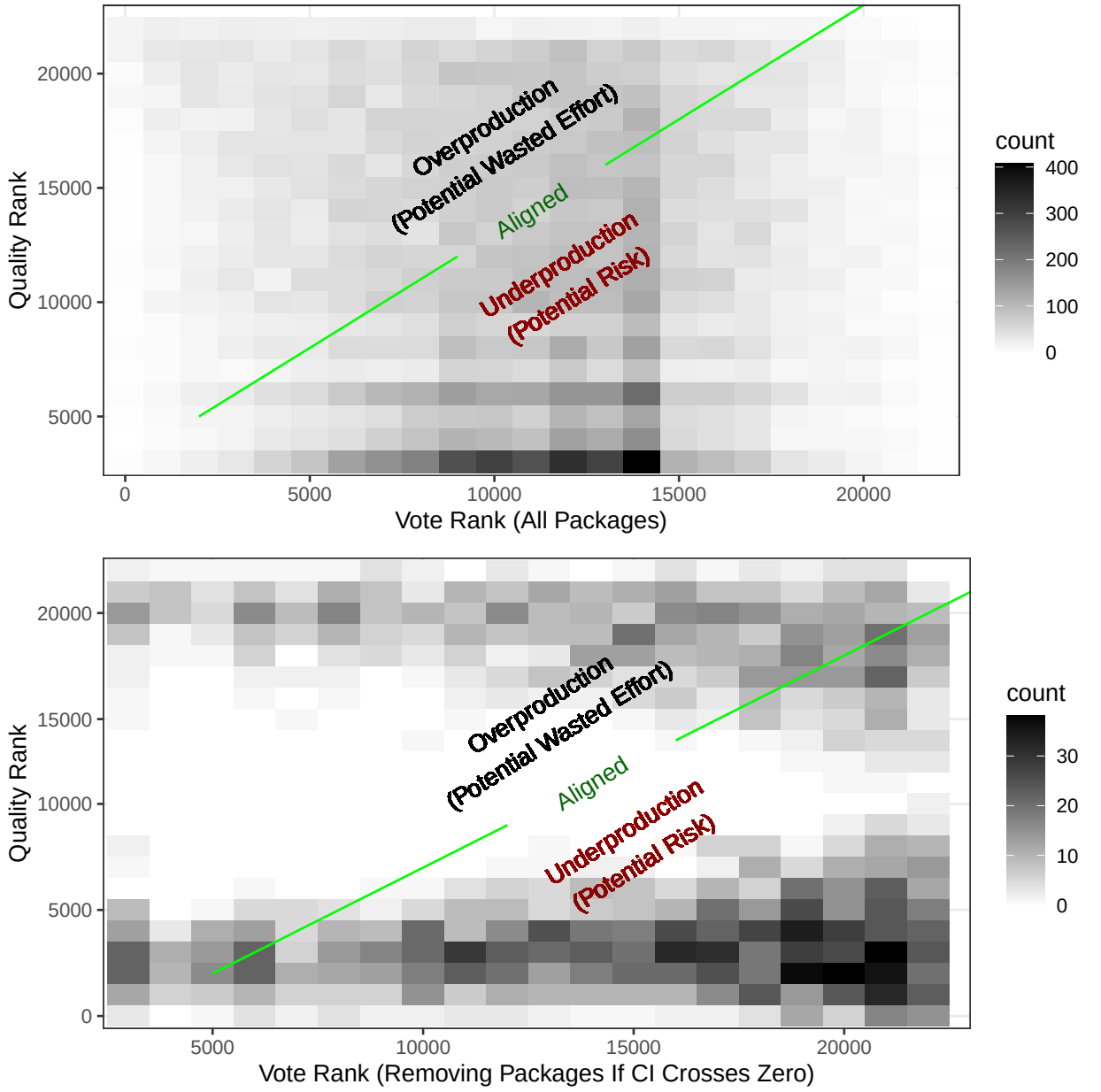


Figure A.3: A heatmap of software alignment. Color intensity indicates the number of packages occupying a given ranking of quality and importance ("vote"). Aligned packages appear along the lower-left to upper-right diagonal. The top heatmap includes all packages, while the bottom heatmap contains only those packages for which the 95% credible interval does not cross zero.

Appendix B

UNDERPRODUCTION CASE STUDIES

This collection of case studies helped me to develop insight into how we might understand underproduction longitudinally, as a follow-on to the work in Chapter 4 and in preparation for the work of Chapter 5. For each case, I used a combination of forensic qualitative analysis (Champion et al., 2019) and quantitative description to construct a narrative account of how quality and importance changed over time.

B.1 Kazam: The Worst Case Scenario

For this case study, I looked at *kazam*—the most underproduced package of the set of all those written in Python, pulled from my datasets developed in Chapter 3 and Chapter 4. Kazam is a screen capture and recording library, and the situation I found seems to be indeed a "worst case scenario." The project was developed steadily at first, and I observed multiple releases while the package was in Debian's testing/pre-release phase, however work ceased about a year after it was adopted into Debian for distribution. Although it might be called simply a case of bad luck due to timing, I observe that the package persisted in the Debian distribution for many years after upstream development ceased.

In the timeline of major events in Figure B.1, the first release occurs in early 2012, Debian's release is in mid-2013, and there were no new releases from the maintainer after mid-2014. Although a within-Debian patch release coincided with a small flurry of activity in 2019, the quality as measured by the pylint tool declined.

One question I had based on these observations is how we might characterize the extent to which projects reach a point of "completion," go into "maintenance-only" mode, or are "abandoned." These states might even overlap. For example, the author might feel essentially

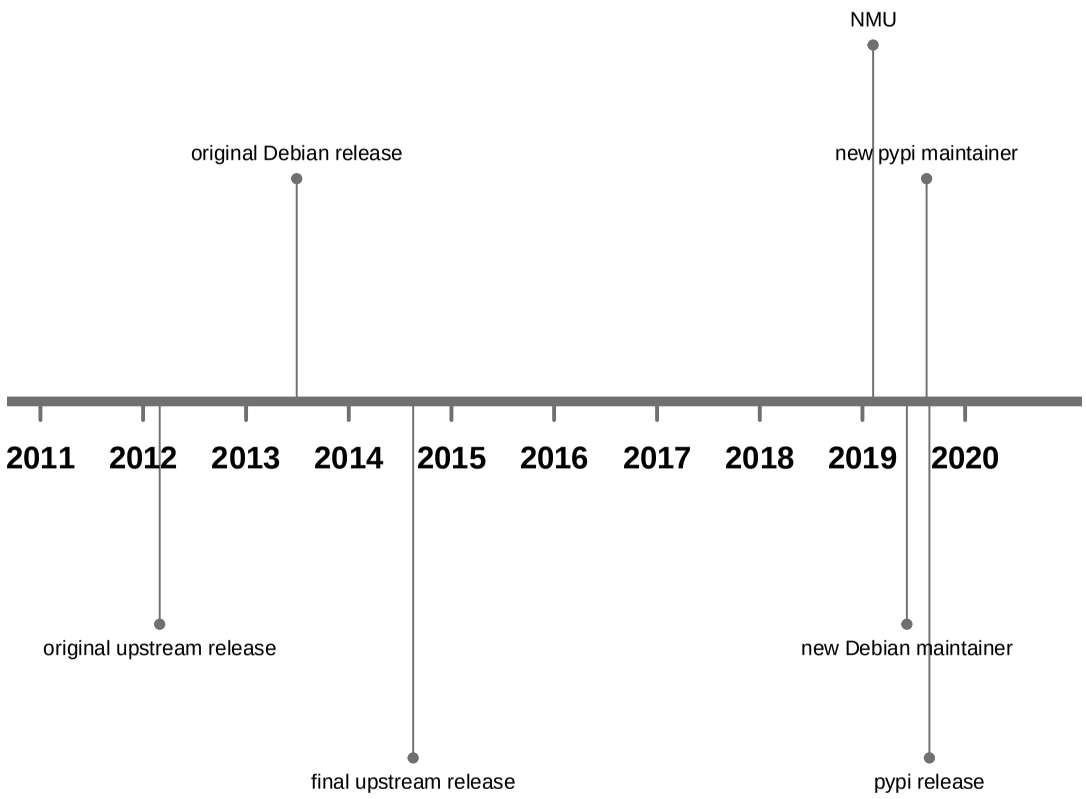


Figure B.1: Major events in the life of kazam

done with a project and be willing to make updates if the need arises—but then perhaps never quite getting around to it, despite the best of intentions. Or the author might feel satisfied with what they've done, but users might have unmet needs that cause them to think of the project as abandoned. External circumstances might impose new time constraints on the maintainer, or dissent within the project might bring work to a standstill.

Even if these uncertainties about developers' commitment to a project can be resolved, there is no general protocol for articulating the maintenance status of a given package. However, we can observe several practices in open source that seek to address this issue. Umbrella organizations for open source software such as the Debian Project and the Apache Foundation have documented policies to articulate support and end of life.[1] GitHub allows a maintainer to declare a project "archived."[2] *End of Life* is an independent effort to catalog end-of-life data and support lifecycles.[3]

Abandonment is not clear cut. Analytically, distinguishing an extended time period between updates from actual abandonment requires imposing a rate of production that is not entirely consistent with a project based on self-selection of tasks, even one with a large user base. Developers may feel they are done with the project but happy to see others use it and leave the task of maintaining it to them, or they may intend to resume work in the future. That said, project abandonment by any definition seems to be a widespread challenge. Security firm Sontatype published a report in October 2023 stating that only 11% of the 1.1 million projects in four major software ecosystems they analyzed (JavaScript NPM, Java Maven, Python PyPI/pip, .NET NuGet) qualified as "maintained," which they measured as having had "regular code commits or timely responses to issues" in the previous 90 days (Sonatype, 2023).

---

[1] e.g., Debian's staging of a given release as being supported first by the security and release teams, then by the long term support team, then declared end of life after a five-year span https://wiki.debian.org/LTS/, and Apache's Attic designation, which declares criteria for projects being end of life https://attic.apache.org/

[2] https://docs.github.com/en/repositories/archiving-a-github-repository/archiving-repositories

[3] https://endoflife.date/

In their analysis, Avelino et al. (2019) define abandonment at the individual level (rather than the project level), defining an individual as having abandoned the project if their last commit (code contribution) occurred more than one year before the most recent commit by any contributor, and consider a project to be inactive if all high-volume contributors have abandoned the project. Avelino et al. (2019) examined 1,932 of the most "starred"[4] repositories for common languages on GitHub and found that 16% of these highly popular projects had lost all of their key developers (in 66% of these cases, there was only one key developer).

In the case of kazam, I observe from Figure B.2 an 8-year span (2014-2021) in which usage of the package was climbing steadily within Debian despite the lack of updates. Perhaps the software was truly complete and fulfilling its role in a solid manner, or perhaps there was a lack of suitable alternatives, or perhaps the potential risks associated with the package were not visible. Given changes in computer architecture, languages, and environments that occur over time periods of this size, we might suppose there were at least some needs going unfulfilled.

We can in this case do more than speculate, since the reason I chose it for my case study was that it was the most extreme example of underproduction among the Python-language projects I assessed. Underproduction is based on a comparison of two measures: importance (operationalized as installation rank) and quality (operationalized as ranked mean time to resolution of bugs). Therefore we can conclude that bugs in kazam were much longer-lived than we would expect given the importance of the package, suggesting that at least from the perspective of people trying to use the package and the experience of the Debian project trying to distribute it, the software had problems.

This case study illustrates several of the challenges of information public goods. First, we see that the peer production of these goods functions on self-selection into tasks: someone chose to write kazam, someone chose to be the packager for kazam, and many people trusted

[4]Starring a repository is akin to favoriting or bookmarking; functionally, it means a GitHub user will be receive a feed of updates about the project; starring is a common measure of general interest or popularity.

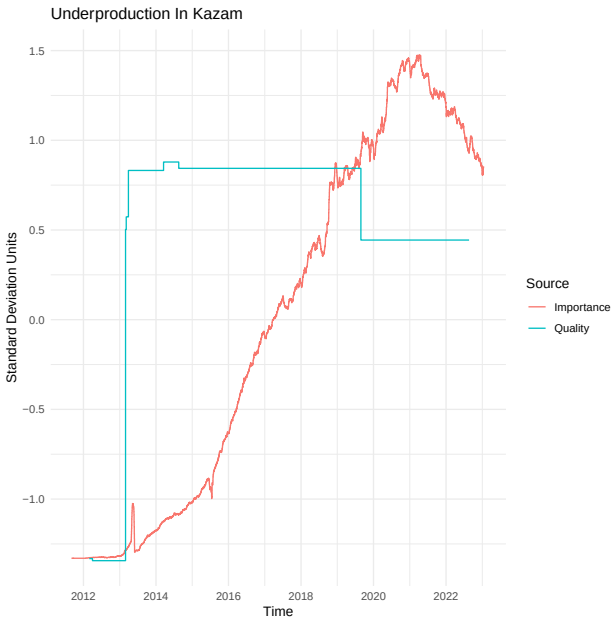


Figure B.2: Quality and Importance of kazam; SD units

Debian as a curator and chose to adopt the package from the distribution. Second, the degree of communication between the developer, Debian, and users is not visible in this analysis, but it seems likely that each decisionmaker has limited information about the priorities and preferences of the others, and limited (if any) obligation. The growing popularity of kazam could have or perhaps should have spurred improvement of the software, but it did not; in this case, no regulating force served to revive production activity to meet the importance, and low quality did not drag consumption down. That said, and although this data is not causal, we do see a fall in popularity late in the package's history, suggesting that eventually these misalignments do diminish.

B.2 Pip: The Median Case

For this case study, I looked at *pip*, chosen because had the median value for underproduction among the Python language packages in my dataset from Chapter 3 and Chapter 4. Pip is a package distribution mechanism for libraries used in the Python programming

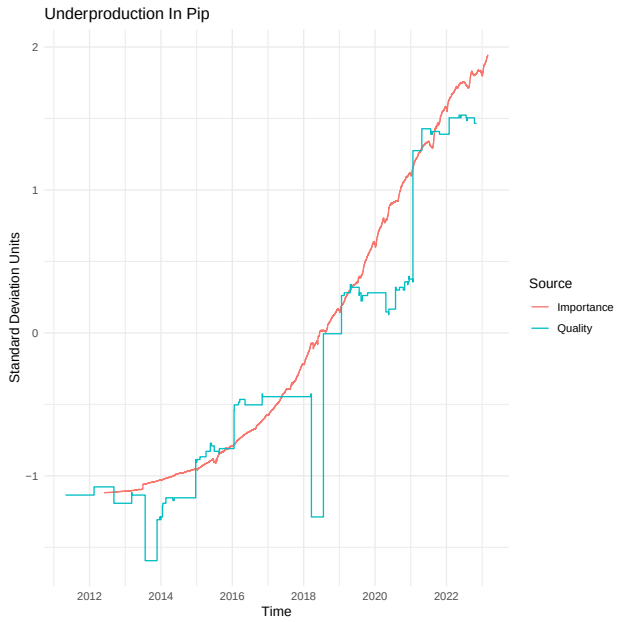


Figure B.3: Quality and Importance of Pip; SD units

language and draws on packages uploaded into the Python Package Index (PyPI), a package repository. One very common route to installing a new Python library is to type `pip install` *`packagename`*. Pip and PyPI are a sometimes-invisible piece of key infrastructure, and are maintained by the Python Software Foundation.[5] suggest that this package is very actively maintained, and package repository notes offer more details than it makes sense to try to capture on a timeline of qualitative events (the current release is version 24.0, with each major release, e.g., 23, 22, etc. associated with minor releases, e.g. 22.2, and patch or bug-fix releases, e.g. 20.2.3).[6]

By these measures and from this investigation, pip seems to be doing well and the graph

[5]See https://en.wikipedia.org/wiki/Pip_(package_manager).

One thing I notice immediately from the quantitative timeline in Figure B.3 is how closely the standard deviations of both quality and importance seem to track: pip is growing in importance, and quality is tending to also increase—sometimes quality steps ahead of importance growth, and sometimes quality falls below importance growth, but generally the trends track one another. Notes about sponsor-funded overhauls https://pypi.org/sponsors/

[6]See pip release news, https://pip.pypa.io/en/stable/news/

in Figure B.3 corresponds to what we would anticipate from an aligned package.

B.3 SEAMicro Client: The Most Overproduced

SEAMicro client is a python client "for consuming SeaMicro REST API v2.0"—that is, a way to manage compute farms from the SeaMicro company (for example, powering up nodes). My results in Chapter 3 found SEAMicro Client to be overproduced, that is, better quality than we might anticipate given its relatively low importance. The quantitative trends in Figure B.4 show a relatively flat quality trend with a rather chunky importance trendline. I investigated the source of the chunky appearance and found that this is because during the study period, SEAMicro client had only a handful of users, therefore it was quite easy for its usage to double or treble. Thus the finding of overproduction is not surprising—for a package to enter our dataset at all, it has to have at least some level of quality, but with survey responses indicating almost no importance, it seems inevitable numerically that such a package would land in the overproduced part of the dataset.

This case offers an example of why I describe overproduction as not particularly harmful except to the extent it represents wasted effort; it seems harmless to have an excellent package used by few people. Waste is, however, contextual, because it depends on assumptions about what might be done differently. Perhaps the people maintaining SEAMicro Client could be persuaded to do different work, serving more people and allowing the quality of the overproduced package to slip, but then again perhaps the maintainers are so devoted that redirection is not practical. This finding may also represent a limitation of the measurement I have used—it may be the case that the primary use case for this particular package is one where users do not opt in to surveys about usage, i.e. the lack of survey responses represents data missing not at random. To the extent that this package only appears to be overproduced due to a lack of communication about its importance, this package could actually be at risk. This work also suggests that I need to be cognizant of the implications introduced by choices of standardization approaches if I want to compare among packages: note that the y-axis range for Figure B.2 is (-1.25, 1.5) and Figure B.3 has a range of (-1.5, 2), but the range

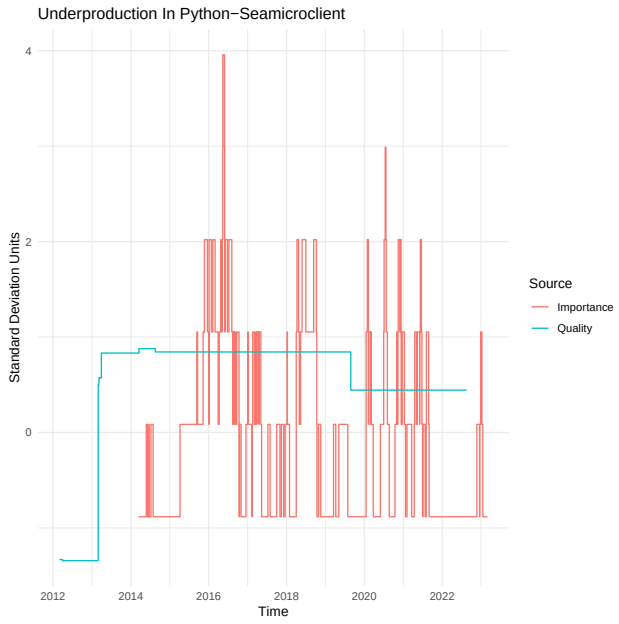


Figure B.4: Quality and Importance of Python-Seamicroclient; SD units.

for Figure B.4 is (-1, 4). Approaches to strip units out of measure have trade-offs: if we use standardization, dividing the de-meaned value by the standard deviation, our range is now subject to the spread of the data. Some packages may have highly extreme values for quality or importance while others are more clustered closer to the mean, or within a given package, one measure may be more widely spread while the other is more clustered. The cross-sectional approach in Chapter 3, by contrast, uses a rank-ordering approach which supports cross-package comparison.

B.4 Looking Across Cases

These cases suggest that examining the temporal trends for each package, in particular the relative shapes of the importance trajectory and the quality trajectory, will offer sufficient variation and insight into the question of underproduction. In developing an extension to the underproduction analysis method described in Chapter 3, these case studies point out the need to address several issues in developing a longitudinal version of the method.

First, as of yet these case studies offer no way to quantify underproduction; the use of standardized units was instructive but makes comparisons between the curves or across packages more difficult. The original underproduction method takes up a position that the highest demand should coincide with the highest quality, but the highest quality for a given package might be 1 standard deviation from the mean in a low-variance package and 3 standard deviations from the mean in a high-variance package. Therefore this suggests the adoption of a measurement transformation approach that makes these measures more comparable while still following the general high demand - high quality normative position.

Second, as with the method described in Chapter 3, there is no avoiding that declaration of a normative position, including the establishment of what we can argue is a reasonable indicator that conditions deviate from the preferred state.

Appendix C

COUNTERING UNDERPRODUCTION OF PEER PRODUCED GOODS

In this chapter, I describe an empirical study of underproduction in the context of Wikipedia. This is the context where underproduction in commons-based peer production was first examined, including work from Warncke-Wang et al. (2015) and Gorbatai (2011b). My key finding in this chapter is that more experienced contributors—including those who contribute without an account—tend to contribute to underproduced goods and that their efforts shift toward underproduced goods over time. This has important implications for underproduction in free/libre open source software as well because it suggests the value of retaining contributors, and serves as an important consideration alongside interventions into commons-based peer production that emphasize governance (Gaughan et al., 2024), community social health (Qiu, Lieb, et al., 2023), and newcomer engagement (Steinmacher et al., 2015).

This chapter was previously published as: Champion, Kaylea and Hill, Benjamin Mako (2024) "Countering Underproduction of Peer Produced Goods" *New Media and Society*. I led all parts of this project and have made minor typographical and formatting adjustments.

C.1 Introduction

Peer production is a collaborative technology-assisted process in which participants select and perform tasks in a self-directed way, then combine their work with others (Benkler, 2006). Peer production plays a critical role in today's information ecosystem. From web servers to programming languages, Internet infrastructure is largely peer produced (Eghbal,

2016). Much of the content we consume online is also peer produced. Peer produced websites such as Wikipedia, Reddit, and Fandom are among the most visited sites on the Internet.[1] Peer produced content includes map data, search results, digital assistants responses, AI training data, and more (McMahon et al., 2017).

How do producers' choices of what to make correspond to what consumers want to use? In a market-driven system, supply and demand are aligned via price. But in commons-based peer production, no such signal is available. This can lead to *underproduction*: the production of high-interest but low-quality goods (Warncke-Wang et al., 2015).

This project seeks to understand the association between contributor experience, account use, and task selection. We first examine what is known about how tasks are selected to provide the rationale for several hypotheses. We then describe the setting, data, measures, and analytical plan. Next, we share empirical results. Finally, we discuss the limitations and implications of our results before concluding.

C.2 Background

C.2.1 Commons-based Peer Production and Underproduction

Commons-based peer production is a term coined by Yochai Benkler to describe an emerging form of cooperative production made possible by new communication technology. Tasks are self-selected and combined through technology, resulting in information public goods. Examples of peer production include free/libre open source software (FLOSS) projects like GNU/Linux and Apache, Wikipedia, and OpenStreetMap (OSM). Benkler (2006) argues that because individuals know their interests, knowledge, availability, and skills, peer production creates the potential for diminished separation between producers and consumers and that self-assignment to tasks supports efficient matching between contributors and tasks.

One way to assess the success of a peer production project is to examine how well it meets the needs of its users. Warncke-Wang et al. (2015) found that over 40% of views to

[1] https://perma.cc/K5VM-V99E

English Wikipedia were to articles that were lower quality than one might expect given their popularity. Countries, religions, LGBT topics, psychology, pop and rock music, the Internet and technology, comedy, and science fiction were found to be disproportionately affected. The misalignment of quality and public interest in these topics is concerning because of their relevance to public affairs and modern culture.

To quantify misalignment, Warncke-Wang et al. (2015) propose that if a system is aligned, goods in the highest demand should be the highest quality while those in the lowest demand should be the lowest quality. When quality is low relative to demand, goods are described as *underproduced*. When quality is high relative to demand, they are *overproduced*. Although overproduction may not be harmful beyond the potential for wasted effort, underproduction can impact public knowledge goods and digital infrastructure (Champion & Hill, 2021; Eghbal, 2016).

C.2.2 Motivation, Experience, and Task Selection

To understand the sources of underproduction, we must consider why people participate in peer production. Studies have observed a range of motivations. Some are motivated to create public goods (Budhathoki & Haythornthwaite, 2013), to address social issues (March & Dasgupta, 2020), to help other community members (Wu et al., 2007), to enhance their own use of a public good (Krishnamurthy et al., 2014; Meng & Wu, 2013), to enhance their reputation or qualifications (Oreg & Nov, 2008; Silva et al., 2020; Xu & Li, 2015), as an experience of self-efficacy (H.-L. Yang & Lai, 2010), as a professional responsibility (Farič & Potts, 2014), for personal enjoyment and learning (Farič & Potts, 2014), out of reciprocity (Xu & Li, 2015), for a class assignment (Coelho, Valente, Silva, & Hora, 2018; Konieczny, 2012; McDowell & Vetter, 2022; Xing & Vetter, 2020), or as part of their job (Germonprez et al., 2019). The results of this body of literature suggest that contributor motivation in peer production tends to be complex and multidimensional and primarily, although not exclusively, intrinsic (Belenzon & Schankerman, 2015; Benkler et al., 2015; Kuznetsov, 2006; Rafaeli & Ariel, 2008).

To the extent that peer production contributors do so because they are told to (e.g., by an employer or instructor in a class), extrinsic rewards and others' interests may be primary motivators. For example, contributing to Wikipedia is an increasingly common assignment in college classes where instructors are driven by concerns about content gaps or their commitment to public goods (Konieczny, 2012; McDowell & Vetter, 2022; Xing & Vetter, 2020), or by a desire to have students engage with their coursework in public (Gallagher et al., 2019). Firms may encourage their employees to contribute to FLOSS to enhance the firm's use of the public good or to build their reputation (Germonprez et al., 2019).

Nonprofit organizations also encourage contributions to peer production projects as part of their missions. Examples include the Missing Maps project by the Red Cross and Doctors Without Borders encouraging contributions to OSM (Herfort et al., 2023), partnerships encouraging contributions to FLOSS projects targeting climate change,[2] and galleries, libraries and museums partnering with Wikipedia (Karczewska, 2023). Some non-profits seek to counter content and participation gaps in peer production. For example, WikiEdu supports instructors using Wikipedia in their course assignments to close content gaps (Wilfahrt & Michelitch, 2022). The Outreachy initiative aims to increase diversity in open source (Ackermann, 2023).

Within this complex motivational landscape, contributor task selection varies across users and within users over time. In their examination of OSM contributors, Budhathoki and Haythornthwaite (2013) reported that high-volume contributors were likelier to seek community recognition, while low-volume mappers reported wanting to contribute to a free and open project. In contrast, Shah (2006) showed that FLOSS contributors describe their first contributions as directly related to personal skills, needs, and priorities, while longer-term participants report working for the good of the project. Likewise, Bryant et al. (2005) found that while Wikipedia editors reported participating within their areas of expertise initially,

[2]e.g., Climage Triage https://perma.cc/9R8F-QQYY

they sought to build the Wikipedia community and serve the public good over time. Restivo and van de Rijt (2014) found that the highest-volume contributors produce more after receiving informal social rewards, while lower-volume contributors did not and may even respond negatively to the same awards.

Previous work offers some additional clues to the relationship between experience and choosing to contribute to underproduced goods. New contributors initially select tasks that match their immediate needs, areas of knowledge, and skills (Bryant et al., 2005; Preece & Shneiderman, 2009). Because people are interested in similar things, it seems likely that these areas of knowledge will, on average, be associated with high-interest subjects. On the other hand, contributors faced with high-quality artifacts may not see where their assistance is needed (Bryant et al., 2005; Preece & Shneiderman, 2009). However, as they accumulate experience, we expect contributors' skills to increase.

Although this literature points in conflicting directions, prior work has emphasized that skilled participants incorporate the public's desire for information as a component of their task selection. As a result, we expect to find that the most experienced contributors are more likely to engage in the improvement of underproduced goods: *H1: individuals with less experience will contribute to less underproduced goods than individuals with more experience.*

In that experienced contributors may seek recognition for their work, social factors may act to disrupt the trend we describe in H1 (Oreg & Nov, 2008). Once a participant is involved in a project, their ongoing participation may be encouraged by a desire for status (Willer, 2009). To distinguish the extent to which social rewards drive task selection, we take advantage of the fact that receiving in-group social rewards partially depends on identifiability. Collaborators may seek to direct their responses to an attributed person. Previous research suggests that creating an account may be driven by a desire to obtain feedback or recognition (Forte et al., 2017). By contrast, Belenzon and Schankerman (2015) argued that anonymous contributors to FLOSS projects were more motivated by a desire to contribute to the public interests, as measured by the fact that they were more likely to contribute to projects operated by a nonprofit than a for-profit entity, and those serving end users (rather than other

developers). Because contributing without an account makes social rewards more difficult, we propose that *H2: contributing without an account is associated with more underproduced goods than contributing with an account.*

Finally, we consider two competing explanations for H1 and H2. An association between task selection and experience could be explained by contributor attrition (i.e., those who go on to make many contributions have always differed from those who make only a few) or by shifting motivations (i.e., persistence causes changes in task selection). In other words, if we find support for H1 and H2, is it due to differences caused by users being "born" or "made"? Support for the "born" explanation comes from past work that has found that individuals who go on to contribute at high volume seem to do so from the start (e.g., Panciera et al., 2009). On the other hand, contributors' motivation appears to shift over time (e.g., Bryant et al., 2005; Shah, 2006). Moreover, peer production projects have successfully invested in efforts to socialize and retain newcomers (e.g., Morgan & Halfaker, 2018; Tan et al., 2020), suggesting that contributors can also be "made", at least to some extent. Given this evidence of motivational shifts, we hypothesize within-person change as *H3A: an individual will shift toward underproduced goods as they accumulate experience*, and *H3B: an individual not using an account will shift toward underproduced goods at lower experience levels than one using an account.* Our reasoning for H3B is that because non-account-users are less likely to receive social rewards, they are even more likely to be motivated by the public interest.

C.3 Research Design

C.3.1 Empirical Setting

This study is conducted in the context of Wikipedia, the Wikimedia Foundation (WMF) website. Wikipedia is one of the most popular websites in the world and is used as reference material, a fact-checking source, and an input to machine learning and AI systems (McMahon et al., 2017). Wikipedia projects in 326 different language editions received 29 billion page

views in January 2024.[3] The largest Wikipedia language edition, English, has more than 6.7 million articles. With some exceptions, clicking on an 'Edit' tab on every Wikipedia page creates an interface for revising the existing text. For most language editions, changes are immediately visible without review.

C.3.2 Data

Our unit of analysis is the *revision*, and we use the complete revision history of Wikipedia, made available by the WMF, from its inception in January 2001 through July 2021. The complete revision history of Wikipedia contains a wide range of content, including discussion and personal pages.

Our hypotheses concern the relationship between experience and having an account with article underproduction, a function of viewership and quality. To test our hypotheses, we filtered the revision population in multiple ways before drawing our sample. We excluded work done by bots, revisions to non-article pages, vandalism and its removal, and purely administrative revisions. Because no viewership data are available before December 2007, we only considered contributions after December 2007. Details on these filters and the total revisions after each step are reported in our online supplement.

In Wikipedia, a small fraction of contributors make a large portion of all contributions. To capture variation based on editor experience, we drew two samples. For the *revision sample* used to test H1 and H2, we first stratified contributions by their ordinal position into increasingly large "buckets" increasing in size by 2^x (i.e., 1, 2, 4, 8, and so on). We then drew equal-sized random samples from each bucket, oversampling where the data was thinner, for a total of 192,672 revisions. The inverse of the resulting sampling proportions was used as weights during subsequent calculations (Tracy & Carkin, 2014).

The second sample is a *within-person sample*, designed to extract the contribution history of editors who differed in their ultimate contribution level. To build this sample, we stratified

[3]WMF Statistics: https://perma.cc/H3NT-RP9N, Wikipedia page about Wikipedia: https://perma.cc/RNC3-4SLN

editors based on their total contributions through July 2021, again by creating increasingly large (2^x) buckets. We then drew a random sample of editors from each. Again, we oversampled where data was thinner and retained proportions as weights. After selecting editors, we gathered all article contributions made by the selected editors for a total of 42,602,912 revisions. This approach gives us the contribution history of a diverse range of editors: those who went on to contribute a great deal and those who only contributed once or twice.

We have made our data and code available at https://doi.org/10.7910/DVN/UDQT6E.

C.3.3 Measures

Our dependent variable is *underproduction factor*. To construct this measure, we drew from elements of both Champion and Hill (2021) and Warncke-Wang et al. (2015) and calculated the negative log of the ratio between quality rank and popularity rank. Both ranks were assigned such that high quality and high popularity receive high ranks, while low quality and low popularity receive low ones. A perfectly aligned article—one with the same popularity and quality rank—would have an underproduction factor of 0 since log(1) is 0. An overproduced article of high quality and low popularity would have a negative underproduction factor. An underproduced article of low quality and high popularity would have a positive underproduction factor.

We count views at the monthly level using data released by the WMF. We calculate quality at the monthly level using the ORES quality measure. ORES is a machine learning-based quality measure that reflects the structural characteristics of articles, such as the presence of references, section headings, and inter-wiki links (Halfaker et al., 2016). ORES is a continuous measure with separate predictions for six article quality levels. We treat these levels as equally spaced to create a single quality score between 0 and 5. More details are provided in the supplement.

We operationalize editor experience as *revision count*: the number of revisions the editor has made before and including the current revision. Therefore, before conducting the filtering described in the data and sample section, we calculated each revision's ordinal position in

its contributor's edit history, ignoring any reverted contributions. To consider the influence of the potential for receiving social rewards, we also introduced a variable reflecting whether contributions were made by users who were not using accounts. Because these users are identified by IP address in Wikipedia, we call this measure *IP-based*.

C.3.4 Analytical Plan

We employ a hierarchical multiple regression model to test our hypotheses H1 and H2. We use *revision count (logged)*, *revision count (logged) squared*, and *IP-based* as predictors and a random intercept term for users using the *lmer* package in R and a 'raw' specification for our polynomials.

Although our hierarchical models attempt to correct for repeated measures of users, this approach does not account for the fact that experienced users might edit different articles than less experienced users in ways that reflect underlying differences in the types of users who go on to become less or more experienced (Panciera et al., 2009). To answer H3A and H3B, we fit a second group of models with user-level fixed effects using the *felm* package in R and our within-person sample. Because we have no within-person variation in account use, we cannot include *IP-based* in the within-person model. Due to heteroskedasticity, we use robust standard errors in all our models.

C.3.5 Ethics

This study was conducted entirely using publicly available data published by WMF and does not involve any interaction or intervention with human subjects. The IRB at our institution has reviewed this type of research using these data and determined it is not human-subject research. WMF fully anonymized article view data before release.

C.4 Results

The results of our polynomial model evaluating the relationship between underproduction and experience (H1) and between underproduction and having an account (H2) are shown

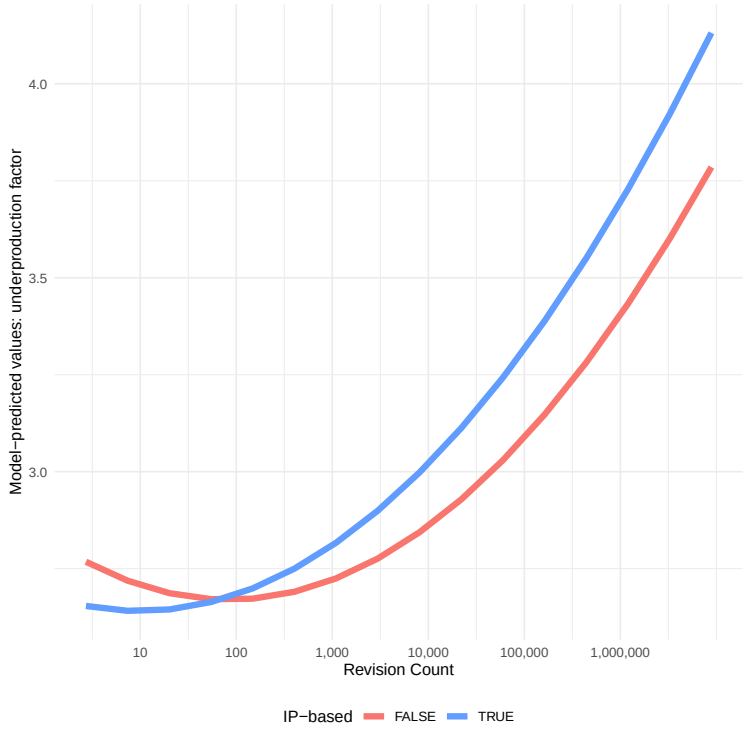


Figure C.1: The marginal effect of having higher experience on the average alignment of an article selected for editing. Increasing values indicate increased levels of underproduction, i.e. low-quality but highly-viewed topics.

in Table C.1 and visualized in a marginal effects plot in Figure C.1.

In the linear model for H1, we find that an increase in one log unit of experience is associated with a 0.0287 increase in the underproduction factor of the article selected for contribution (see Table C.1). The quadratic specification of our model is shown alongside the linear specification and is a substantially better fit for the data ($\delta 2LL \approx 200$; $\chi^2 = 405$; $df = 2$; $p < .0001$). The parameter estimates for all terms in both models are statistically significant, except for the interaction of revision count and IP-based. The polynomial model describes U-shaped curves with an inflection point at about 150 revisions for those contributing with an account (see Figure C.1). These results provide some support for H1: contributions by low-experience individuals are to less underproduced articles than those

	Linear Model	Polynomial Model
Intercept	2.5604	2.8029
	[2.5358; 2.5851]	[2.7682; 2.8377]
Revision Count (ln)	0.0287	−0.0738
	[0.0250; 0.0323]	[−0.0847; −0.0628]
Revision Count ($\ln^2$)		0.0083
		[0.0075; 0.0092]
Editor was IP-based	0.0609	−0.1511
	[0.0303; 0.0916]	[−0.1920; −0.1103]
Revision Count (ln) and IP-based	−0.0198	0.0376
	[−0.0272; −0.0123]	[0.0165; 0.0586]
Revision Count ($\ln^2$) and IP-based		−0.0004
		[−0.0035; 0.0027]
AIC	656075	655675
BIC	656136	655756
Log Likelihood	−328032	−327829
Num. obs.	192672	192672
Num. groups: editor_id_or_ip	89445	89445
Var: editor_id_or_ip (Intercept)	0.4476	0.4510
Var: Residual	164.7730	164.2004

Table C.1: Results from our revision-level hierarchical model of average alignment level of articles selected for editing. Bracketed values indicate a 95% confidence interval.

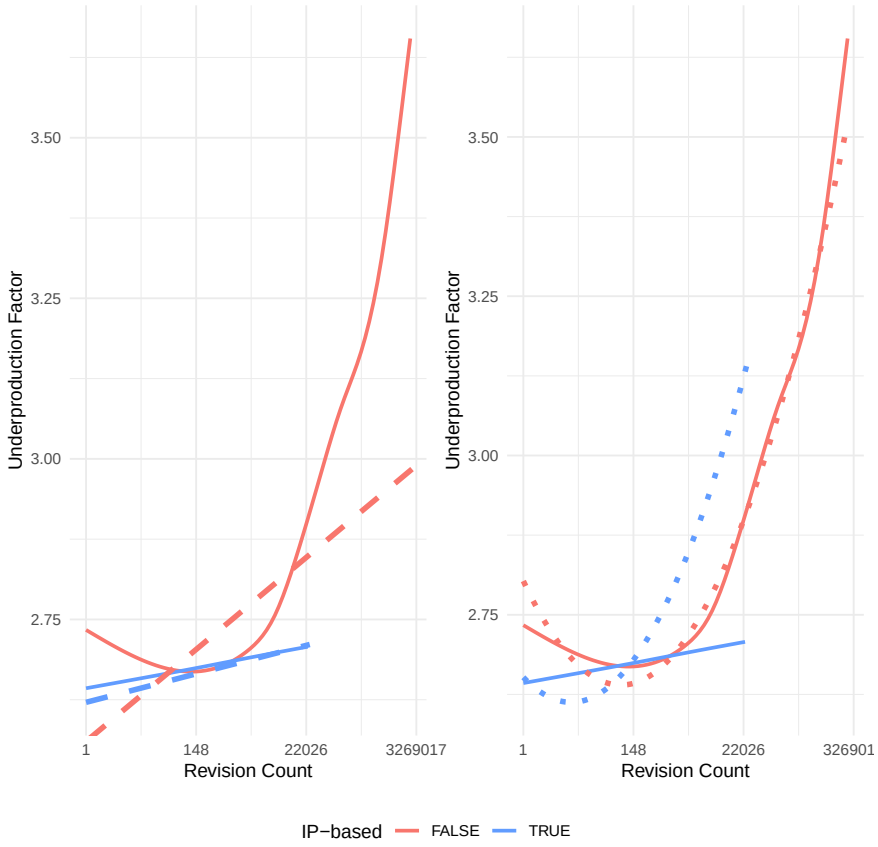


Figure C.2: The left pane shows a GAM smoothed line fit to a 10% sample of the data from the random sample with the result of the linear model superimposed as a dashed line; the right pane shows the same GAM line with the result of the polynomial model superimposed as a dotted line. Note the log-scaled x-axis.

from users with high experience levels. However, as we see from the polynomial model, the relationship is U-shaped. We also note that the highly skewed distribution of contributions is such that most contributors do not make more than a small number of contributions.

For H2, our linear model shows that contributing using an IP address is associated with a 0.0609 unit increase in the underproduction factor of the selected article. An increase in one log unit of experience is associated with a difference of -0.0198 in the same outcome. The quadratic specification of our model shows the opposite: the parameter estimate of the

	Within-Person: With Account	IP-based	With Account, Quadratic	IP-based, Quadratic
Revision Count (ln)	0.0808	0.0219	−0.0193	−0.0253
	[0.0803; 0.0812]	[0.0174; 0.0264]	[−0.0205; −0.0180]	[−0.0342; −0.0164]
Revision Count ($\ln^2$)			0.0077	0.0087
			[0.0076; 0.0078]	[0.0073; 0.0101]
Num. obs.	42170545	432367	42170545	432367
R^2 (full model)	0.1830	0.3575	0.1835	0.3578
R^2 (proj model)	0.0030	0.0002	0.0036	0.0006
Num. groups: editor_id_or_ip	9054	4290	9054	4290

Table C.2: Results of cross-sectional panel modeling to examine within-person change in the alignment of articles selected. Bracketed values indicate a 95% confidence interval using robust standard errors. The projected R^2 indicates modeling this data with random effects for experience level but without fixed effects for the individual would result in poor model fit.

main effect associated with using an IP address is a -0.1511 difference in underproduction factor, with a positive interaction term (revision count and IP-based) of 0.0376. The squared interaction term (revision count squared and IP-based) is both small in magnitude and not statistically significant.

To help understand these results, we construct Figure C.2, which shows a nonparametric GAM smoothed line fit to a 10% sample of the same data used to fit the models alongside the predicted values from both the linear and polynomial models. While the quadratic model is a better fit for the data, the linear model is more consistent with the behavior of IP-based contributors as shown in our nonparametric data visualization. If we use the linear model as the standard for evaluating our hypothesis about IP-based editors, we find that although they shift towards underproduced materials as they accumulate experience, they do not do so at greater levels than those contributing with an account. Overall, these results contradict our proposal in H2.

H3A and H3B relate to within-person change. The results of our model testing both parts of H3 are presented in Table C.2. In H3A, we proposed a within-person shift toward underproduced goods as contributors increase in experience. All models support H3A. We

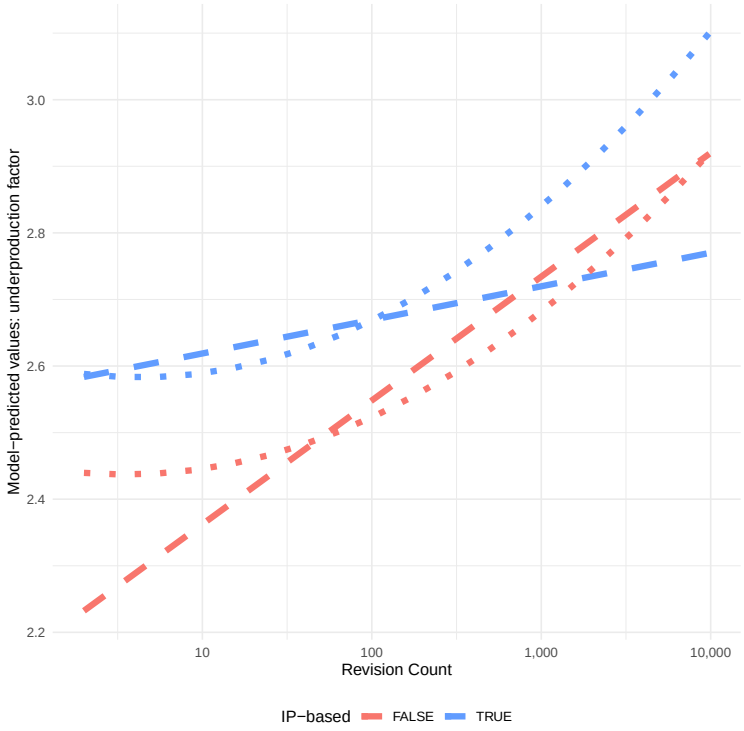


Figure C.3: Marginal effect of increased experience on contributor task selection from our within-person sample. We use median individual-level fixed effects. Dashed lines are predicted values using a linear model, while dotted lines use a quadratic model, see Table C.2.

observe a positive coefficient in the linear specification, suggesting an upward trend toward increasingly underproduced articles as individuals accumulate experience. In the quadratic specification, we observe a positive coefficient associated with the second-order term, suggesting a U-shaped relationship such that at higher revision counts, contributors select underproduced articles. We see mixed results for H3B related to those contributing without an account. When fit with a linear specification, the magnitude of the effect of experience is smaller for those contributing without an account (0.0219) than the coefficient associated with contributing with an account (0.0808). However, in the quadratic model, the coefficient in the squared term is slightly larger for those contributing without an account (0.0087) than those contributing with an account (0.0077).

To understand the implications of these models, we simulated task selection across experience levels, with the individual-level fixed effect set at the model estimated median. Figure C.3 shows the model-predicted marginal effect of accumulating editing experience on task selection. The figure shows that contributors tend towards underproduced articles as they accumulate experience.

Overall, our results from these models provide evidence that contributors' task selection shifts toward underproduced articles as their experience increases. This trend is present in people who contribute with and without accounts. However, some models point in the opposite direction of our hypothesis in H3B that those who contribute without an account will shift sooner than those who use an account. We offer additional interpretation of these results in our supplement.

C.5 Limitations

Although we measure the revision count of IP-based contributors, IP addresses change in ways that mean we cannot consistently associate individuals with IP addresses. Furthermore, some IP-based revisions are likely to be authored by experienced Wikipedians. As a result, our model may understate the influence of editing with an IP at low experience or overstate it at high experience.

Using revision count to operationalize experience introduces important limitations. For example, our measure does not distinguish between a revision that fixes a typo and one that adds a paragraph. Similarly, it does not differentiate between long-lived text and quickly deleted text. We use revision count because it is the measure of editor experience most widely used by both Wikipedians and Wikipedia researchers and because the most discussed alternative measures based on content persistence are both computationally complex and require a range of difficult decisions. Other research has shown that revision count is highly correlated with, and frequently leads to similar results as, measures that account for content persistence (e.g., Hill & Shaw, 2021). Ultimately, assessing how contribution types vary over the lifespan of editors remains a subject for future research.

Additionally, our design does not address causal factors among quality, popularity, and task selection. Indeed, we use measures that cannot be interpreted causally due to their granularity: our measures of quality and popularity are taken at the monthly level, not at the moment of editing.

Another threat was described by the Wikipedia editing community when Warncke-Wang et al.'s (2015) work on misalignment was first published. Community members stated that one explanation for underproduction is that some articles are more difficult to write than others and that this difficulty may be systematically distributed in a way that coincides with how people search for and consume articles.[4] There may be factors (such as conceptual generality) driving both high popularity and low quality. In part, using ORES as a quality measure mitigates this risk because ORES measures quality based on structural characteristics independent of conceptual qualities like completeness. However, this approach does not address the fact that, as a tertiary source, Wikipedia is limited by the availability of reliable published information, including the biases in the topics these sources cover. Writing a high-quality article about a high-interest, under-reported topic may be more difficult than writing about widely documented topics.

C.6 Discussion

C.6.1 Born, Made, or Something Else?

This article offers insight into community-wide trends and within-contributor changes in task selection behaviors. Our results in H1 and H3A suggest that as they increase in experience, contributors tend to shift their attention to underproduced content. This is consistent with the "made" argument that accumulating experience is associated with editing underproduced articles and, presumably, with increased exposure to socialization. Our results in H2 and H3B call this interpretation into question. Socialization suggests a social process, but contributors who participate without an account have less opportunity to build a social identity

[4]Wikipedia community-run internal newsletter, The Signpost: https://perma.cc/GX5U-32R9

in Wikipedia. How does this occur if these contributors also shift towards underproduced articles?

Our models suggest that IP-based contributors are initially less inclined to select underproduced articles but are comparable to account-based contributors for moderate contribution levels (i.e., in the 100-200 range). This finding opens up a new set of challenges for social research. Given that contributors without accounts also move toward underproduced goods—and hence may represent a valuable subgroup to seek to influence and retain—how should we explain their persistence and changing task selection?

One way to understand this shift in the behavior of IP-based contributors is through an extension of learning theory—e.g., Preece and Shneiderman's (2009) reader-to-leader perspective and Lave and Wenger's (1991) legitimate peripheral participation. As participants gain experience, perhaps their awareness of underproduced articles increases. Given that this process is mediated through technology in rich social computing environments, this may be evidence of what we call "technosocial learning," rather than social learning *per se*.

We suggest that technosocial learning may be an important mechanism to shape behavior in peer production environments. For example, consider the general phenomenon of feedback on a contribution. Someone contributing without an account might not receive such feedback. However, this contributor could notice whether their past contribution remains present.

Platforms offer other opportunities to learn from technical traces: contributors might experience success in mastering a particular formatting trick, observing the work of others, reading documentation, or seeing feedback others receive. A technosocial learning process, illustrated in Table C.3, may provide a mechanism for the changes in task selection we observe.

C.6.2 How May Underproduction Be Countered?

Given the previous finding of widespread underproduction in peer production (Champion & Hill, 2021; Warncke-Wang et al., 2015), countering it is an important challenge. One approach to addressing underproduction is to increase retention overall. Another approach is

	implicit	**explicit**
social	observing the interests of others	personal feedback
technical	noticing a good is low quality	automated feedback

Table C.3: Dimensions of technosocial learning in peer production with examples. We might expect contributors to learn both how to contribute and about underproduction through these channels. Observing public interest and noticing low quality are implicit signals available without creating an account.

nudging contributors towards more severe areas of underproduction earlier: both when they are newer to platform, and more quickly in response to emerging needs. For example, peer production communities responded during the COVID-19 pandemic by developing FLOSS analytic and visualization tools (Wang et al., 2021), organizing to produce Wikipedia articles to provide reliable information at high speed (Avieson, 2022), and producing personal protective equipment and filling supply chain gaps (Sarkar et al., 2023). Evaluation of recommender systems suggests they can influence contributors to improve underrepresented topics, providing the recommendation's relevance is kept constant (Houtti et al., 2023). Creating interest groups within a larger project may also be an effective way to tackle underproduction. In Wikipedia, contributors have self-organized around themes of interest to identify and work to close a wide range of participation and content gaps—e.g., WikiProject Women in Red (Tripodi, 2021), WikiProject Women Scientists (Halfaker, 2017), and WikiProject Vital Articles (Houtti et al., 2022).

However, more work is needed to understand when these interventions are effective. Ford et al. (2018) found that notification of existing contributors, writ large, was insufficient to address content gaps, instead recommending recruiting people with expertise related to the gap. A computational linguistics analysis from Schmahl et al. (2020) found that gendered bias in articles changed modestly, at best, between 2006 and 2020 when many targetted interventions were conducted. Successful interventions in task selection include not only involving diverse participants but also participants with diverse motives (e.g., through course

assignments and non-profit outreach activities (as per Coelho, Valente, Silva, & Hora, 2018; Gallagher et al., 2019; Herfort et al., 2023; Karczewska, 2023; Konieczny, 2012; Xing & Vetter, 2020).

C.6.3 Problematizing Retention and Account Creation

Our analysis points to promoting contributor retention as a means of countering underproduction. Scholarship in online community and peer production community management has also emphasized retention as a path toward building healthy online communities (e.g., Kraut et al., 2012). However, our findings should be read alongside the evidence against requiring account creation and the risks associated with overemphasizing the retention of existing users.

Requiring accounts has been found to carry unintended consequences in terms of diminishing overall community activity levels (e.g., Hill & Shaw, 2021). Being prompted to create an account increases the barrier to contributing. Maintaining a low barrier to entry may be essential to the success of peer production projects. Beyond this risk, Shaw and Hill (2014) found that governance of peer production tends to concentrate power in the hands of elite long-standing contributors, emphasizing that longevity risks further empowering elites. Additional analysis of long-term contributors appears in our supplement.

Increased retention may also magnify systemic biases reflected in peer production. For example, previous work has explored geographical biases in OSM (Thebault-Spieker et al., 2018), gender inequality in FLOSS participation (Qiu, Zhao, et al., 2023), gender bias in how FLOSS projects engage with contributors (Chatterjee et al., 2021), gender inequality in Wikipedia contributors (Hill & Shaw, 2013) and gender bias in Wikipedia content (Adams & Brückner, 2015). Therefore, while retaining contributors is valuable, it may reinforce inequality in the population of early contributors. Scholarship has shown that communities can make progress in closing participation gaps with interventions that address bias in social and technical processes through which people are retained unequally (Evans et al., 2015; Hilderbrand et al., 2020; Karczewska, 2023; Langrock & González-Bailón, 2022).

Along with recruiting and retaining diverse contributors, we believe there is also value in retaining those who contribute without accounts. A proposed update to Wikipedia's software slated for 2024 describes a change to the way that users without accounts are identified by replacing visible IP addresses with temporary identifiers.[5] Although details of this proposal are still in flux, this change will improve the privacy of those contributing without an account by hiding IP addresses that may reveal geographic location. Although the proposed alternative will improve the ability of researchers to track contributions from an individual across locations, it will also make it impossible to do studies of long-term engagement of users contributing without accounts—like ours. Evidence-based management of peer production communities faces multiple critical challenges: countering underproduction, enabling diverse participation, expanding community safety, building long-term commitment, and minimizing power concentration. Some of these goals may force difficult tradeoffs, as interventions intended to address one area may make others more difficult to address.

C.7 Conclusion

Although previous research suggests that the underproduction of peer produced goods is widespread, understanding how underproduction happens and how it can be addressed is still emerging. Our work advances a theory that experience contributes to changes in production. We test this theory and find some support: experienced contributors tend to select underproduced goods, whether they contribute with or without an account. However, contrary to our expectations, contributors not using an account are not different than those contributing using an account. In addition to the explanations of social rewards explored in previous work, we suggest that task selection changes in contributors without an account may be explained by our proposed notion of 'technosocial learning.' Our modern communication environment relies on peer production for both content and infrastructure. Understanding how to better counter underproduction is a necessary part of protecting and continuing to

[5]See implementation proposal: https://perma.cc/3H59-YLH8

expand the benefit of public goods developed through peer production.

Acknowledgement for Appendix C

Morten Warncke-Wang of the WMF provided both valuable advice on multiple occasions and access to his dataset. Aaron Halfaker provided advice in the effective use of the ORES machine learning system for quality analysis. A pilot version of this project was completed as part of the Advanced Regression course taught by Jeffrey Arnold in the Center for Statistics in the Social Sciences at University of Washington. Members of the Community Data Science Collective provided advice on early drafts of this project, including Jeremy Foote, Floor Fiers, Wm Salt Hale, Sohyeon Hwang, Sejal Khatri, Charles Kiene, Sneha Narayan, Aaron Shaw, and Nathan TeBlunthuis. The creation of our dataset was aided by the use of advanced computational, storage, and networking infrastructure provided by the Hyak supercomputer system at the University of Washington. The authors gratefully acknowledge support from the National Science Foundation, awards CNS-1703736 and CNS-1703049, and the Sloan Foundation through the Ford/Sloan Digital Infrastructure Initiative, Sloan Award 2018-11356.

C.8 Supplement to Appendix C

This supplement contains additional information about our methods, including our measures and dataset filtering. We also share additional interpretations and analyses: we discuss building reasonable hypotheticals and then use them to interpret our results, then offer some additional exploration of our results with respect to highly experienced contributors.

C.8.1 Additional Notes on Measures

We use ORES scores for our quality measure. ORES was trained by Wikipedians using quality classifications from 2015 and 2016, and uses the six quality levels defined in Wikipedia: Stub, Start, C, B, Good Article, and Featured Article. ORES produces a continuous prediction value from 0 to 1 for each quality grade, and the predictions for all six possibilities

sum to 1. In order to produce a single measure of quality, we use the weighted sum metric developed in Halfaker (2017), such that the lowest theoretical quality is 0 and the highest theoretical quality is 5; in our sample we observe a range of 0.04 to 4.87. As a robustness check on this continuous quality measure based on machine learning observations, we calculated the Pearson's correlation between the Wikipedian-assigned quality and the ORES-detected quality for the month of data analyzed by Warncke-Wang et al. (2015), which we found to be .696. Our article quality measure is calculated on a monthly basis, therefore in the case that a given editor made multiple contributions to the same article in a given month, we drop all but one of these as duplicate.

C.8.2 Impact of Filtering Population-Level Dataset

Viewership data for Wikipedia is not available before December 1 2007. Hence although we count a person's total number of contributions from the founding of Wikipedia, the fact that viewership is unavailable for the full timespan means we do not assess task selection before December 1 2007. Given our objective to understand human article editing, we also discarded revisions by accounts known to be bots, which we identified by scraping a Wikipedia webpage listing current registered bots, and from a historical dataset produced and released as part of Geiger and Halfaker (2017).

Our interest is in articles whose quality can be measured consistently, so we eliminated revisions made to non-article pages and two types of pages that serve special functions: revisions to "List of..." and revisions to "Disambiguation" pages, both are only lists of other articles. We also excluded the revisions made to the simple functional pages called 'redirects' (Hill & Shaw, 2014). Redirects are not articles but rather are used to manage the fact that a single article may be known by multiple names or spellings; although creating redirects is an important form of work in Wikipedia and serves the needs of information-seekers, we do not have a measure of quality for them (they are typically a single line of code, rarely viewed by the public).

We also removed any revisions that appear in the logs due to quality management pro-

cesses. These include "identity reverts": quality control actions which 'undo' some previous revision or revisions and restore the article to a previous state without adding any new content. Reverts of this kind are often made through tools which look for patterns characteristic of vandalism and then route work to vandal-fighters. Identity reverts are detected by examining the cryptographic hash of a page after an edit action and then looking forward and backward 15 steps in the revision history. We also removed revisions made to pages which were moved because the moving procedure renames and collapses the history of the page in ways that obscure the quality and popularity of the page at the time it was edited (Hill & Shaw, 2015). We also excluded revisions that were later deleted. Revision deletion is relatively rare and occurs only in cases of extreme violations such as the posting of malicious or harassing content.[6] We also excluded revisions made to articles that were later deleted.

C.8.3 Distribution of Fixed Effects

The violin plot in Figure C.4 shows the overall distribution of fixed effects; the relatively thin tails suggest that the median is indeed a reasonable measure of central tendency for our fixed effect.

C.8.4 Building Reasonable Hypotheticals

We face several choices in exploring our models via hypothetical scenarios. First, we need to select sufficiently different prototypical contributors. According to Wikipedia's published contributor statistics[7], even making one revision qualifies a contributor for the top 30% of all users, since the majority of accounts do not make a contribution, while making 500 contributions qualifies a contributor for the top 0.25% of all users. Making 500 contributions also grants additional privileges, including editing of articles under the most restrictive forms of editing protection. Approximately 110,000 contributors are in this category. The next

[6]https://en.wikipedia.org/wiki/Wikipedia:Revision_deletion

[7]See https://en.wikipedia.org/wiki/Wikipedia:Wikipedians

Table C.4: Population sizes after filtering steps, in sequence.

Filtering Step	Number of Revisions in Population
Before filtering	626,444,823
After Date Filter	463,410,567
After Removing Bots	385,404,057
After Removing Non-Articles	286,639,725
After Removing "List of..." Articles	274,787,780
After Removing Disambiguation Pages	273,855,868
After Removing Redirects	266,228,183
After Removing Identity Reverts	235,711,469
After Removing Moved Pages	158,139,159
After Removing Deleted Revisions	157,844,764
After Removing Deleted Articles	130,537,186

lower tier of permissions is unlocked at 100 contributions; this places the contribution to the left of the inflection point we observe in our model plots. We argue that people who have contributed once versus those with 500 revisions of experience seem meaningfully distinct without being unattainably different.

Another important choice in building our hypothetical is accounting for the complex construction of our underproduction factor measure. If we want to compare contributors with respect to the number of people who benefit from their effort, given differences in underproduction factor, we face a challenge: since underproduction factor is a ratio of quality rank and popularity rank, there are multiple ways numerically that a given underproduction factor might be achieved. We also need to account for the fact that underproduction factor is a continuous measure, and in our dataset we might observe relatively few values or no

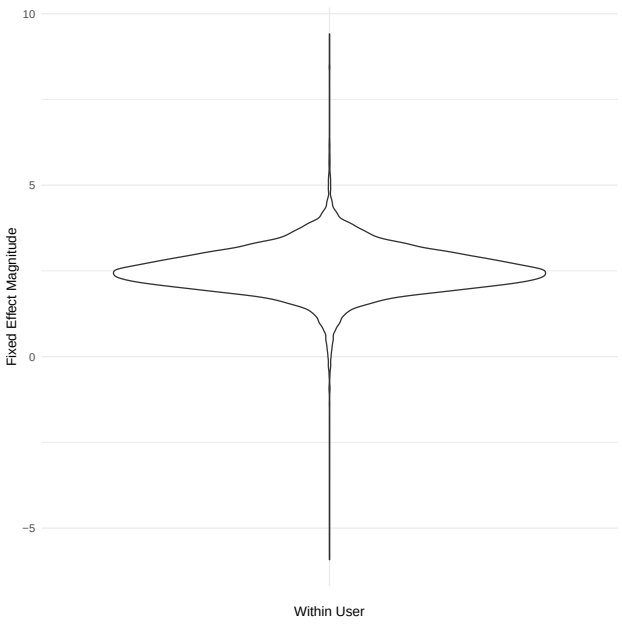


Figure C.4: The distribution of fixed effects in our model.

values at a given point estimate for underproduction.

We therefore draw two subsamples: all revisions to articles that were at the time within .01 of the model-predicted underproduction factor of an article edited by our prototypical newcomer (who we selected as someone having made 1 contribution, and for whom the model predicts an underproduction factor 2.8029) and those within .01 of the model-predicted underproduction factor of an article edited by our prototypical experienced contributor (with 500 edits, underproduction factor 2.6661). To understand the central tendency of these subsamples, we then take the median number of views they received during the same month they were edited.

C.8.5 Interpretation using hypotheticals

Given an observed range of underproduction factor from -7.986 (overproduction) to 12 (underproduction), these effects appear to be relatively small numerically. However, if we interpret the practical effect from a utilitarian point of view, we find that the impact does indeed

hold practical importance. Imagine a prototypical new editor with a revision count of 1, who is predicted as editing articles which have an underproduction factor of 2.8029, while an editor with 500 revisions is predicted as editing articles which have an underproduction factor of 2.6661. What if everyone edited like the person with 1 edit, versus if everyone edited like the person with 500 edits? In this hypothetical situation (see the Methods Supplement for additional methods notes), the median views to the article edited by our expert is 1810, and the median views to the article edited by our newcomer is 1267. Therefore, we can anticipate that the revision of our expert serves 543 more people that month. Given that the number of edits made per month is quite large (e.g. 5 million to English Wikipedia in August 2023), the potential benefit of task selection that looks more like the expert is likewise quite large: the scale of Wikipedia can magnify even small changes in task selection patterns.

C.8.6 Divergent Paths for Long-term Contributors

Although our results suggest that experienced contributors tend to contribute to underproduced articles in ways that are consistent with our hypotheses **H1** and **H3A** and **H3B**, the heteroskedastic nature of the data (confirmed through a Breush-Pagan test, $p < 0.05$) and the improved fit from a polynomial model for account holders led us to conduct some additional descriptive analysis to better understand high volume contributors. We drew a random sample of 12 contributors from our within-person sample; the individual-level trajectories are depicted in Figure C.5. Individuals are left-censored (e.g. Account H) because our underproduction measure requires view data which is unavailable before 2008. Individuals such as Account E follow the general pattern we found in our model, tending toward underproduced articles as they develop experience. However, we observe that some individuals, such as Account G, trend in the opposite direction from that predicted by our model. We have much more to learn about task selection in high-volume contributors.

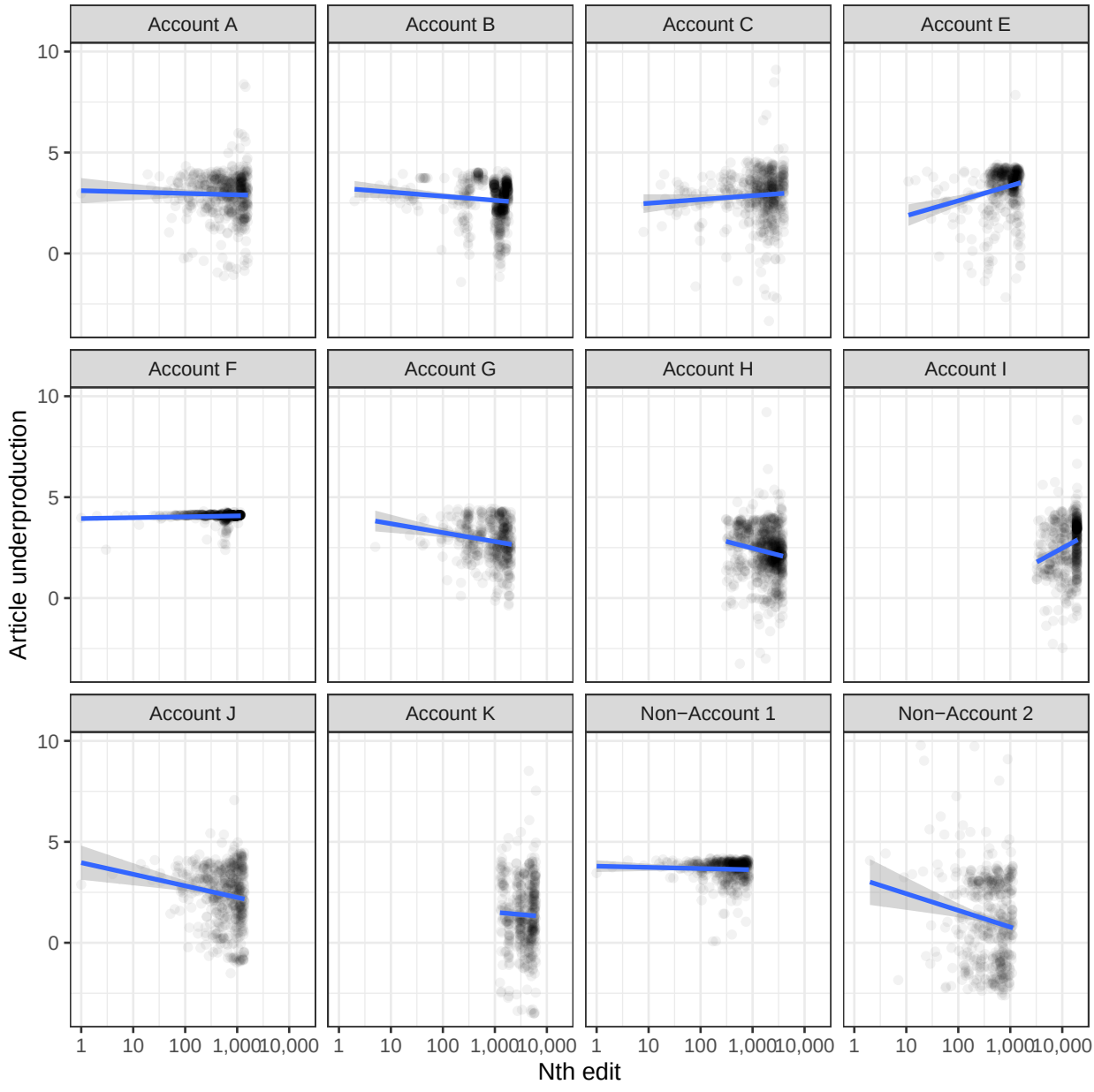


Figure C.5: Task selection trajectories for a random sample of individuals.

Appendix D

QUALITIES OF QUALITY: A TERTIARY REVIEW OF SOFTWARE QUALITY MEASUREMENT RESEARCH

This chapter describes a systematic process I followed in order to understand the current state of measurement techniques in computer science for measuring quality—a key input into the analysis of underproduction. The measure of quality I used in Chapters 3 and 5 are informed by the observation that on a conceptual level, quality may be measured through structural metrics, outcomes, or holistically—I used a structural approach in Chapter 5 and an outcomes approach in Chapter 3.

I led all parts of this project. This chapter has been previously posted as a working paper as: Champion, Kaylea; Khatri, Sejal; and Hill, Benjamin Mako (2021) "Qualities of Quality: A Tertiary Review of Software Quality Measurement Research" [Working Paper]

D.1 Introduction

Software quality is of immense concern to both software engineering and associated disciplines, and to society in general: our day-to-day lives depend on an ecosystem of software components serving as global digital infrastructure (Eghbal, 2016). In order to understand software quality, we tackle three fundamental questions:

RQ1: How does the software quality field approach the analysis of software quality?

RQ2: What kinds of evidence is used when measuring software quality, and what progress has been made?

RQ3: What challenges have been identified in the software quality field?

Given the existence of a large number of secondary reviews on software quality measurement, we conducted a tertiary review—that is, a review of reviews. A tertiary review supports the broad perspective required to answer these research questions, building on the efforts of secondary review authors to assess individual studies and synthesize subfields. Further, a tertiary review can suggest where subfields working independently of one another may find shared concerns. We begin by reviewing the insights of past tertiary review efforts in §D.2, before offering a description of our tertiary review method and dataset in §D.3. We offer a synthesis of the secondary reviews we identify in the form of three broad perspectives on quality and the evidence and progress made within them in §D.4. We describe challenges that emerged across these perspective in §D.6. We comment on these findings in §D.7, suggesting how this work can be used by quality researchers. We detail potential threats in §D.8 before concluding in §D.9 with implications for software engineering research.

Our paper offers three contributions to the software engineering literature on quality through the examination and synthesis of 75 secondary reviews. First, we develop an ontology of three perspectives used in current lines of research with respect to software quality. Second, we synthesize the current state and key findings within each of these perspectives. Third, we offer an analysis of the key victories and most vexing struggles currently facing software quality researchers.

D.2 Related Work

In 2004, a highly cited piece from Kitchenham et al. called for software engineering research to adopt an evidence-based approach, inspired by evidence-based medicine (Kitchenham et al., 2004). Kitchenham et al. describe evidence-based software engineering as tackling not only carefully defined research questions and developing strong evidence, but also "critically appraising that evidence for its validity... impact ... and applicability" (p. 275), integrating this critical appraisal into findings and future work, and continuing to evaluate past work

for improvement. Our review is a response to this challenge.

A series of previous reviews have attempted to sketch the contours of a still-emerging body of research on software quality in various ways. Previous reviews have tended to focus more narrowly on specific challenges with respect to evidence and validity, as well as specific promising approaches. Our review is complementary to these prior reviews in that it grapples with software quality in a cross-perspective manner and in that it is broader both in scope and scale.

With respect to evidence, the mapping review in Bailey et al. (2007) found an absence of comparative evaluation of the quality of software produced through Object-Oriented versus non-OO paradigms. With respect to validity, Syeed et al. (2014) found in their systematic review of contributions from software quality research to open source development practices that many of the most significant quality predictors were rarely used in research. Previous authors have also identified potential avenues for progress: mining software repositories unlocks longitudinal data for quality research (De Farias et al., 2016), and AI techniques have much to contribute if data and metric validity concerns can be addressed (Fernandez Del Carpio & Angarita, 2018).

D.3 Review Methodology

As a systematic literature review, this tertiary analysis follows a reproducible process for identifying, analyzing, and synthesizing a comprehensive set of relevant high-quality research. Just as secondary reviews treat primary studies as their evidence, tertiary reviews use secondary reviews as inputs. In the following, we describe our use of the systematic review method described in Kitchenham et al. (2004) using the stages illustrated in Figure D.1. When we refer to papers that are part of the corpus analyzed through systematic review, we use a separate index number, prefixed by 'S' (i.e., S1-S75). All of these papers are listed in this review in the Secondary Literature section. Other citations are included as normal.

Our review protocol was informed by guidelines articulated for software engineering, information systems, and medicine (Kitchenham et al., 2004; Okoli, 2015; Smith et al.,

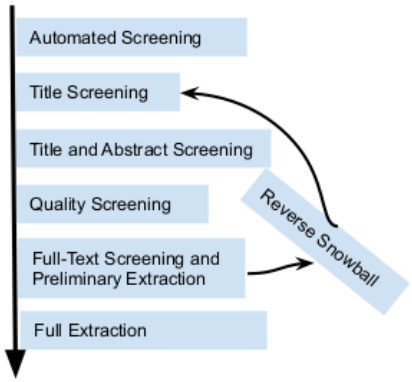


Figure D.1: Stages of the systematic literature search and screening process

2011). During the conduct of this review, the review team regularly met to verify fidelity to the protocol and update the search plan when needed. We adopted shared processes around search and retrieval including the use of shared spreadsheets for data capture and the use of a shared reference database using Zotero.

D.3.1 Automated Screening

We collected our initial dataset from a set of database indexes of scholarly papers using a schema of keywords, operators, and fields. The search process was developed in collaboration with research librarians with expertise in Computer Science and in Communication. We ensured that publications by the ACM and IEEE were fully indexed by these services. The final list of databases indices used and counts of results from our search process is listed in the supplemental materials.[1] Because software quality is a subject of active debate and consideration, we refined the list of keywords iteratively, ultimately using those in Table D.1. Indices vary in their search capabilities; we include the specific queries we ran in each database as supplemental material.[2] To manage the scope of the project, we limited our search to work published in the years 2000 to 2020 inclusive and written in English. Our

[1]https://doi.org/10.7910/DVN/XPOUO4

[2]https://doi.org/10.7910/DVN/XPOUO4

Table D.1: Search keywords

Term	Field	Synonym
systematic review	abstract	literature review, systematic mapping, meta-review
quality	abstract	bug, flaw, vulnerab*, problem, reliab*, stab*, performance, issue
software	subject	—

initial search results returned 9,916 results.

As S22 and S56 found, index-only search techniques are often insufficient to find all literature related to a question. As a result, we supplemented our primary results using the reverse snowballing technique (Wohlin, 2014)—i.e., when the secondary works we identified referenced other secondary studies, we screened those as well. We repeated the snowball process three times, stopping when we found no references to any new secondary work. Reviews found through snowballing were filtered in the same way as the results of our index searches.

We conducted both an automated and a manual inspection process to identify and remove all duplicates. This resulted in a total of 7,811 articles in our initial deduplicated dataset.

D.3.2 Title, Abstract, Quality, and Full-Text Screening

After deduplication, we screened reviews based on their content and quality using the three inclusion criteria listed in Table D.2. We conducted our screening in four rounds, reviewing

Table D.2: Screening Criteria

Criteria	Description
in scope	must be a secondary analysis; must define software quality; must attend to measurement of quality
generalizable	must offer insight for a range of quality measurement tasks; exclude articles that are narrowly applicable to a single domain
high quality	must be peer reviewed; not gray literature, self-published, or publications from predatory journals

Table D.3: Articles present after screening stages

Round	Initial	Unique	Automated	Title	Abstract	Final
Primary	9916	7656	-	627	194	54
Snowball 1	227	106	93	88	80	18
Snowball 2	87	48	37	36	33	3
Snowball 3	4	1	1	1	1	0
Total	10234	7811	131	752	308	75

each article's (1) title, (2) title and abstract, (3) quality, and (4) full text. At each step, we made conservative decisions and retained a paper for the next round of deeper review if we were uncertain. The results of each screening step are summarized in Table D.3.

Our inclusion criteria for quality were modeled after Kitchenham et al.'s (2009) and intended to limit our sample to peer-reviewed publications. We removed 11 items published through predatory publishers that we determined by examining whether a given journal appeared on Beall's list.[3] We did so because predatory publishers do not engage in substantive peer review (Djuric, 2015). We also excluded one paper which appeared only in a preprint form and five items that were PhD theses.

We examined conference papers with care and searched for associated journal articles in each case because it is common for preliminary results to be presented at a conference, revised, and then later published in a journal. In the 13 occasions when a journal article was found with the same authors and general findings as a previously published conference paper, we excluded the conference paper and retained the journal article to prevent redundancy.

The process resulted in a dataset of 75 secondary reviews of software quality. We summarize the results of the search, screening, and the snowballing process in Table D.3. Bibliographic data for the full list of 75 reviews used in this paper are available in the secondary bibliography. The full dataset is available in the supplemental materials.[4]

D.3.3 Synthesis Through Thematic Analysis

In order to answer our research questions, we conducted a thematic analysis of the 75 reviews in our final corpus following the qualitative research method described by Braun and Clarke (2006). Thematic analysis is a form of content analysis, and involves careful reading of texts and the identification of patterns and relationships across those texts (Cruzes & Dybå, 2010).

We use the thematic analysis method primarily as a means to generate insight into the kinds of work being done in the software quality research field, and as a means of constructing

[3]https://beallslist.net/

[4]https://doi.org/10.7910/DVN/XPOUO4

a synthetic view of both the overall argument, approach, and assumptions being made in this work as well as the specific and countable traits of the reviews.

Our thematic analysis was focused on understanding how each review approached software quality (RQ1), the current state of work in each perspective, including both evidence and results (RQ2), as well as problems and challenges that emerged (RQ3). The output of our thematic analysis is both a series of categories and descriptions that capture the approaches being taken across the field and a set of shared challenges. Preliminary data extraction was done to a spreadsheet, shared with the team, which assigned a unique ID to each study. We recorded the source (e.g., original search, snowball round), publication venue, abstract for quick reference, the date of publication, and the author's name for their overarching method (systematic mapping, systematic literature review, etc.).

To perform our thematic analysis, the first author read each review in detail and summarized the methods, data, findings, and concerns expressed in each one. The first author weighed multiple categorization schemes and common threads, re-reading each article as necessary. This process culminated in an iterative series of proposals made to the other two authors for ways to describe how the perspectives expressed in the reviews might be grouped and distinguished from one another. Individual reviews and characterizations of challenges were sorted and re-sorted by consensus until we arrived at three themes that characterize differing perspectives on the subject of software quality. As part of the synthesis process that followed, we also identified opportunities and concerns that recur across varying perspectives on quality research and seem to characterize the state of software quality research as a subfield; these observations are included in §D.6.

D.4 Findings: Three Perspectives on Quality

We identified three perspectives in the software quality field using thematic analysis (RQ1). The first perspective is that **quality research is a methodological challenge**; work within this perspective approaches quality as a question of identifying the right approach for analysis: either through application of *quality heuristics*, i.e. reviews that describe the

Table D.4: Reviews in each theme

Theme	Count	Proportion
Quality Research is Methodological	30	40%
Quality Research is Outcome-Oriented	27	36%
Quality Research is Holistic	18	24%

detection of generalized rules-of-thumb, practice-oriented antipatterns and "bad smells" in code (§D.4.1), and/or through the detection and prediction of *quality structures*, applying metrics that assess code structurally for specific traits such as complexity that are seen as problematic (§D.4.1).

The second perspective we identified is that **quality research is oriented to real-world outcomes**, with various aspects such as *dependability* and *maintainability*. This real-world orientation is not conducted in isolation—indeed, often this work builds on the methodological groundwork of the first perspective. However, this perspective is distinguished by its orientation toward grounding results in the resulting impact on users, engineers, systems and application staff, etc. Although the reviews in this perspective did not typically take up the ISO standard directly, and in fact commented on a lack of engagement with the standard in primary studies, we also found that work done within this perspective nonetheless echoes the language and typology articulated by the ISO standard.

The final perspective we identified is that **quality research is holistic** for studies taking up a very broad or implicit approach that seeks to be comprehensive even at the cost of introducing fuzzy definitions (D.4.3). This approach

The distribution of reviews across these three perspectives is shown in Table D.4. Although we placed each secondary review in our corpus into a single perspective, a given empirical (primary) study may appear in multiple secondary analyses. We present our syn-

thesis of the work done within each perspective in turn by describing the theme in terms of both its definition (RQ1) and the evidence and progress we saw (RQ2) within each perspective. We then synthesize challenges and future areas of work (RQ3), many of which cut across the software quality field.

D.4.1 Quality Research is a Methodological Challenge

Some quality research focuses on refining the underlying techniques used to analyze code and measure its traits. We saw two areas of focus within this perspective: some work took up a heuristic focus, in which the analysis was guided by broad but open-ended design principles about what makes code good quality (e.g., avoiding duplicated code), while other is oriented to the use of more closed-form structural metrics (e.g., measures of complexity) as a means of predicting future defects and prioritizing quality assurance efforts.

Focus on Heuristics

This theme gathers work taking the perspective that readable, well-designed code is easier to maintain and less likely to cause future issues; gauging whether code is readable or well-designed is fundamentally qualitative. The primary strategy for improving quality within this theme is refactoring: revising code in order to improve the underlying design without altering overall functionality. All of the review articles we identified as focused on heuristics either directly referenced Fowler's 1999 book *Refactoring* (Fowler, 1999) and/or discussed the metaphorical notion of code smells. Fowler describes Kent Beck's turn of phrase "code smells" as "not-quite-right code:" a series of tell-tale signs of code problems, each with a conceptual and evocative name (e.g., God Class). Fowler characterized these smells as inherently subjective but relatively easy for inexperienced programmers to identify. Fowler also associated each bad smell with a proposed set of refactoring strategies that could remedy the issue.

Although some of the empirical work reported by these reviews also proposes new smells, research on this topic typically addresses Fowler's list of smells (especially two of Fowler's

original 22: Duplicated Code and Large Class) [S41]. The earliest work in this theme used manual identification techniques, but machine learning (ML) techniques are increasingly prevalent. This includes lines of work seeking to identify optimal algorithms and cutpoints, and others seeking automation of Fowler's associated refactoring strategies.

Substantial effort has been put into the refinement of code smell detection techniques. S30 provides a review of code smell mining approaches including static, dynamic, semantic, metrics, and historical analysis, using structural features but ultimately oriented to a given heuristic. S30 also enumerates several detection techniques in use, with disadvantages noted in parentheses: manual (time-consuming and error-prone); metric-based (no consensus on thresholds); symptom-based (low precision due to varying interpretations of symptoms); probabilistic (no disadvantages listed); visualization-based (about 50% precision and recall), search-based techniques using ML (success varies); and cooperative-based using genetic programming (may not generalize).

Our analysis found substantial disagreement in the literature on whether bad smells are a problem for quality. We found that some reviews assumed that code smells were negatively associated with quality and focused on assessing empirical work oriented to detecting smells [S12, S14, S75]. Other reviews examined the weight of evidence associating bad smells with low quality software [S4, S36, S41, S42]. The latter group found that bad smells had only a weak, and sometimes even positive, relationship with measures of software quality. These results suggest that future researchers should not assume that the presence of bad smells is an indicator of lower quality. Instead, researchers should evaluate the evidence relating each specific code smell heuristic they are using for their measurement to a specific facet of quality.

Recognizing that code smells are both widely used and unreliable indicators of quality, S59 took up the task of categorizing false positives in antipattern and code smell research. S59 characterized the evidentiary challenge as twofold: a practical challenge due to the fact that manual validation of results from code smell analysis can be time-prohibitive, and a construct validity challenge caused both by a limited understanding of what maintainable

code looks like and a lack of empirical support. S5 found that ML-based studies using metrics associated with smells performed better than ML studies that used smells directly. However, this result was uncertain given weak evaluation metric reporting—S5 observed that studies more often reported threshold-dependent metrics instead of metrics such as AUC.

S42 expressed concern about the relative predominance of academic researchers in this area. While S34 found that all authors in the 78 studies they examined were from academic environments, they were "mainly working in research groups with support from industry." This more nuanced view of affiliation led S34 to conclude that there is no clear answer as to whether research on smells is primarily conducted by industry or academia.

Many researchers called for better coordination between research and practice. In addition to addressing basic validation challenges, future work taking the quality as heuristic perspective may find substantial value in learning from practitioners directly. S52 found that many refactoring strategies commonly used by practitioners have not been investigated while some rarely-practiced approaches have been the object of substantial study. Because code smells are a reflection of practitioner experience, maintaining close awareness of practitioner experience seems likely to support more applicable results.

Focus on Structure

Of our 75 identified secondary studies, ten examined quality through the lens of predicting defect-prone areas of code. This approach can optimize the development process by alerting developers to issues early, and can assist QA engineers in targeting the most fault-prone areas for more intensive scrutiny.

Many studies in this theme used static metrics extracted from Halstead (Halstead, 1979) and McCabe (McCabe, 1976) [S15, S25] with OO bug prediction studies tending to use the Chidamber-Kemerer (C&K) (Chidamber & Kemerer, 1994) method-level metrics, although several other sets of metrics have been proposed [S15, S62, S28]. S43 reviewed dynamic metrics, which allow for measurement to occur in a contextualized running system. Because dynamic metrics generally extend the essential logic of static metrics such as C&K,

they are at risk of inheriting any validity problems present in the underlying static metrics. However, dynamic metrics also allow for the study of runtime traits, such as utilization of polymorphism. S28 found that despite this proliferation of metrics, relatively few had been validated at scale (with the exception of C&K). S23 encouraged extending the use of class-level metrics because of their applicability during the design phase. Despite this heavy use of code-oriented metrics, S28 found that process-oriented metrics such as code delta, code churn, code age, and change set size were stronger predictors of post-release bugs than static code metrics.

S20 engaged with the relatively new topic of cross-project defect prediction—i.e., the notion that ML algorithms might be trained on one project and then used to assess another. S20 observes that although many ML algorithms "work well under the assumption that source and target data are drawn from the same distribution, [this assumption] might not hold for cross-project defect prediction" (p. 124). Several studies summarized by S20 found that prediction may not be bidirectional: models trained on open-source code had high accuracy when applied to closed-source code, but the reverse was not true.

S27 argues that the ability to predict bugs suggests that early intervention may be possible, but that more work is needed to apply strong empirical methods to identify effective metrics to address early parts of the software lifecycle. S15 and S23 called for research into class-level metrics to enable fault prediction early (i.e., during the design phase). S28 found that static metrics were of limited utility in predicting fault-proneness in highly iterative development processes. Another area of concern shared among defect prediction scholars is the connection between academic research and applied and industrial environments. S20 expressed concern that defect prediction research continued to be conducted in laboratory environments rather than in the context of practitioners' work. S28 remarked that academic researchers conduct the majority of defection prediction research and that academics were more likely to consider OO metrics while industry researchers tended to use process metrics. Given S28's finding that the process metrics used by industry researchers were also more effective, they recommended that academic researchers follow the lead of industry researchers

in this regard.

D.4.2 Quality Research is Oriented to Real-World Outcomes

Some lines of quality research are oriented to specific desirable outcomes, with particular focus on maintainability (which emphasizes the experience of developers maintaining code), and on dependability (which emphasizes the experience of users and systems and application administrators)

Focus on Maintainability

The perspectives in the studies we discuss in this section describe quality in relation to the process of software maintenance. We identified four distinct lines of work within this broad theme: (1) studies exploring the broad notion of *maintainability in general*; research on *evolvability* as a measure of quality; research on *technical debt*; and work seeking to understand processes of *aging* in software.

Maintainability in general Maintainability as a facet of software quality is concerned with how efficiently and effectively a given codebase can be maintained over time. Despite the fact that multiple studies have found that maintenance is the most costly portion of a software product's lifecycle, S31 and S72 observed that very little is known about predicting maintainability. S73 observed a lack of consensus with respect to open source maintainability metrics but that coupling metrics were most common, appearing in 5 of the 14 studies they identified.

S31 recommended more robust scales to quantify maintainability and emphasized the need for model validation. In particular, S31 pointed out that only 4 of the 15 studies they identified used a validation technique like leave-one-out (LOO) cross validation. S54 examined maintainability measurement studies in the range of 2003-2012 and also found that external validation was lacking.

S72 suggested that developer team performance-based metrics like Mean Time to Repair, Mean Corrective Maintenance Time, and so on, were appropriate for measuring maintain-

ability. S72 found that few studies had considered the maintainability of non-OO systems, the impact of Agile processes, or component-based development.

Evolvability Maintainability studies were also concerned about evolvability. S29 framed the evolvability challenge as a kind of "stability"—that is, whether a code base can be adapted to incorporate new needs with minimal effort rather than needing to be disruptively or deeply overhauled. The measurement of evolvability as a facet of quality is in a formative phase. As a result, reviews categorized with this theme used a variety of terms including evolvability, stability, and reusability. S11 examined architecture quality studies as they relate to evolvability, but found most were case studies. Of the 82 studies they examined, only 10 were metric-oriented studies with no common or externally validated measures among them. S29 observed that while multiple evolvability metrics have been proposed and validation work is needed, the metrics share an emphasis on the fundamental design principles of low coupling, high cohesion, and functional independence. S35 conducted an exhaustive review of the "stability" concept at the design, architecture, code, and requirements level across 166 articles. Although the amount of work done in this area is increasing, S35 found that stability studies have not generally been conducted as empirical work in applied settings to understand stability successes and failures in practice.

A related concept is "reusability" which is a sub-facet of maintainability in the ISO25010 quality standard. S10 considered quality as reusability and sought to determine which of a long list of non-functional requirements (NFRs) were loosely related to, positively correlated with, or negatively correlated with, reusability. The tradeoffs among quality facets observed in the articles re'viewed in §D.4.1 were also evident in S10's review of quality as reusability: of the quality facets outlined in ISO25010, reusability was negatively correlated with 4 of the 47 facets considered (performance, dependability, cost, and, surprisingly, portability) and positively correlated with 16 of the 47 (including safety, interoperability, and documentation).

Technical debt Technical debt is a metaphor used in software engineering since at least 1993 when Cunningham wrote that "shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite" (Cunningham,

1993, p. 30). Given the descriptive success of the technical debt metaphor, S68's 2012 article set out to establish a theoretical model using a systematic review. S68 found that technical debt encompasses a range of concepts and is involved with examples including lack of abstraction and code duplication, excessive complexity, and deviation from a reference architecture. S68 also found that the definition of technical debt varied across the software engineering literature and was used to invoke known bugs, incomplete testing, missing features, scant documentation, and more. S68 proposed a theoretical framework that elaborates the metaphor into such concepts as interest, bankruptcy, and gearing (also known as "leverage" or sustainable debt level). S68's elaboration of the concept includes such elements as code decay, design/architectural debt, infrastructural debt, documentation debt, testing debt, known issues, and missing features. S68 identified potential causes of technical debt as project constraints, lack of debt visibility, reckless decisions, and deliberate (or inadvertent) debt accrual. Potential consequences of these forms of debt include improvement or risk to timelines, maintenance difficulty, and ongoing quality problems.

S58 extended S68's analysis to incorporate practitioner perspectives, expanding the metaphor further to incorporate the very real cost of technical debt and to include the notions of different kinds of decision-making processes associated with debt (strategic debt, tactical debt, and incremental debt), debt amnesty, and debt repayment, as well as the morale and productivity impacts. S53 offered several additional categories of debt, such as people debt, process debt, build debt, service debt, usability debt, and versioning debt.

S22 observed that although the term technical debt was coined in the early 1990s, the recent growth in published studies of technical debt did not begin until 2009. S22 speculated that the Managing Technical Debt workshop may have raised awareness of this research avenue. S22 found that empirical studies of technical debt differ in their definitions of debt, especially with regard to whether defects should be considered technical debt. Although the concept of technical debt has been broadly elaborated, S22 found that most technical debt research was focused specifically on code technical debt, especially the measurement and identification of it, although architectural and design debt have also received some attention.

Two secondary studies we identified specifically addressed architectural debt. S8 enumerated types of architectural technical debt, described refactoring strategies and proposed a unified model of architectural debt. S8 found that no tools tackled the full spectrum of architectural technical debt and its management. S47 observed that the bulk of architectural technical debt studies were conducted from an academic perspective and that more collaboration with industry could lead to beneficial knowledge exchanges.

Although most authors articulated technical debt as a concept distinct from code smell, S41 described them as synonymous. In any case, the concepts of debt and smell have a complex relationship: a bad-smelling section of code may well represent substantial technical debt. However, some debt may not smell and some smells may not be debt sources but rather, say, an intentional design choice that improves performance. S75 described design smells as a leading source of technical debt. Smells and debts may also share an ultimate cause: S14 lists lack of skill or knowledge, changing requirements, technology constraints, process problems, time pressures, and politics as precursors of code smells. These same contingencies, constraints, and limitations are also implicated in technical debt, particularly code-level and design-level debt.

S22 found that the most widespread approach to measuring technical debt is a series of calculation models, although some approaches also use code-level metrics. S53 found that the most common indicator of technical debt is the code smell, with common measures including detection of the God Class code smell, measures of code complexity, and searches for duplicated code. S22 found that, as with code smells, the most common approach to repayment of technical debt is refactoring, and to a lesser extent full rewriting, re-engineering, or automation. S22 identified 29 different tools for tackling technical debt, including both products from major vendors and numerous free and open source options. S22 found that these tools typically tackle code and design debt using source code as input, but that none work to prevent technical debt. S32 and S56 took up the task of developing decision criteria for the question of repaying technical debt. Once again, evidentiary challenges persist: none of the 38 articles reviewed in S32 and only 4 of the 100 articles reviewed in S53 include any

empirical work.

Software Aging Software aging studies typically use one of two definitions: they either define aging as a consequence of code flaws that come to light in a long-running system (e.g., memory leaks) that need rejuvenation (e.g., through a reboot) or else they define aging as a property of code bases, that emerges as maintainers diverge from original requirements and architecture. It should be noted that these two definitions, although using the same terminology and referring to change over time, are substantively different. Addressing the needs of long-running systems is a relatively narrow perspective with a focused set of concerns, while divergence from an original state is a broad and ongoing consideration.

Approaching aging from the narrow long-running-systems perspective, S57 finds that most models for studying aging are Markov-based—i.e., they model the likelihood of an aging failure based on recent history and randomization. However, measurement-based approaches taking advantage of time-series analysis on aging indicators from system-level variables are also in use. ML-based techniques using large numbers of system-level variables and threshold-based approaches which monitor aging indicators and trigger a rejuvenation action when the threshold is surpassed are also common.

S57 emphasized a need to connect aging and rejuvenation studies to real systems, including safety-critical systems. This is especially important for techniques that allow detection of aging problems during software design and validation rather than requiring long run-times. There appears to be some progress with respect to connecting this line of work with practitioner needs. S22 found that although aging and rejuvenation work in the 1990s was purely analytical, the proportion of studies engaged in empirical or analytical-empirical hybrid work has increased substantially since then. S46 identified embedded systems as an important target for future research. This field is also responsive to the existence of dedicated publication venues: S46 observed that publications in this topic were generally increasing but that in a year when the Workshop on Software Aging Research was not held, the number of relevant publications plummeted to 1, from 15 the year prior and 25 the year after.

Taking up the latter definition of aging as divergence from an original state, S6 places

aging under the heading of code decay. For S6, code decay is composed of architectural flaws (including smells, erosion, and general decay), design/code flaws (also including smells, erosion, and general decay) and software aging. S6 enumerates the many metrics used for decay detection in the literature, including increases in change span, increases in the number of classes, increases in the values of coupling metrics (e.g., the average number of classes used per class in a package, called coupling between objects, or CBO) between versions, increases in the number of God Classes, Data Classes, or Brain Classes, increases in the number of code smells (e.g., Shotgun Surgery, Feature Envy), as well as increases in the number of design rule violations.

Focus on Dependability

We identified eight reviews that defined quality as dependability and robustness—i.e., the consistent operation of software despite the occurrence of faulty input or irregular operating conditions. Most robustness studies focused on the design / implementation and verification / validation development phases rather than considering the requirements engineering and maintenance processes [S39].

S70 identified some reliability-specific process and product-level metrics in use: number of defects, defect rate, mean time to failure, mean time to repair, and transaction time.The authors of S39 expressed concern about the lack of empirical validation work on this topic, a finding consistent with S70. Most studies were solution proposals and tended to emphasize method; 65% had weak or no evaluation. S39 and S17 observed that a majority of papers were academic experiments, conducted in lab settings, with relatively few examining real-world case studies such as open source or industrial deployments. S50 also observed a disconnection from practice and a tendency toward theoretical work, observing that a lack of public experimental datasets may be partially to blame.

S19 examined 27 studies of testability and robustness and found that testability analysis offers multiple types of insight into robustness. Testability incorporates error propagation analysis, assessment of exception handling, and implementation of strict rules for design and

implementation. S40 examined studies developing and evaluating metrics for testability as a facet of software quality and found that the most common metrics in use considered traits of observability (the ability to determine if an input influences output) and controllability (the ease of producing a specific output given a specific input).

Future researchers may find the research taxonomies proposed in S17 and S50 useful. Although both used keyword aggregation, S17 focused on a taxonomy of modeling approaches, while S50 proposes a taxonomy that includes research into both testing, assessment, and analysis as well as modeling.

Yet again, we also observed concern about the measures in use: S39 expressed concern about the lack of empirical validation work, observing that 65% of the articles they reviewed reported weak or no evaluation. Instead, most studies were solution proposals and tended to emphasize method, a finding consistent with S70. Several authors also commented on the lack of connection between practice and theory. S39 and S17 observed that a majority of papers were academic experiments, conducted in lab settings, with relatively few examining real-world case studies such as open source or industrial deployments. S50 also observed a lack of direct relevance to practice and a tendency toward theoretical work, suggesting that a lack of public experimental datasets may be partially to blame. S38 observed a lack of sufficient detail about the methods used for the measurement. Only 25% of the security evaluation methods they reviewed included sufficient detail for the method to be applied by others, and only a few reported validity measures such as reliability and accuracy.

Finally, S38 discussed the question of security as a measurable dimension of software quality, examining 16 papers concerning security evaluation. Although S38 was the only review that took this approach, the potential of a security-informed approach to quality seems high. S38 observed a strong relationship between security evaluation studies and practical engineering concerns and found that security evaluation studies tended to use some of the same software metrics as quality studies. Given the large amount of work being done in software security overall, we take the lack of security studies in our quality studies corpus to be a reflection of disciplinary boundaries. This may represent opportunities for deeper

collaboration between software engineering researchers concerned with quality and computer science researchers concerned with security.

D.4.3 Quality Research is Holistic

We found 18 secondary studies that examined quality from a holistic or multifaceted perspective. These studies either invoke broad or common-sense notions like system success, or referenced structured quality standards like those developed by the ISO. Two of the reviews we identified are broader syntheses that include a section on quality [S51, S67]. Five of the reviews we identified took up the task of quality definition, prediction, and modeling [S44, S33, S26, S3, S74]. Seven worked to identify specific metrics associated with a broad view of quality [S21, S24, S37, S45, S48, S55, S63]. Finally, four take up the particular question of whether user participation in development influences overall quality [S2, S7, S61, S65].

S33 and S67 sought to compare the relative quality of software based on differences in software production processes. In particular, they asked if the open code base and interest-driven software development process found in peer production/open source would be associated with better overall quality. S67 found that openness of code alone is not enough to produce quality: developer commitment (and action) is also required. S33 found that empirical work on this topic reached divergent conclusions depending on the measures and evaluation criteria used, suggesting that the answer to the question "is open source better?" is largely contingent.

Another common question animating holistic measurement studies asked: Does involving end users in system development increase the probability of success? Unsurprisingly, the answer based on these reviews is again contingent on measurement approach. If success is measured by users' attitudes toward the resulting software, then the answer seems to be yes. However, if success is measured by productivity outcomes, then the effect is inconsistent and smaller if present at all [S61, S65, S7]. That said, more recent studies tend to focus on attitudinal measures and HCI notions like ease of use, while older studies focus on productivity [S2]. This leaves open the question of whether studies of more recent development projects

would now find productivity gains associated with end-user involvement.

S33 and S3 considered the applicability of holistic models to open source and recommended that measures of quality should incorporate factors of the production community as well factors of the software artifact. However, S51 and S67 both found that most research on open source software quality focused only on artifacts. In S63's systematic review of quality metrics, 79% of the metrics they identified concerned software products, 12% operated at the software project level, and 9% concerned software processes.

The holistic perspective on quality found in secondary literature was not always well-served by the empirical literature on which it was built. S67 found that while many authors proposed holistic and multidimensional measures of quality, most empirical studies only took up a single dimension. S26 found that relatively few studies made use of multi-dimensional software quality models and advocated greater attention to the ISO 25010 or "SQuaRE" standard when considering software quality. This suggests substantial empirical work remains to be done, including holistic comparative studies incorporating variables from both artifactual and process-level perspectives.

D.5 Findings: Trends in Conducting and Publishing Secondary Reviews on Software Quality

In RQ2, we asked what evidence has been used in the measurement of software quality and what progress has been made. Although we summarized the current state of the field within each perspective, we argue that literature reviews themselves have an important role to play. Hence, as a complement to our answer to RQ2 elaborated in §D.4, we also conducted a quantitative analysis of the dissemination trends and research designs present in secondary reviews of software quality. We analyzed the timespan coverage of the reviews we identified, the venues in which they were published and their methods.

Figure D.2 offers a visualization of the range of publication dates for source articles reviewed by each secondary review as well as the publication dates for each of the systematic reviews. We use the reported date range when the authors include it. When they do not

report this information, we record the date range based on the date range we observe when sufficient bibliographic detail is reported; when neither is available, no date range appears. Note that this means some studies do not have a date range, and because a study might review a paper that is not officially published until later, it is possible for reviews to be published before the end of the time range covered.

The number of secondary studies has increased over the last two decades, with most studies published within the last 10 years. For each theme, at least one article in our dataset reviewed work published in the 1990s or before. This analysis provides evidence for a relatively robust research community, with both a continuing flow of primary findings and secondary syntheses.

The secondary quality studies we identified were published most often in journals (n=49) but also appeared in conferences (n=22), symposiums (n=3), and a workshop (n=1). Journals publishing secondary reviews were both discipline-wide and subfield-oriented. Those journals publishing more than one review are listed in Table D.5. The broad range of venues suggests that the topic is of high interest across the field of software engineering.

The secondary analyses we identified were conducted using a range of methods. Most of the studies identified described themselves as performing a systematic review of the literature (n =41) and often referred to Kitchenham et al. (2004) and Kitchenham et al. (2009). Other secondary studies identified their method as "grounded theory," a "survey," a "literature survey," a "meta-synthesis," or as a "multivocal literature review." Two identified their method as simply a "literature review" but met our criteria because they described following a systematic search process.

Secondary authors used a range of approaches to conduct synthesis and analysis. 62 studies categorized their identified primary articles into thematic categories, while 58 report additional statistics beyond the thematic categories they identified. Although only three articles describes themselves as a "meta analysis," authors of 6 reviews were able to conduct some form of secondary analysis of the data from the studies they identified. 9 studies offered a paragraph summary of each article they identified. This diverse range of publication

Table D.5: Venues publishing the most software quality reviews.

Venue	Count
Information and Software Technology	8
Journal of Systems and Software	8
IEEE Transactions on Software Engineering	3
Empirical Software Engineering	2
Empirical Software Engineering and Measurement (ESEM)	2
IEEE Int. Conference on Software Architecture (ICSA)	2
Int. Conference on Software Technologies (ICSOFT)	2
Journal of Software Maintenance and Evolution: Research and Practice	2

venues and relative consistency in approaches within the secondary review community provides evidence that secondary reviews are both providing value and achieving some degree of methodolological coherence. However, the highest standard of evidence through secondary analysis—meta-analysis—remains largely out of reach; more convergence in reporting standards and definitions in the primary research community would support progress in this regard.

Reproducibility is another hallmark of high-quality research. Several practices support reproducibility in secondary reviews: dataset publication, code publication as applicable, the in-text use of paper-specific identifiers (e.g. "S1") rather than only stating aggregate statistics such as "3 of 32 papers," (in this case we might ask, which 3 of which 32?) coupled with inclusion of a secondary bibliography that maps these identifiers back to the primary work.

Given the current efforts being made both to develop the field empirically and synthesize findings, what factors may be limiting progress for the software quality field ? One observation we made is that secondary reviews take on a diverse range of strategies for reporting source material. 67 studies included a list of all of their primary studies, of which 49 studies included what we describe as "all data"—not only a bibliographic record, but also sufficient detail on the codes and traits assigned to each record that the analysis could be reproduced. Some included these materials in tabular dataset form. 8 studies did not offer bibliographic records for the studies they identified and reported upon. This lack of transparency in secondary research undermines reproducibility, discourages future authors from building on the reviewers' efforts, and limits the ability of empirical researchers to use these reviews as guides.

D.6 Findings: Shared Challenges

We also conducted a thematic analysis of the challenges and gaps reported by authors of systematic literature reviews. These challenges span the three perspectives we identified.

D.6.1 Popular Measures of Questionable Validity

While many authors commented on the widespread use and availability of some measures, several in our corpus also cast doubt on the validity of these very measures. The popularity of length, complexity, and cohesion and coupling metrics drew comments from several authors [S21, S28, S31, S33, S43, S63, S70, S74]. These metrics are also commonly found in metrics analysis tools. Several of these authors raised concerns about the validity and utility of commonly used measures [S21, S28, S55, S63, S74]. S21 found that some metrics may indicate higher or lower quality, depending on how quality is defined, although However, S43 found several studies that validated the relationship between complexity and faults. For example, in the mapping study S55, the authors present a series of concerns about the theoretical validity of two long-standing metrics: the Lack of Cohesion of Methods and Coupling Between Objects metrics from Chidamber and Kemerer (C&K). This is a significant critique given that 24 of the 25 empirical evaluation studies S55 identified made use of them. Of the five metrics S37 found to be the most common in holistic studies, we observe that concerns about theoretical validity have been raised with four of them. Likewise, the four metrics that S70 found were used most frequently in reliability studies had been reported to have validatity issues. S55 also describes theoretical validity concerns with the Lines of Code metric as a quality measure, given its relationship to rate-based metrics such as those examining defect density.

These issues go beyond theoretical concerns. Multiple reviews have reported inconsistent empirical results with respect to quality metrics. To make sense of this conflict, S21 examined multiple facets of quality (including reliability and maintainability), and found that different metrics performed better for different facets. Many widely used measures were gauged significant less than 50% of the time, the least reliable being Number Of Children (NOC) and Depth of Inheritance (DIT), which in reliability studies were significant 29% and 25% of the time, respectively. Similarly, with respect to maintainability, S21 found that while some cohesion metrics offered useful results, DIT and NOC were much less reliable

and were significant in only 16% and 11% of datasets respectively. These troubled metrics are all widely available through metric extraction tools [S45]. Although the heavy use of questionable metrics like NOC and DIT in these metric extraction tools is a liability for the quality field, we suggest that tool improvement represents a ready solution.

D.6.2 Missing Vital Details in ML Studies

Machine learning methods are increasingly being used to measure quality Fernandez Del Carpio and Angarita, 2018 and studies using ML appear in our corpus from multiple perspectives [S5, S15, S20, S23, S25, S28, S62, S71]. However, several authors found that a lack of shared reporting standards for evidence and evaluation in ML is substantially hampering progress toward the use of ML to support evidence-driven engineering [S20, S23, S71].

For example, when S71 examined the state of the evidence for ML techniques in bug prediction, they began with 208 studies but found that only 36 passed their quality criteria. The excluded articles either lacked details on the context of the study or, despite claiming prediction as their goal, only conducted correlational analysis. Of the remaining 36, only 19 reported sufficient performance data to allow direct comparison across results, leading S71 to conclude that "the vast majority are less useful than they could be"(p. 1292). We want to highlight the missed opportunity S71 describes. Fewer than 10% of the empirical studies in this area articulated evaluation measures with detail sufficient to allow their results to be systematically compared to other studies.

D.6.3 Data Sources

Many authors remarked on the predominance of Java codebases as objects of study [S14, S21, S23, S28, S37, S52, S53, S71]. In fact, software quality research's empirical focus is even more narrow: Eclipse was the most common codebase in the studies S21, S23, and S71 reviewed, and JHotDraw was the most common codebase in the studies S52 reviewed. There are abundant tools to assess Java metrics and both Eclipse and JHotDraw have freely licensed public Java codebases [S53]. The growing body of work using similar tools, metrics,

and languages suggests the potential for future meta-analytical work. However, we observe that this concentration on Java and Eclipse may be a limitation on metric generalizability since effective metrics may vary by language and the ultimate purpose of the codebase, even within the OO paradigm.

Multiple studies remarked on the influence of the PROMISE repository, a set of public datasets released in 2005, as substantially increasing not only the amount of research investigating software quality, but also the use of ML techniques and the number of studies using public datasets [S15, S20, S21, S23, S27, S28, S37, S62, S64, S71]. However, as S20 describes, the use of PROMISE datasets may be problematic, given both a series of issues with their quality as well as limitations imposed by their age given that much of the data is derived from NASA projects in the 1970s-1990s. Several authors expressed concern that use of public data is too low since the results of proprietary studies are not repeatable [S21, S23, S28]. S28 found the number of studies using small datasets to be relatively high (33% of their sample).

The authors of S41 observed the importance of open science practices, including public validated datasets. Adopting an even more open approach may indeed be necessary for the field to progress. Given that so many findings are inconsistent across empirical settings, more durable conclusions may only be possible when data is sufficiently available to enable large-scale cross-project comparative work. Worryingly, Mahmood et al. (2018) found that of the 208 defects prediction studies identified through a widely cited SLR [S71], only 13 had attempted replication. Of those 13 studies, the replications agreed with the original in only 62% of the cases.

D.7 Discussion

D.7.1 The Three Perspectives on Quality

We identified three dominant perspectives on quality that are used by the software engineering research community. Taken together, these three form an ontology that can be used to

classify and characterize the development of future theoretical approaches to software quality measurement.

Perspectives emphasizing methodological concerns often orient toward one of two strategies: toward developing metrics (often static, structural metrics), or toward detecting violation of general rules of design (a heuristic-oriented approach). Metric-oriented approaches draw from classic engineering and manufacturing perspectives about defects as a predictable aspect of a product; areas of work include refining these calculations and can be readily integrated into development environment and analytic tools for display and control. Software engineering researchers taking up a heuristic-oriented approach articulate what makes some code more desirable (less "smelly") than other code, and then attempt to embed this wisdom into problem detection algorithms. These perspectives tend to emphasize the engineer, including their aesthetic preferences, sense of proficiency, and personal productivity. Methodologically-oriented quality research tends to treat user-oriented considerations as a desirable and assumed, but often unverified, after-effect (e.g., code that is coherent and well-formed is understood to satisfy developers and is assumed to be faster to adapt to new requirements, assumed to have fewer bugs, and so on).

Perspectives emphasizing outcomes tend to take up either or both of two key outcomes: maintainability, which centers the experience of developers after the code is released, and dependability, which centers the experience of software implementation sites. Maintainable code can be patched more efficiently, while dependable code performs its role in context with less need for fault-related patching. These factors are interrelated, since modern software production is not the completion of a single bounded project. Rather, it is treated as an ongoing cycle of development and release that proceeds over time. Outcome-oriented perspectives may make use of the efforts of more methodologically-oriented perspectives, drawing in measures that perform well as predictors of desirable outcomes.

Perspectives taking a more holistic approach examine not only the outcomes and metrics of the other two, lower-level perspectives, but also seek to be broad or all-encompassing. The mixing of diverse kinds of evidence allows for a more detailed explanation of how a

software product is performing in a given context, but does so potentially at the expense of comparability and generalizability. One potential avenue for growth here is the exploration of a more formal and deliberate approach to mixed-methods and multi-method research.

D.7.2 Roadblocks to Evidence-Driven Engineering

To what extent have more than two decades of software engineering research resulted in the support for evidence-driven software engineering practices? Secondary reviews depend in large part on the quality of the primary empirical studies on which they are based. Tertiary reviews such as this one are substantially dependent on the practices of both primary studies and secondary reviews. We note that authors of secondary works were able to draw the strongest conclusions when primary authors used similar measures and consistently reported statistics that support comparison across studies, such as AUC, sample size, and correlation strength. There are robust meta-analytical methods available and in use, perhaps most notably in medicine but also throughout the biological, physical, and social sciences. However, meta-analytical approaches require sufficient field-level agreement with respect to conceptualization, shared measures, and the reporting of descriptive statistics. Without more rigorous standards for evidence and reporting, the highest quality of evidence for practice—meta-analysis—will remain largely out of reach for the software quality field.

We observe that secondary reviews have found multiple outlets for publication and used diverse methods under the overall umbrella articulated in the work of Kitchenham and colleagues to draw their conclusions—but some have failed to live up to the promise of their genre and methods by a lack of detail about the papers on which the reviews are built. Likewise, secondary studies that observe that a metric was used often without weighing the degree to which it has been validated may contribute to a misunderstanding of the nature and relative weight of the evidence. Authors and reviewers of systematic literature reviews should weigh to what extent the evidence reporting practices they follow and accept are able to support the development of evidence-based software engineering.

D.7.3 Toward a Theory of Software Quality

Despite the relative maturity of software quality research as a field, the growth of analytical tools, the numerous methods applied, and the repeated re-evaluation of empirical work over decades, empirical software quality research is frequently disconnected and in conflict. When empirical work is failing to bear consistent fruit, it may be, as Kurt Lewin offered, that "there is nothing as practical as a good theory." To support the articulation of a more coherent theory of quality, we offer a series of observations about what components such a theory might contain based on the current state of the field as viewed through secondary reviews.

Our thematic analysis provides evidence that software quality is most typically discussed at three levels—the methodological, the outcome-focused, and the holistic. Each of these levels offers a set of risks and challenges. The items in each level are also connected to the other levels, at times in a tangle of untheorized and unexplored ways. We suggest that by placing empirical software quality work into this framework, we can understand how apparently contradictory research on software quality may, in fact, be complementary. By calling our attention to how we are conceptualizing and measuring quality, we can identify things that we have taken for granted in previous research and identify next steps.

There is wide recognition within software engineering research that quality is multifaceted. Certain interventions to improve quality may increase quality in one dimension while decreasing it in another (e.g., attempts to make software more reliable might decrease performance). What is foregrounded less frequently is the fact that a multifaceted approach to conceptualizing quality means that desirable dimensions of quality can be, and often will be, in direct conflict. When it comes to software quality, we simply cannot always have it all. Future work should map out the relationships and conflicts between different facets of quality identified in this tertiary review and in the rest of the literature.

A more fundamental challenge is that individual dimensions of quality (e.g., dependability) will always be operationalizable into multiple conceptually valid measures. As we have described, there are many reasonable ways to approach measuring a quality dimension such

as maintainability, reliability, or dependability. Sometimes, these different measures will conflict with each other. In some cases, this may reflect a lack of conceptual clarity. For example, the fact that different measures of smell and technical debt can point in such different directions is evidence that the concepts of code smell and technical debt may be overloaded. The software engineering literature has not consistently or clearly mapped specific concepts to measures. It can do so much better. Discussion of "validation" will be misleading unless we specify the underlying measure(s) against which a concept or an intervention is being validated. ML tools can allow for the faster assessment of large volumes of code, but cannot relieve the challenges of conceptual validity.

The lack of validation of widespread measures of code smells and technical debt in terms of key outcomes reflects the most important example of low hanging fruit in software quality research. Software engineering researchers can articulate the relationships between widely used measures and overarching concepts. Each relationship can serve as a theoretical proposition and a testable hypothesis. There is no consensus about what, specifically, code smells are bad for: Will smelly software be buggier? Will it lead to lower levels of end-user satisfaction? Will it be more expensive to maintain? Will it lead to unhappy engineers and higher developer turnover? Once those expected outcomes are articulated, it is possible to begin to assess relative effect sizes. Which smells have the most substantive influence and which are mere inconveniences? Doing so will help us identify not only what works and what does not work, but also the reasons for success and failure. In addition, this approach brings assumptions and tacit definitions to the surface, supporting the identification of the underlying conceptual models behind software quality. Ultimately, software quality is not a thing in the world about which there can be some measure of ground truth. Software quality is contextual and must always be framed relationally—that is, as quality in what circumstance, according to whom, and by what criteria.

Of importance is the recognition that no measure can be expected to perfectly operationalize any concept. No approach to bug identification or tracking will result in a complete or unbiased measure of faultiness. No survey can fully capture how end users feel about

a piece of software. Although we may hunt for metrics that can make code "fault-prone," developers will always discover ways to work to the metrics in ways that are at least partially at odds the desired underlying outcome. Using multiple metrics can help but is no panacea. Instead, it may be more useful to examine the various leadership, management, and development practices of teams producing highly successful software to better understand how quality is achieved.

D.7.4 Implications for Software Engineering Research

These findings have implications for early-stage software engineering research in three respects. First, we encourage consideration of how a current project might fall into the three perspectives we identified (methodological, outcome-focused, and holistic). Next, the researcher might consider the challenges and gaps we have identified. Finally, in light of the classification and challenges, the researcher might weigh opportunities to innovate in two dimensions—both within the current perspective and with respect to the other two perspectives. For example, a study focused on maintainability outcomes might weigh the relationship to other outcomes while also considering the validity and origin of the methodological approaches on which their work is built, as well as the more holistic perspective that their work might ultimately feed into.

A second line of implications is for the review process. We encourage peer reviewers to weigh not only how a given empirical project has been executed with respect to disciplinary standards—including shared challenges about validity—but also whether the work has reported sufficient detail to enable others to build on the effort.

D.7.5 Implications for Software Engineering Practice

Practitioners interact with each of the three perspectives we identified in the quality software field. The methodological perspective speaks directly the same metrics that appear in analytical tools. We encourage software engineers to interrogate the key metrics by which their work is assessed; the presence of a measure in an analytical package is no guarantee

that the measure is meaningful, but research on these measures is a useful guide to the evidence as developed so far. The outcome-focused perspective on software quality speaks to the trade-offs engineers face in software design and architecture between competing priorities and competing facets of quality; the primary implication of our work here is to emphasize the importance of continuing to indeed view these as trade-offs, to be articulated and explored with evidence. It may be that a satisfying holistic evaluation of quality is elusive—instead, it may be more valuable to focus on outcomes and on the underlying heuristics and measures on which a more holistic view can be built in context. In a given project, where is technical debt building up and going unacknowledged? Which types of bugs are proving the most elusive? Which measures point to something real and which are dashboard fodder? How do their expert assessments compare with the best evidence developed by the empirical engineering community? For all of the above topics, we point to the value of secondary reviews as a source of guidance on the status of the quality research field.

D.8 Threats to Validity

One key limitation to this study is imposed by our method: by focusing on secondary studies, we omit lines of investigation that have not yet attracted secondary reviews. Our work does not quantify the relative weight of empirical evidence with respect to particular findings, given that our collection of secondary studies likely summarizes some of the same studies. Our object is to synthesize the results of analyses that weigh this evidence. To the extent that multiple authors arrived at similar conclusions given sets of empirical papers that only partially overlap, we argue that these shared observations of common challenges and flaws are highly credible.

The volume of available literature and the polysemy of relevant terms may limit our ability to be comprehensive. We have sought to mitigate this risk by using both diverse digital libraries for our preliminary search and the reverse snowballing technique. By omitting work published in predatory outlets, self-published materials, and gray literature as a way to filter for quality, we may have neglected useful contributions or offered an incomplete picture of

researcher interest. Given that we found a substantial volume of studies, we take this risk to be minimal and believe it is outweighed by the gain in terms of validity. Furthermore, our finding of diverse publication venues suggests that there are many outlets for high-quality reviews.

Finally, our search and review process may have introduced some unconscious bias into our results; we selected keywords based on our own understanding of the field, and although we iterated on keywords, we were inevitably limited; further, we made determinations of which studies were concerned with contexts we gauged as too narrow for our purposes (e.g., automotive software) versus those contexts we gauged sufficiently broad (e.g., object-oriented code). We sought to diminish the impact of this bias through the reverse snowball process.

D.9 Conclusion

Empirical engineering challenges us to use evidence to drive practice—and we argue that this should apply to research as well as software development. This study synthesizes current knowledge about how software quality can be measured and summarizes current understanding of how the concept of quality is being defined and determined. We found no easy answer: software quality research is a highly active field of inquiry, engaging with definitional, tactical, and strategic challenges. The field has made substantial progress but remains far from achieving consensus. Lack of validated empirical work and gaps in statistical methods and reporting is a persistent concern. ML techniques offer exciting new strategies for tackling the data-intensive nature of software development, but generally require valid training data. Studies taking on more comprehensive or holistic assessments and engaging in comparative work, if accompanied by validation, are also needed. Some long-standing metrics examining complexity and cohesion have demonstrated their empirical validity for specific facets of quality, but attempts to validate many others have served to illustrate the complex trade-offs engineers must make when they design, implement, and maintain software.

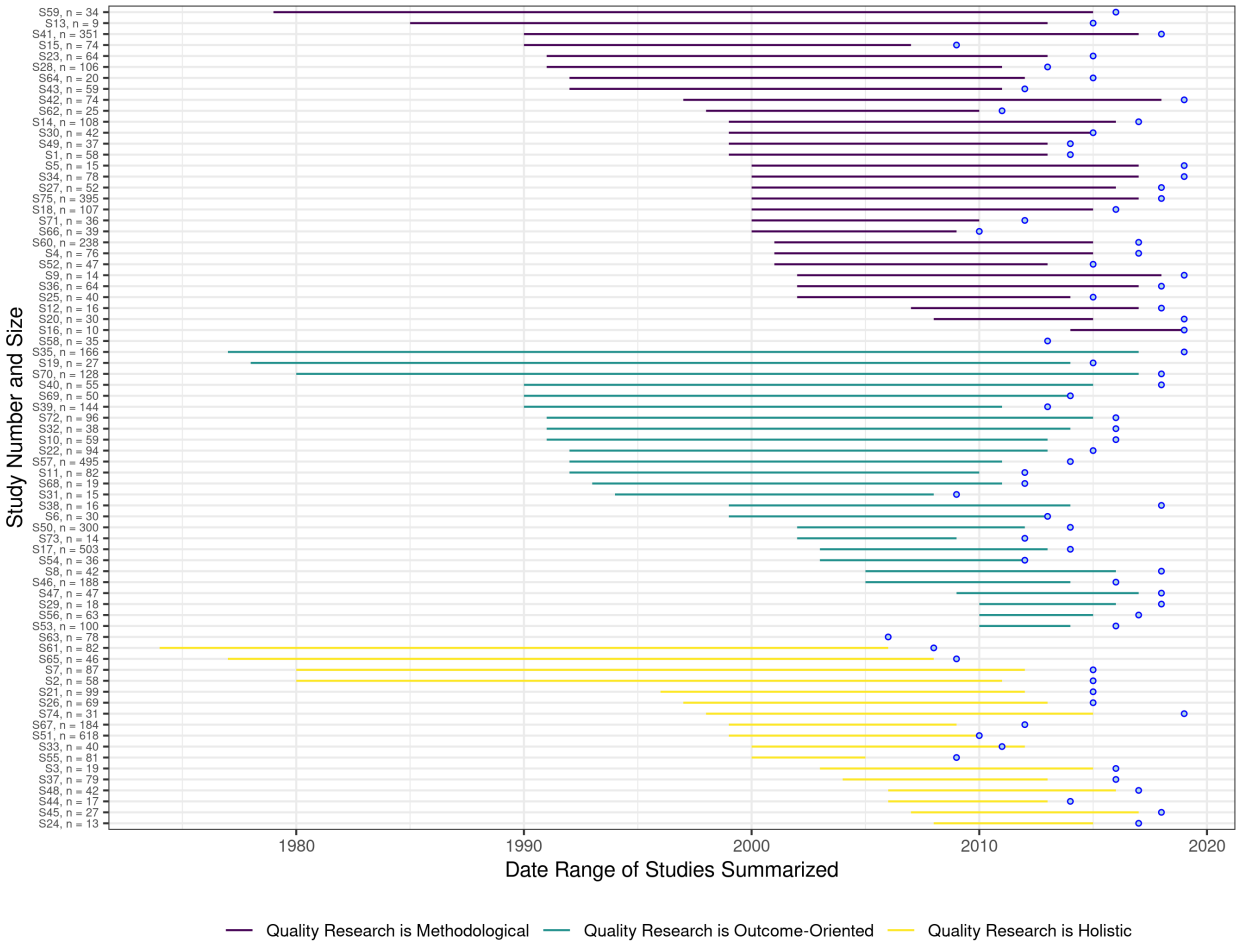
Software quality research addresses a vital topic of concern to our economic, political, and social life. Evidence-driven software engineering needs measurement strategies that are

both actionable and valid. As development practices continue to evolve and new design challenges emerge, the challenge for the field is to remain both sufficiently grounded in the day-to-day realities of practical development and sufficiently oriented by robust theoretical work and empirical analysis to offer evidence to guide that development.

Acknowledgment for Appendix D

University of Washington librarians Jessica Albano and Mel DeSart provided helpful advice in the planning of this project. Members of the Community Data Science Collective provided excellent advice on an early draft of this project, including Jeremy Foote, Floor Fiers, Wm Salt Hale, Sohyeon Hwang, Charles Kiene, Aaron Shaw, and Nathan TeBlunthuis. Attendees at LinuxFest Northwest and the Seattle Area GNU/Linux conference provided valuable feedback throughout this project. The authors gratefully acknowledge support from the Alfred P. Sloan Foundation through the Ford/Sloan Digital Infrastructure Initiative, Sloan Award 2018-11356.

Figure D.2: Visualizing secondary study coverage. Each of the 75 secondary studies we identified are represented as a line. A blue dot indicates year of publication, segment color indicates perspective on quality, and segment length indicates the duration of the primary studies they examine. Y-axis labels give the study identifier and the number of studies examined by each.



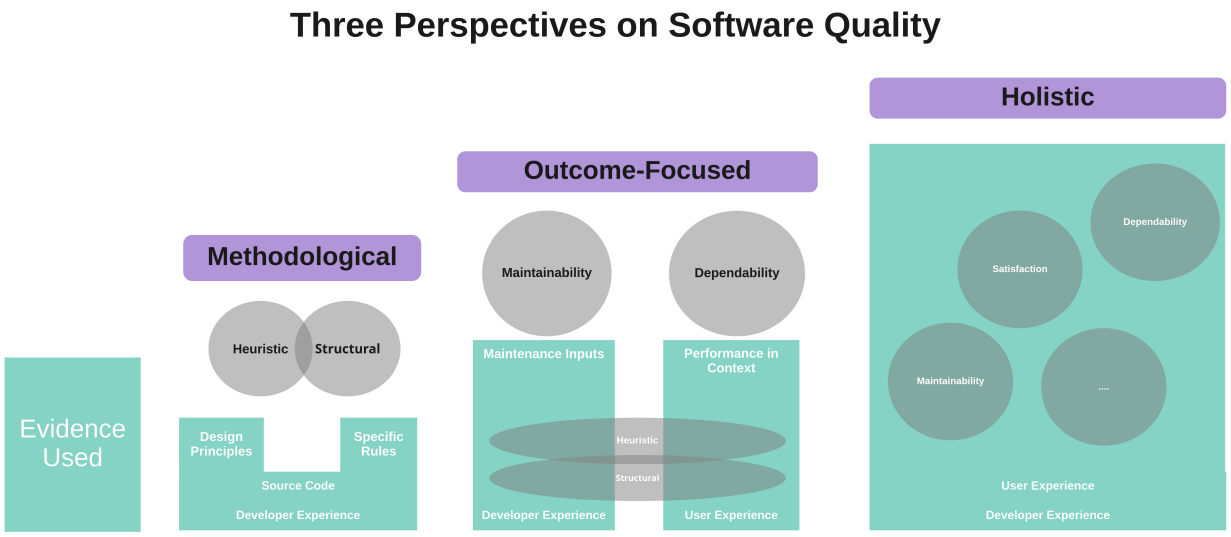


Figure D.3: The three perspectives we identified operate at progressive layers of abstraction, from methods-oriented, to outcome-oriented, to making broad or holistic assessments. The evidence at each layer is often cumulative with respect to previous layers.

BIBLIOGRAPHY

[S1] M. Abebe and C. J. Yoo, "Trends, opportunities and challenges of software refactoring: A systematic literature review," *International Journal of Software Engineering and its Applications*, vol. 8, no. 6, pp. 299–318, 2014.

[S2] U. Abelein and B. Paech, "Understanding the Influence of User Participation and Involvement on System Success – a Systematic Mapping Study," *Empirical Software Engineering*, vol. 20, no. 1, pp. 28–81, Feb. 2015.

[S3] A. Adewumi, S. Misra, N. Omoregbe, B. Crawford, and R. Soto, "A systematic literature review of open source software quality assessment models," *SpringerPlus*, vol. 5, no. 1, pp. 1–13, Nov. 2016.

[S4] 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, 2018.

[S5] M. I. Azeem, F. Palomba, S. Lin, and W. Qing, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–38, Apr. 2019.

[S6] A. Bandi, B. J. Williams, and E. B. Allen, "Empirical evidence of code decay: A systematic mapping study," in *Proceedings - Working Conference on Reverse Engineering, WCRE*, 2013, pp. 341–350.

[S7] M. Bano and D. Zowghi, "A systematic review on the relationship between user involvement and system success," *Information and Software Technology*, vol. 58, pp. 148–169, 2015.

[S8] T. Besker, A. Martini, and J. Bosch, "Managing architectural technical debt: A unified model and systematic literature review," *Journal of Systems and Software*, vol. 135, pp. 1–16, Jan. 2018.

[S9] J. Bogner, T. Boceck, M. Popp, D. Tschechlov, S. Wagner, and A. Zimmermann, "Towards a Collaborative Repository for the Documentation of Service-Based Antipatterns and Bad Smells," in *2019 IEEE International Conference on Software Architecture - Companion, ICSA-C 2019, March 25, 2019 - March 29, 2019*. pp. 95–101.

[S10] D. Bombonatti, M. Goulão, and A. Moreira, "Synergies and tradeoffs in software reuse – a systematic mapping study," *Software - Practice and Experience*, vol. 47, no. 7, pp. 943–957, 2016.

[S11] H. P. Breivold, I. Crnkovic, and M. Larsson, "A systematic review of software architecture evolution research," *Information and Software Technology*, vol. 54, no. 1, pp. 16–40, 2012.

[S12] A. S. Cairo, G. d. F. Carneiro, and M. P. Monteiro, "The impact of code smells on software bugs: A systematic literature review," *Information*, vol. 9, no. 11, 2018.

[S13] C. Carroll, D. Falessi, V. Forney, A. Frances, C. Izurieta, and C. Seaman, "A Mapping Study of Software Causal Factors for Improving Maintenance," in *International Symposium on Empirical Software Engineering and Measurement*, November, 2015, pp. 235–238.

[S14] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, Apr. 2018.

[S15] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert Systems with Applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[S16] M. A. Zaidi and R. Colomo-Palacios, "Code smells enabled by artificial intelligence: a systematic mapping," in *Computational Science and Its Applications - ICCSA* 2019. pp. 418–27.

[S17] F. Febrero, C. Calero, and M. A. Moraga, "A systematic mapping study of software reliability modeling," *Information and Software Technology*, vol. 56, no. 8, pp. 839–849, 2014.

[S18] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *EASE '16 Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, 2016.

[S19] M. M. Hassan, W. Afzal, M. Blom, B. Lindstrom, S. F. Andler, and S. Eldh, "Testability and Software Robustness: A Systematic Literature Review," in *Proceedings - 41st Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2015*, 2015, pp. 341–348.

[S20] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," *IEEE Transactions on Software Engineering*, vol. 45, no. 2, pp. 111–47, Feb. 2019.

[S21] R. Jabangwe, J. Borstler, D. Smite, and C. Wohlin, "Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review," *EMPIRICAL SOFTWARE ENGINEERING*, vol. 20, no. 3, pp. 640–693, Jun. 2015.

[S22] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015.

[S23] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Applied Soft Computing Journal*, vol. 27, pp. 504–518, 2015.

[S24] G. Me, G. Procaccianti, and P. Lago, "Challenges on the Relationship between Architectural Patterns and Quality Attributes," in *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA* pp. 141–144.

[S25] J. Murillo-Morera, C. Quesada-López, and M. Jenkins, "Software fault prediction: A systematic mapping study," in *CIBSE 2015 - XVIII Ibero-American Conference on Software Engineering*, 2015, pp. 446–459.

[S26] S. Ouhbi, A. Idri, J. L. Ferńandez-Alemán, and A. Toval, "Predicting software product quality: A systematic mapping study," *Computacion y Sistemas*, vol. 19, no. 3, pp. 547–562, 2015.

[S27] R. Ozakinci and A. Tarhan, "Early software defect prediction: a systematic map and review," *Journal of Systems and Software*, vol. 144, pp. 216–39, Oct. 2018.

[S28] D. Radjenovic, M. Hericko, R. Torkar, and A. Zivkovic, "Software fault prediction metrics: A systematic literature review," *INFORMATION AND SOFTWARE TECHNOLOGY*, vol. 55, no. 8, pp. 1397–1418, Aug. 2013.

[S29] S. M. Ramirez, K. Cortes, J. O. Ocharan-Hernandez, and A. J. Sanchez Garcia, "Software stability: A systematic literature review," in *6th International Conference in Software Engineering Research and Innovation, CONISOFT* 2018. pp.109–115.

[S30] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 867–895, 2015.

[S31] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *3rd International Symposium on Empirical Software Engineering and Measurement, ESEM*, 2009, pp. 367–377.

[S32] L. F. Ribeiro, M. A. F. De Farias, M. Mendonça, and R. O. Spínola, "Decision criteria for the payment of technical debt in software projects: A systematic mapping study," in *ICEIS - Proceedings of the 18th International Conference on Enterprise Information Systems*, vol. 1, 2016, pp. 572–579.

[S33] C. Ruiz and W. N. Robinson, "Measuring Open Source Quality: A Literature Review," *International Journal of Open Source Software and Processes*, vol. 3, no. 3, pp. 48–65, Jul. 2011.

[S34] F. Sabir, F. Palma, G. Rasool, Y. G. Gueheneuc, and N. Moha, "A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems," *Software: Practice and Experience*, vol. 49, no. 1, pp. 3–39, Jan. 2019.

[S35] M. Salama, R. Bahsoon, and P. Lago, "Stability in Software Engineering: Survey of the State-of-the-Art and Research Directions," *IEEE Transactions on Software Engineering*, 2019.

[S36] J. A. M. Santos, J. B. Rocha-Junior, L. C. Lins Prates, R. S. do Nascimento, M. F. Freitas, and M. G. de Mendonca, "A systematic review on the code smell effect," *Journal of Systems and Software*, vol. 144, pp. 450–477, Oct. 2018.

[S37] M. Santos, P. Afonso, P. H. Bermejo, and H. Costa, "Metrics and statistical techniques used to evaluate internal quality of object-oriented software: A systematic mapping," in *Proceedings - International Conference of the Chilean Computer Science Society, SCCC*, 2016.

[S38] S. Sentilles, E. Papatheocharous, and F. Ciccozzi, "What do we know about software security evaluation? A preliminary study," in *CEUR Workshop Proceedings*, vol. 2273, 2018, pp. 44–51.

[S39] A. Shahrokni and R. Feldt, "A systematic review of software robustness," *Information and Software Technology*, vol. 55, no. 1, pp. 1–17, 2013.

[S40] R. Sharma and A. Saha, "A systematic review of software testability measurement techniques," in *2018 International Conference on Computing, Power and Communication Technologies, GUCON* 2018, pp. 299–303.

[S41] E. V. d. P. Sobrinho, A. De Lucia, and M. d. A. Maia, "A systematic literature review on bad smells—5 W's: which, when, what, who, where," *IEEE Transactions on Software Engineering*, 2018.

[S42] A. Kaur, "A Systematic Literature Review on Empirical Analysis of the Relationship Between Code Smells and Software Quality Attributes," *Archives of Computational Methods in Engineering*, 2019.

[S43] A. Tahir and S. G. MacDonell, "A systematic mapping study on dynamic metrics and software quality," in *IEEE International Conference on Software Maintenance, ICSM*, 2012, pp. 326–335.

[S44] O. Franco-Bedoya, D. Ameller, D. Costal, and X. Franch, "QuESo: A quality model for open source software ecosystems," in *ICSOFT-EA - Proceedings of the 9th International Conference on Software Engineering and Applications*, 2014, pp. 209–218.

[S45] K. Valenca, E. D. Canedo, and R. M. Da Costa Figueiredo, "Construction of a Software Measurement Tool Using Systematic Literature Review," IEEE 2018 International Congress on Cybermatics, pp. 1852–1859.

[S46] N. A. Valentim, A. Macedo, and R. Matias, "A Systematic Mapping Review of the First 20 Years of Software Aging and Rejuvenation Research," in *Proceedings - IEEE 27th International Symposium on Software Reliability Engineering Workshops, ISSREW* 2016. pp. 57–63.

[S47] R. Verdecchia, I. Malavolta, and P. Lago, "Architectural technical debt identification: The research landscape," in *Proceedings - International Conference on Software Engineering*, 2018, pp. 11–20.

[S48] T. Wahyuningrum and K. Mustofa, "A systematic mapping review of software quality measurement: Research trends, model, and method," *International Journal of Electrical and Computer Engineering*, vol. 7, no. 5, pp. 2847–2854, 2017.

[S49] M. Abebe and C.-J. Yoo, "Classification and Summarization of Software Refactoring Researches: A Literature Review Approach," in *Advanced Science and Technology Letters*. Science & Engineering Research Support soCiety, Apr. 2014, pp. 279–284.

[S50] J. Xavier, A. Macêdo, R. Matias, and L. Borges, "A survey on research in software reliability engineering in the last decade," in *Proceedings of the ACM Symposium on Applied Computing*, 2014, pp. 1190–1191.

[S51] A. Aksulu and M. Wade, "A Comprehensive Review and Synthesis of Open Source Research," *Journal of the Association for Information Systems*, vol. 11, no. 11/12, pp. 576–656, Nov. 2010. https://aisel.aisnet.org/cgi/viewcontent.cgi?article=1552&context=jais

[S52] J. Al Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and Software Technology*, vol. 58, pp. 231–249, Feb. 2015.

[S53] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology*, vol. 70, pp. 100–121, Feb. 2016.

[S54] B. A. Orenyi, S. Basri, and L. T. Jung, "Object-Oriented Software Maintainability Measurement in the Past Decade," in *2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*. Kuala Lumpur, Malaysia: IEEE, Nov. 2012, pp. 257–262.

[S55] B. Kitchenham, "What's up with software metrics— A preliminary mapping study," *Journal of Systems and Software*, vol. 83, no. 1, pp. 37–51, 2009.

[S56] C. Fernández-Sánchez, J. Garbajosa, A. Yagüe, and J. Perez, "Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study," *Journal of Systems and Software*, vol. 124, pp. 22–38, Feb. 2017.

[S57] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "A survey of software aging and rejuvenation studies," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 10, no. 1, pp. 1–34, Jan. 2014.

[S58] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, Jun. 2013.

[S59] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Antipattern and Code Smell False Positives: Preliminary Conceptualization and Classification," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Mar. 2016, pp. 609–613.

[S60] S. Singh and S. Kaur, "A systematic literature review: Refactoring for disclosing code smells in object oriented software," *Ain Shams Engineering Journal*, vol. 9, no. 4, pp. 2129–2151, Dec. 2018.

[S61] J. He and W. R. King, "The Role of User Participation in Information Systems Development: Implications from a Meta-Analysis," *Journal of Management Information Systems*, vol. 25, no. 1, pp. 301–331, Jul. 2008.

[S62] R. Malhotra and A. Jain, "Software fault prediction for object oriented systems: a literature review," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 5, p. 1, Sep. 2011.

[S63] O. Gómez, H. Oktaba, M. Piattini, and F. García, "A Systematic Review Measurement in Software Engineering: State-of-the-Art in Measures," in *ICSOFT*, 2006. pp. 165–176.

[S64] P. K. Singh, D. Agarwal, and A. Gupta, "A systematic review on software defect prediction," *IEEE INDIACom* 2015.

[S65] Z. Bachore and L. Zhou, "A critical review of the role of user participation in IS success," in *AMCIS 2009 Proceedings*, 2009. http://aisel.aisnet.org/amcis2009/659

[S66] M. Zhang, T. Hall, and N. Baddoo, "Code Bad Smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 3, pp. 179–202, Oct. 2010.

[S67] K. Crowston, K. Wei, J. Howison, and A. Wiggins, "Free/Libre open-source software development: What we know and what we do not know," *ACM Computing Surveys*, vol. 44, no. 2, pp. 1–35, Feb. 2012.

[S68] E. Tom, A. Aurum, and R. Vigden, "A Consolidated Understanding of Technical Debt," in *ECIS 2012 Proceedings*, 2012.

[S69] P. K. Singh, O. P. Sangwan, A. Pratap, and A. P. Singh, "An Analysis on Software Testability and Security in Context of Object and Aspect Oriented Software Development," *International Journal of Information Security and Cybercrime*, no. 1, pp. 17–28, 2014.

[S70] E. Ronchieri and M. Canaparo, "Metrics for Software Reliability: a Systematic Mapping Study," *Journal of Integrated Design & Process Science*, vol. 22, no. 2, pp. 5–25, Apr. 2018.

[S71] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A Systematic Literature Review on Fault Prediction Performance in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, Nov. 2012.

[S72] R. Malhotra and A. Chug, "Software Maintainability: Systematic Literature Review and Current Trends," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, no. 8, pp. 1221–1253, 2016.

[S73] A. B. M. Sultan, A. D. Bakar, H. Zulzalil, and J. Din, "Systematic literature review in open source software maintainability," *International Review on Computers and Software*, vol. 7, no. 5, pp. 2200–2205, 2012.

[S74] M. Yan, X. Xia, X. Zhang, L. Xu, D. Yang, and S. Li, "Software quality assessment model: a systematic mapping study," *Science China Information Sciences*, vol. 62, no. 9, 2019.

[S75] K. Alkharabsheh, Y. Crespo, E. Manso, and J. A. Taboada, "Software Design Smell Detection: a systematic mapping study," *Software Quality Journal*, 2018.

Appendix E

INSTITUTIONAL STATEMENTS AS A DRIVER FOR OSS PROJECT SUCCESS: A BAYESIAN REPLICATION AND EXTENSION OF YIN ET AL.'S "OPEN SOURCE SOFTWARE SUSTAINABILITY"

This chapter takes a step away from underproduction per se, and instead explores an alternate approach to thinking about digital infrastructure communities and their sustainability. This work replicates and extends descriptive work examining these communities by making use of an explicitly causal and intervention-oriented perspective. This allows us to measure the anticipated effect of mentorship as an intervention into the evolution of a community into an organization able to meet the Apache Software Foundation's criteria for inclusion under their umbrella.

This chapter has been circulated as a working paper:

Champion, Kaylea and Hill, Benjamin Mako (2024) "Institutional Statements as a Driver for OSS Project Success: A Bayesian Replication and Extension of Yin et al.'s 'Open Source Software Sustainability.' "

I led all parts of this project.

E.1 Introduction

Modern communication, education, commerce, and research all depend on a complex and decades-old collaborative effort to build and share open source software (Champion & Hill, 2021). The ubiquity and quality of open source software means it serves as infrastructure in many contexts. Making this infrastructure sustainable is a critical challenge (Eghbal, 2016). Although writing code and supporting users is part of building infrastructure software,

some of the key challenges these projects face are social and organizational rather than technical (Eghbal, 2020). Open source communities are self-organized, with participants choosing their tasks, developing their novel forms of governance, and establishing their own rules and decision-making processes. L. Yin et al. (2022) take an institutional analysis and sociotechnical network approach to understanding open source software organizations.

Our project seeks to replicate and extend the work done in L. Yin et al. (2022). In §E.2, we describe L. Yin et al. (2022) in further detail.We describe both the methods used in L. Yin et al. (2022) and our approach to extend their work in §E.3. The results of our analysis are detailed in §E.4, with a discussion of implications in §E.5. We articulate key limitations in §E.6 before concluding in §E.7.

E.2 Background

To understand "organizational and social factors associated with a sustainable free software development organization", L. Yin et al. (2022) use records from the ASF's incubator process (N. Yang et al., 2022), in which projects seeking to operate under the ASF umbrella receive mentorship and support. The ASF is a long-standing software development community that is responsible for not only the development of the Apache web server but also modern file, application, database, and high-performance computing technologies such as Arrow, Hive, Hadoop, Struts, and Spark. Projects *graduate* from the ASF's incubation program based on a vote informed by whether their project has developed a sustainable community, demonstrated good governance under a consensus-driven process, adopted an Apache license, and produced a code base of sufficient quality.[1]

E.2.1 The Yin et al. Analysis

L. Yin et al. (2022) take a sociotechnical analysis in their attempt to understand why some projects *graduate* from the ASF incubator, while others do not. To do so, L. Yin et al.

[1]https://incubator.apache.org/

(2022) apply the Institutional Analysis theoretical framework developed by Nobel laureate in economics Elinor Ostrom and perspectives informed by science and technology studies (STS). They tackle three research questions using computational methods.

First, they evaluate the viability of using automated approaches on digital trace data to detect whether project participants are making *institutional statements*—that is, statements about "rules and norms...which we define as a shared linguistic constraint or opportunity that prescribes, permits, or advises actions or outcomes for actors (both individual and corporate)" [p. 4]. L. Yin et al. (2022) hand-annotate data to train a BERT classifier which they use to identify *institutional statements*. Having developed a dataset of statement counts per group, they use Mann-Whitney U tests to assess the statistical significance in the number of statements made by projects that *graduated* versus those that did not.

The second research question in L. Yin et al. (2022) concerns whether changes in project structure are consistent with predictions from Institutional Analysis. They assess this question using the same digital trace data sources. They characterize discourse by fitting an LDA topic model to the institutional statements they identified. They find a statistically significant difference in the incidence of 10 of the 12 topics using a Mann-Whitney U test. L. Yin et al. (2022) also offer a visualization of the changes in each topic's relative prevalence over time and discussion of the temporal trends.

The third research question concerns the ordering of changes—that is, the extent to which changes in the rate of institutional statements precede social and technical network change and vice versa. They answer this question by constructing a weighted directed social network in which contributors are nodes and edges exist whenever a contributor has replied to another person's message. They also construct an undirected bipartite network with contributors and files in which edges are drawn when a developer contributes to a file. L. Yin et al. (2022) use these networks to conduct a Granger analysis, which identifies whether pairs of variables consistently precede or follow changes in one another, thereby meeting one necessary but insufficient criteria for a causal relationship (thus the somewhat misleading moniker "Granger causal"). They use the resulting map of relationships among variables to

build diagrams contrasting projects that *graduate* from those that don't.

E.3 Methods

Replications can take on many forms and multiple typologies have been proposed (Juristo et al., 2010). Replication styles differ in whether new data, methods, or settings are used. In this paper, we confirm key findings in L. Yin et al. (2022) using the same methods and data. We also use this analysis as a jumping-off point to analyze their data using more rigorous statistical methods which allow us both to confirm and extend the original findings.

E.3.1 Data

The raw data in L. Yin et al. (2022) is composed of software commit messages, policy documents, email messages among developers, and the outcome of the incubator process. The ASF has a norm about using public and shared electronic discussion tools to make deliberations transparent and as a form of documentation. This means we can have a reasonable level of confidence that this data captures important communication among members. The key outcome in L. Yin et al. (2022) reflects whether the project has achieved milestones and *graduated*. We use only the processed data published by L. Yin et al. (2022). These data were collected longitudinally and analyzed at the project-month level for a period of exactly 24 months. People in the dataset are identified as belonging to one of three types: whether they are *mentors* from the ASF, *committers* registered with the ASF, or *contributors.*

E.3.2 Measures

The analysis of the raw data in L. Yin et al. (2022) includes hand annotating of discussions to indicate when people are discussing governance and rules as part of developing their organization (i.e., making 'institutional statements'), BERT-based detection of these institutional statements trained by the hand-annotated data ($\sim$ 1.2M institutional statements), topic modeling of discussions (12 topics, with data presented at the project-month level from

4,794 project-months and 253 projects), network analysis of who interacts with whom measured at the project-month, and network analysis of who interacts with what files (again as project-month).

The variables of interest in the original analysis in L. Yin et al. (2022)—and in our replication—are:

- *num nodes*: node count in the social network, per project, per month
- *graph density*: social network density, per project, per month
- *weighted mean degree*: social network degree, per project, per month
- *num file node*: node count in the file half of the bipartite technical network, per project, per month
- *num file per dev*: technical network density, per project, per month
- *num dev nodes*: node count in the developer half of the bipartite technical network, per project, per month
- *topic proportions*: proportion of messages identified as part of each topic, per project, per month
- *mentor IS*: # institutional statements by mentors, per project, per month
- *committer IS*: # institutional statements by committers, per project, per month
- *contributor IS*: # institutional statements by contributors, per project, per month
- *graduated*: the outcome of incubation (binary), per project

E.3.3 Analytic Plan

In developing an analytical plan for our replication, we focused on two goals: first, to replicate findings from L. Yin et al. (2022) using rigorous inferential methods, and second, to extend their approach to assess the potential to intervene in the context they describe.

Frequency of Institutional Statements

The first model L. Yin et al. (2022) present estimates the relationship between *institutional statements* made by authors from three different groups (mentor, committer, contributor) and *graduation*. In L. Yin et al. (2022), the *institutional statement* counts are divided by the group membership of the author—mentor (from the ASF), committer (registered with the ASF), or contributor (not registered with the ASF). A statistical test for differences in these counts between projects that *graduated* and projects that did not was conducted using a Mann-Whitney U test. Each of the group memberships is tested in isolation.

Project *graduation* is a one-time observation at the level of the project. *Institutional statement* counts were collected at the month level. This creates a risk of overconfidence due to non-independence of observations. To address this risk, we collapsed the data to a project-level mean value for each predictor of interest. After this procedure, the dataset contains information about 253 projects, of which 204 successfully completed the incubator process and *graduated*.

To replicate this analysis, we used a Mann-Whitney U test. Our results show the same pattern of statistical significance as L. Yin et al. (2022) reports. We then extend their analysis by fitting three Bayesian logistic regression models in Stan using the RStan Library (Team, 2023), with likelihood, prior, prior predictive check, and posterior predictive check as described in the Appendix.

Discussion Topics

L. Yin et al. (2022) evaluated whether projects that go on to *graduate* tend to discuss different topics than those that do not.We replicate this analysis using a Mann-Whitney U test with similar results in terms of statistical significance. Because the data are measured within months, we extend the analysis in L. Yin et al. (2022) by collapsing the dataset to a mean prevalence of each topic per project. Further, because we intend to fit a model with all topics simultaneously, we account for the fact that topic model proportion is 'zero-sum' (i.e., one value cannot rise without re-adjusting the prevalence of the others). Therefore, we use demeaned percentage and no intercept in our model fit—i.e., how high or low a topic's prevalence is, relative to the topic mean.

Causal Effect Estimation

L. Yin et al. (2022) explored the relationship between *institutional statements* and *social and technical network measures*. They evaluated how these relationships differ for *graduated* and *non-graduated* projects using Granger analysis. L. Yin et al. (2022) offered a diagram of the pairwise relationships among these variables as an outcome of this analysis. The original paper does not include *graduation* itself in its Granger analysis. Instead, the Granger diagram illustrates pairwise connections in terms of how they appear for *graduated* projects and how they appear for *non-graduated* projects.

We now go beyond the findings in L. Yin et al. (2022) and embrace an extension with an analysis built on a conjecture as follows. Imagine that we want to intervene in the system depicted in Figure E.1 to improve a project's *graduation* chance. Further, imagine that we hope to use data to identify the interventions that might more generally cause projects to evolve into successful, healthy, sustainable, well-governed communities. What kind of intervention might we imagine for these projects and what might the effect of such a proposed intervention be? The challenge we face is that the data as presented are observational: there is no random assignment, and no intentional treatment *per se*.

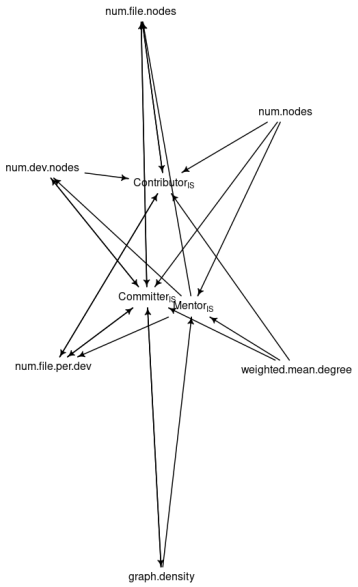


Figure E.1: The union of all Granger relationships as described in Yin et al. (2022).

Projects are similar in their desire to become successful members of the ASF. They were also all evaluated and accepted by the ASF to undergo incubation. Perhaps, as a result, they are comparable in all the dimensions that matter. Additionally, given norms around using mailing lists and code commits as the core components of building software and coordinating work, perhaps it is reasonable to assume that we have captured all relevant components and controls. This opens up the possibility of using causal estimation techniques under the *conditional ignorability* assumption, following the process and line of argument detailed in Oganisian and Roy (2021). Conditional ignorability is a strong assumption, but it opens the possibility of using these data to make causal estimations. The value of such an analysis is that it allows us to evaluate in advance what the impact of a proposed treatment might be and assess whether the treatment is worth trying at all.

Developing and evaluating an intervention conjecture requires several components: an evaluation of causal pathways between our measures and the outcome, a proposed intervention that is operable within those pathways, and an estimation of effect (Oganisian & Roy,

2021). We make several assumptions to transform the Granger-derived diagram from L. Yin et al. (2022) into a DAG that can be used for Bayesian estimation of causal effects. For example, we must place our outcome variable on the diagram and generate directed edges from what we consider to be factors that have a direct impact and that outcome. We generate directed edges from each of the six sociotechnical factors considered in L. Yin et al. (2022), supported by the following argument. Consider that the ASF's graduation criteria includes evaluating whether the project is sustainable and being governed appropriately. Although the criteria are holistic, one might expect that material production activity are the most direct and proximate causes of voting for graduation because they are visible and countable. These might be reflected in measures of generating code, fixing bugs, and so on, as well as the number of people involved. It seems most likely that a project that is actively engaging participants in production is likely to be accepted. A project that is not developing or improving its code seems unlikely to graduate even if it seems to make decisions reasonably. Projects violating governance standards would probably have received negative feedback along the way that, if not resolved, would have diminished participation, perhaps generating a negative feedback loop. Likewise, those where no code was being written also seem likely to enter a negative loop in which lack of progress on the work led to dissatisfaction and dissent around governance.

Therefore, although governance is part of the voting standard, we might assume that participants base their votes on their understanding of how successfully the organization is accomplishing work, including making decisions in ways that lead to more work getting done and participants persisting in their involvement. We therefore place directed edges between each of the six social and technical measures. However, we do not place directed edges from *institutional statements* and *graduation*, on the assumption that *institutional statement* counts, although identified in the L. Yin et al. (2022) inspection of commit and email messages, are not displayed directly to voters.

This suggests the relationships depicted in Figure E.2. This proposed connection between the work done and the graduation votes accomplishes our goal of placing the outcome variable

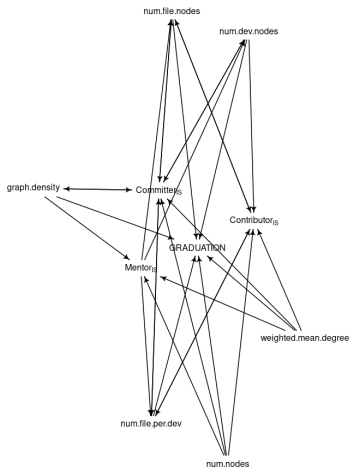


Figure E.2: Speculatively adding the outcome variable *graduation* as connected to each technical measure.

graduation on the graph.

L. Yin et al. (2022) make the case that *institutional statements* are associated with *graduation*. However, for our causal analysis, we need to evaluate the available space for interventions. We know we want to cause projects to develop healthy community governance and a critical mass of activity and participants. FLOSS projects are self-organized and volunteer-driven. Indeed, a key feature of open source is participants' abilities to choose tasks in their range of skills and oriented to their interest (Benkler, 2002). Some interventions might disrupt the factors that make open source work. For example, direct interventions on people to cause them to join and contribute (e.g., by paying them), might lead to motivational crowding or reduce efficiency. We need an intervention that does not involve directly perturbing the motivations of participants.

One might imagine such an intervention in the form of mentorship training. For example, mentors might be unaware of the value of making *institutional statements* in response to some forms of change. If we were to train mentors on the significance of making *institutional statements*—natural, well-reasoned, and sincere statements like the ones they already make—

we might persuade them to make such statements when the most valuable opportunities arise or make it more likely that they notice those opportunities.

The Granger analysis offers hints as to the kind of intervention we might need to make. We also observe that in projects that *graduated*, a Granger relationship exists between *weighted mean degree* and *mentor institutional statements*—but not in those that did not *graduate*. Further, in projects that did not *graduate*, an increase in *num nodes* and *graph density* drove a change in *mentor institutional statements*. Degree and density both measure the connectedness of a network but it is possible for these measures to diverge. For example, high-degree, low-density networks might have a strong core-periphery structure, while low-degree, high-density networks might be more hub and spoke or star structure. Given that mentors were seemingly responding to changes in the connectivity strength in the network, they might benefit from information about whether the structure is headed in a direction associated with *graduation* and future sustainability. Encouraging and supporting mentors in identifying network structure might support them in more effective responses.

Further, in *graduated* projects, increased *num nodes* (which we might think of as newcomer influx) drove *committer institutional statements*—perhaps handling newcomers is a task more suited to project leadership—and did not drive *mentor institutional statements*. In projects that did not *graduate*, increased *num nodes* drove *contributor institutional statements*. Perhaps dealing with newcomers tended to fall to community members instead of leaders. Or perhaps some unobserved variable of the state of the project drove rank-and-file members to feel the need to make more *institutional statements*? At any rate, these results suggest that an intervention might be crafted that (a) increased the number of *institutional statements* and (b) included encouragement to attend the measures L. Yin et al. (2022) identify and we highlight. A deeper analysis of the events associated with these measures might allow the "just so" story we are weaving to be more persuasive.

At any rate, now we have an argument that motivates a measurement of a potential causal effect. If we could increase *mentor institutional statements*, what is the potential range of results we might expect?

Figure E.3: Dropping variables based on guidance from the Dagitty library to identify bad controls versus good controls, and focusing on mentor institutional statements simplifies our analysis.

To build a causal analysis of the system we have outlined using a DAG approach, we need to engage in one final step: trimming out any variables that are bad controls and maintaining those that are good ones. We can draw from the approach described in Cinelli et al. (2022) and examine the DAG resulting from our conjecture. The Dagitty R library (Textor et al., 2017) allows us to evaluate what controls are necessary to evaluate the direct effect of *mentor institutional statements* on *graduation*.

The graph analysis suggests that we can drop *committer* and *contributor institutional statements* from the final model as shown in Figure E.3. This final model suggests that the pathway by which *mentor institutional statements* can cause *graduation* is by influencing the *num file nodes*, the *num files per dev*, and the *num dev nodes* (the outbound arrows from mentor IS that are then inbound to graduation), and confounders to consider are *weighted mean degree*, *num nodes*, and *graph density* (measures that are inbound to both textitmentor IS and *graduation*). We can therefore model the total effect of *mentor institutional statements* on *graduation* using a far simpler model than the one we began with, using only our treatment

mentor IS, outcome *graduation* and three controls: *weighted mean degree*, *num nodes*, and *graph density*. We fit this model using Bayesian logistic regression, with a likelihood, a prior predictive check, a posterior predictive check, and model diagnostics as presented in the Appendix.

E.3.4 Ethics

This study was conducted entirely using publicly available data published by L. Yin et al. (2022). Our analysis only uses their final dataset, which does not contain identifiable data and does not involve interaction or intervention with human subjects. We used our institution's self-determination worksheet and determined that this research using these data is not considered human subjects research per institutional guidelines.

E.4 Results

E.4.1 Institutional Statements

	mean	sd	2.5%	50%	97.5%	n_eff	Rhat
sigma	0.99	0.99	0.03	0.69	3.68	13603	1.00
Intercept	0.52	0.22	0.09	0.52	0.94	7960	1.00
mentor IS	0.06	0.01	0.03	0.06	0.09	8081	1.00

Table E.1: Coefficients from fitting a Bayesian logistic regression where the predictor is increased mean monthly institutional statements from mentors.

We successfully replicated the L. Yin et al. (2022) analysis of the association between *institutional statements* and *graduation* for each of the three groups (mentor, committer, contributor), evaluating each with a Mann-Whitney U-test, $p < .001$. We then evaluated the association between the mean monthly count of *institutional statements* between each group and project *graduation* with model results as shown in Table E.1.

Since we estimated this association using a logistic regression, the direct interpretation of the parameters is differences in log-odds. One common way to discuss log-odds parameters is to convert them to differences in model-predicted probabilities. The estimated probability of graduating without mentors having made *institutional statements* is 0.626. We might interpret this finding using values inspired by the median of the mean monthly *institutional statements* (15) and the grand mean or mean of the mean monthly statements (23.3). For projects where mentors make on average 10 *institutional statements* per month (i.e., below the median), the probability of graduating overall is increased to 0.747. For projects where mentors make on average 30 *institutional statements* per month (i.e., above the grand mean), the probability of graduating overall is increased to 0.902.

	mean	sd	2.5%	50%	97.5%	n_eff	Rhat
sigma	0.99	0.99	0.02	0.69	3.65	15679	1.00
Intercept	0.71	0.19	0.34	0.71	1.09	9985	1.00
committer IS	0.07	0.02	0.04	0.07	0.11	9948	1.00

Table E.2: Coefficients from fitting a Bayesian logistic regression where the predictor is increased mean monthly institutional statements from committers (bg).

We fit similar models for the other two sources of *institutional statements*: contributors and committers. We again interpret the model by using sample values and converting to probabilities. For the committer model, the model-predicted probability of graduating with committers not having made *institutional statements* is 0.671. For projects where committers make on average 10 *institutional statements* per month, the probability of graduating overall is increased to 0.807. For projects where committers make on average 30 *institutional statements* per month, the probability of graduating overall is increased to 0.946.

For the contributor model as we did for the committer and mentor model, the model-predicted probability of graduating with contributors not having made *institutional statements* is 0.6. For projects in which contributors make on average 10 *institutional statements*

	mean	sd	2.5%	50%	97.5%	n_eff	Rhat
sigma	1.00	1.02	0.02	0.68	3.77	13733	1.00
Intercept	0.41	0.22	-0.03	0.40	0.84	8232	1.00
contributor IS	0.07	0.02	0.04	0.07	0.10	8163	1.00

Table E.3: Coefficients from fitting a Bayesian logistic regression where the predictor is increased mean monthly institutional statements from contributors.

per month, the probability of graduating overall is increased to 0.751. For projects in which contributors make on average 30 *institutional statements* per month, the probability of graduating overall is increased to 0.924.

E.4.2 Discussion Topic Prevalence

We were successful in replicating the findings in L. Yin et al. (2022) with respect to topic prevalence using their approach: a Mann-Whitney U-test using a month-level dataset with each topic and group (graduated versus not) de-meaned separately. We extend this perspective with our model—fit with all topics and using the centered topic-wise mean from all projects and no intercept—to allow evaluation of whether projects that graduate have an overall higher or lower prevalence of each topic overall in opposition to the others. The results of our more conservative modeling approach extends the findings in L. Yin et al. (2022) about temporal trends. The largest differences in our analysis (based on coefficient magnitude) are in projects that discuss their Progress Report and Collective Decisions more often than the average project. There are modest differences in the rate of discussion of Project Release, Report Review, Mailing List Issues, and Software Testing.

E.4.3 Causal Effect of Mentors' Institutional Statements

We assessed the potential causal effect of *mentor institutional statements* as described in detail in §E.3.3.

	mean	sd	2.5%	50%	97.5%	n_eff
sigma	0.99	0.98	0.02	0.69	3.65	6684
1: Progress Report	0.89	0.37	0.16	0.89	1.64	1701
2: Collective Decision	1.01	0.37	0.29	1.01	1.76	1662
3: Project Release	0.90	0.37	0.19	0.90	1.64	1636
4: Community	0.79	0.38	0.06	0.78	1.55	1656
5: Report Review	0.92	0.38	0.18	0.91	1.68	1636
6: Mailing List Issues	0.91	0.38	0.18	0.91	1.66	1627
7: Documentation	0.83	0.37	0.11	0.83	1.59	1625
8: Software Testing	0.88	0.37	0.16	0.87	1.62	1637
9: Licensing Policy	0.88	0.38	0.15	0.88	1.63	1616
10: Routine Work	0.86	0.37	0.16	0.86	1.60	1630
11: Mentorship	0.47	0.38	-0.27	0.47	1.23	1752
12: Software Distrib.	0.81	0.38	0.09	0.81	1.56	1674

Table E.4: Coefficients from fitting a Bayesian logistic regression where the predictor is the project's centered mean percent of monthly discourse within each of 12 topics. All Rhat values are 1 (omitted for space concerns).

The output of our Bayesian logistic regression model suggests that the causal effect of *mentor institutional statements* from mentors on graduation is on the order of a 0.012 increase in log-odds with each additional statement, all other measures being equal. As in the previous estimate, we can use prototypical values to interpret this coefficient.

In a model with all controls (*weighted mean degree*, *num nodes*, and *graph density*) held at median values, the model-predicted probability of graduating without mentors having made *institutional statements* is 0.816. For projects in which mentors make on average 10 *institutional statements* per month, the probability of graduating overall is increased to 0.833. If these project mentors were to increase their *institutional statement* frequency such

	mean	sd	2.5%	50%	97.5%	n_eff
sigma	0.99	0.99	0.03	0.69	3.64	20425
Intercept	-1.04	0.36	-1.77	-1.04	-0.36	11486
mentor IS	0.01	0.01	0.00	0.01	0.04	18157
Graph Density	0.41	0.27	0.02	0.37	0.95	16255
Num Nodes	0.17	0.04	0.10	0.17	0.25	11886
Wt. Mean Deg.	0.02	0.02	0.00	0.02	0.06	17565

Table E.5: Coefficients from fitting a Bayesian logistic regression to assess the causal effect of increased institutional statements by mentors. Controls are graph density, num nodes, and weighted mean degree. All Rhat values are 1 (omitted for space concerns).

that they make on average 30 *institutional statements* per month, they would cause the probability of graduating overall to increase to 0.863, all other factors being equal. This improvement is 4.7%.

However, in a model with controls held at observed minimum values, the model-predicted probability of graduating without mentors having made *institutional statements* is 0.261. For projects in which mentors make on average 10 *institutional statements* per month, the probability of graduating overall is increased to 0.284. If those project mentors were to increase their *institutional statement* frequency such that they make on average 30 *institutional statements* per month, they cause the probability of graduating overall to increase to 0.334, all other factors being equal. This improvement is 7.3%.

E.5 Discussion

E.5.1 Replication Observations

We find that the findings in L. Yin et al. (2022) replicate under strict alternate inferential modeling methods. The significance of the *institutional statements* described in L. Yin et al. (2022) appears prominently in our results. Our re-analysis of the topic model findings in L.

Yin et al. (2022) allows for an overarching view of project discourse that is complementary to their descriptive temporal approach, particularly in the emphasis of the importance of discussing the topic L. Yin et al. (2022) call Progress Report and the topic they call Collective Decisions as part of making *institutional statements*.

In all, these results suggest that the *institutional statements* framework presented in L. Yin et al. (2022) provides particularly valuable insight into project graduation and adds evidence to the perspective that Ostrom's Institutional Analysis framework is useful for studying novel organizational forms such as open source software.

E.5.2 The Role of Mentors' Institutional Statements

We used Bayesian estimation of causal effects and directed graphs that we derive from the work in L. Yin et al. (2022) to identify a potential causal pathway for intervention. This analysis allows us to develop a proposed data-driven strategy to increase the probability that a project will graduate from an incubation program: by guiding mentors to make more *institutional statements*. Our work suggests that *mentors' institutional statements* are valuable elements in developing successful open source projects.

E.5.3 Implications for Software Development Organizations

Our results suggest that software incubation programs could benefit from interventions that cause mentors to make more *institutional statements*. The causal pathway for these statements may be that they tend to lead to changes in the number of developers in the project, the number of files that developers work on, and the number of overall files in the project. L. Yin et al. (2022) suggest that in projects that graduated, mentors responded to the *weighted mean degree* of the project, while in retired projects, mentors tended to respond to changes in *num nodes* (i.e., the overall number of participants) and the *graph density*. We observe that metrics like *weighted mean degree* are easily calculated by tools and might support mentors in noticing patterns in the community that might otherwise be difficult to assess without computational support. However, as with any metric, this observation should be used with

caution in the absence of more evidence on both the positive impact and the costs that tracking these metrics might have on organizational performance.

E.6 Limitations

Our replication is limited in that we used the published data from L. Yin et al. (2022) and worked only to replicate their results using alternate statistical methods. Translation of Granger results into a directed graph for Bayesian causal estimation required us to assume conditional ignorability—i.e., that we had so thoroughly captured the relevant factors that we could treat the data as resulting from a causal design. Although these assumptions were necessary to elaborate a causal mechanism and assess a potential treatment effect, the ultimate result should be considered speculative.

L. Yin et al. (2022) characterized the top 2% of each measure as outliers and omitted these observations, while we included them. However, we agree that the data does contain extreme values. One project (Open Office) has a mean of over 800 for *mentor institutional statements*, which is almost 3 times the next highest value (Cloudstack). Open Office also has the second-highest number of *committer institutional statements* (288, with Flex having the most at 320). Open Office also has the highest number of *contributor institutional statements* (375), with Cloudstack in second place (280) and Flex in third (164). Given the relatively small sample size (n=253), these extreme values might pose a risk to the analysis. To address this threat, we ran the same analysis using median monthly statements to measure central tendency. We gained confidence from the fact that our results were similar.

Additionally, we observe that Open Office—like CloudStack and Flex—was developed under a commercial development model and then donated to the ASF. We might suspect that the transformation of a project developed under a corporate umbrella might present different challenges than transforming an independent community project. Although we believe this is an important topic for future research, evaluating the role of *institutional statements* based on the diverse development models that predate incubation is beyond the scope of this analysis.

E.7 Conclusion

Overall, we find that the conclusions drawn in L. Yin et al. (2022) replicate using robust statistical methods. In our extension of their work, we take an inferential and Bayesian approach that allows us to compare the relative importance of the factors L. Yin et al. (2022) analyzed. Our results suggest that mentors have a powerful role to play in the success of projects in an incubation program like the ASF's and that discourse about governance originating from someone in a mentorship role may cause the project to be more likely to achieve the success that participants seek.

Acknowledgements for Appendix E

This replication was originally developed as a course project conducted by the first author under the supervision of Carlos Cinelli at the University of Washington. The paper would not have been possible without the original publication from L. Yin et al. (2022) and its authors' commitment to open science principles.

We are also indebted to the generous volunteers of the Apache Software Foundation who, in addition to producing FLOSS, have made their records available to the public. We also gratefully acknowledge support from the Sloan Foundation through the Ford/Sloan Digital Infrastructure Initiative (Sloan Award 2018-113560 and the National Science Foundation (Grant IIS-2045055). This work was conducted using the Hyak supercomputer at the University of Washington and research computing resources at Northwestern University.

E.8 Bayesian Methods Supplement for "Institutional Statements as a Driver for OSS Project Success: A Bayesian Replication and Extension of Yin et al.'s 'Open Source Software Sustainability"'

This methods supplement describes the likelihood, prior, prior predictive check, and posterior predictive check (including MCMC diagnostics) for the Institutional Statements and Bayesian causal estimation models.

E.8.1 Institutional Statements

The coefficients of the mentor institutional statements model appear in Figure E.4.

Institutional Statements: Likelihood

For the *likelihood* for this model, we consider what we know about the data-generating process: the treatment variable is the mean number of institutional statements made by individuals in one of three groups and varies from 0 to 825 (median 12, mean 20, sd 42). With a standard deviation more than twice its mean, we know that this count is overdispersed. The outcome, graduation, is a logical true or false. These two factors suggest using a logistic regression model for the likelihood, $logit(\theta_i) = \alpha + \beta_g x_i$, where the expected outcome is $\theta_i \in [0, 1]$, α is our intercept or the predicted log-odds of a successful intervention absent any treatment and β_g is the incremental increase in log-odds associated with a one-unit increase in the treatment (mean count of institutional statements per month). Following the lead of the previous work in L. Yin et al. (2022), we fit these three models independently.

Institutional Statements Prior

For our *prior*, we expect that the impact of each additional statement might be relatively small, but positive: we also know from prior work that graduated projects are fairly common (in this data, 204 of the 253 projects graduated), and at least some number of institutional statements may be relatively common whether a project is dysfunctional or succeeding. Therefore, we would expect an intercept to be normally distributed around 0 (corresponding to a probability of 0.5—maybe it is just random chance that leads a project to graduate if there are no institutional statements made) with a standard deviation of 0.5 ($\alpha \sim \mathcal{N}(0, 0.5)$). This is a pretty weak prior. The impact of making institutional statements seems like it should be positive and small, but not so small that we give ourselves numerical problems, so we adopt a normal distribution with a mean of 0.05 and a standard deviation of 0.05—since we already know that we have some overdispersion, this would allow for substantial variation

Figure E.4: Predicted log-odds of graduation associated with increases in average monthly institutional statements from mentors, committers, and contributors. 'a' is the model intercept (the log-odds of graduating with members of the group having made 0 institutional statements) and 'bg' is the log-odds associated with incremental change in the number of institutional statements. Red lines indicate an 80% confidence interval and black lines indicate a 95% confidence interval.

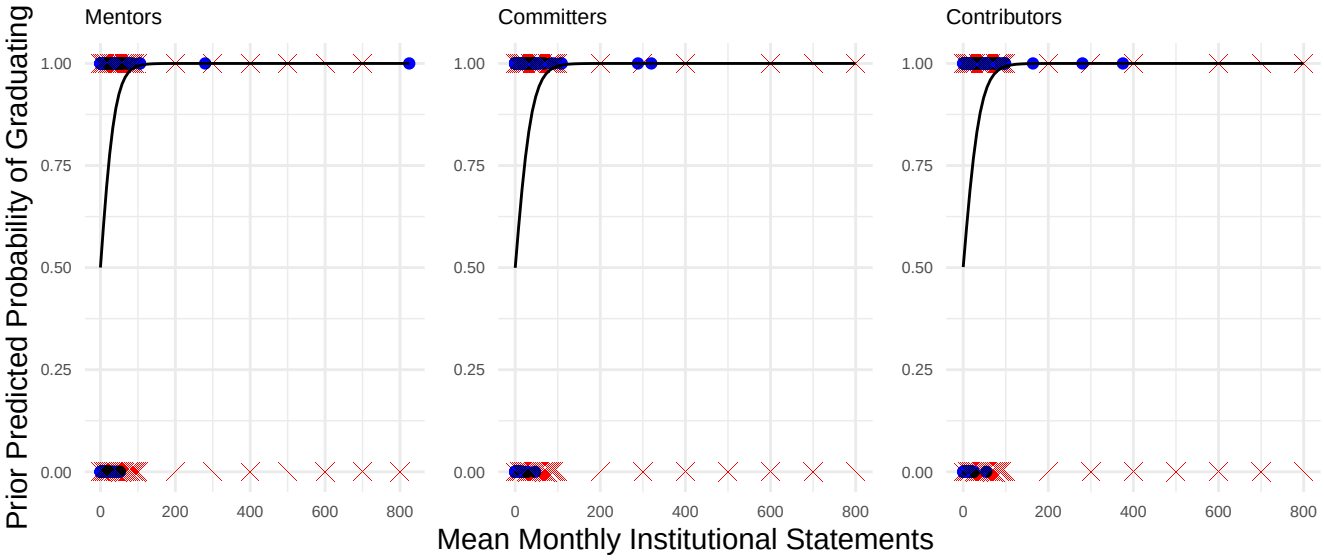


Figure E.5: The prior we set on our model generates the logistic regression curve seen in black. Red indicates data simulated from the provided prior. Actual data in blue.

$(\beta_g \sim \mathcal{N}(0.05, 0.05))$.

Institutional Statements Prior Predictive Check

To evaluate the reasonableness of these priors relative to the data, we performed a *prior predictive check*. The results of these checks can be seen in Figure E.5. These plots suggest that we selected reasonable priors.

Institutional Statements Posterior Predictive Check

To validate these three models, we performed a *posterior predictive check*. The results of these checks can be seen in Figure E.6. These three plots all show values in reasonable locations and that my posterior seems well suited to the data; this suggests that the model fit was successful.

In general, comparing the prior to the posterior, we can see that the posterior model is closer to the y-axis sooner.

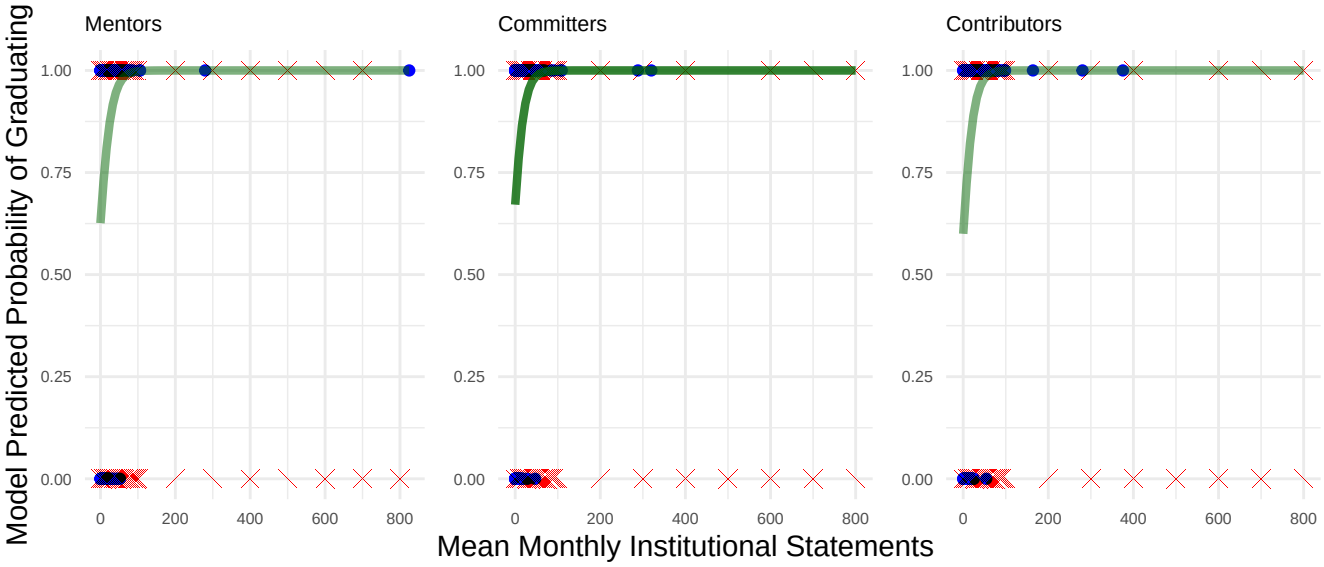


Figure E.6: Our posterior model generates the logistic regression curve seen in green. Red indicates data simulated from the provided prior. Actual data in blue.

Finally, I offer some basic diagnostics of the MCMC chain, in two forms: traceplots and the $\hat{R}$ diagnostic. Evaluating traceplots means looking for a chain that explores the space and does not get stuck, and for $\hat{R}$ (the Gelman-Rubin diagnostic), values of 1 or less indicate that the chain has converged.

In Figure E.7, we see healthy traceplots in which the chains explore the space, do not seem to get stuck, and each of the chains overlaps the others. $\hat{R}$ values for the mentorship model are: σ : 1, α : 1.0007, β : 1.0008. $\hat{R}$ values for the contributor model are: σ : 1.0002, α : 1.0001, β : 1.0002. $\hat{R}$ values for the committer model are: σ : 1.0002, α : 1.0003, β : 1.0003. In each case, we observe that the values of $\hat{R}$ are quite close to 1, indicating that chains converged.

E.8.2 Speculative Causal Analysis

The causal effect of mentor institution statements is visualized in E.8.

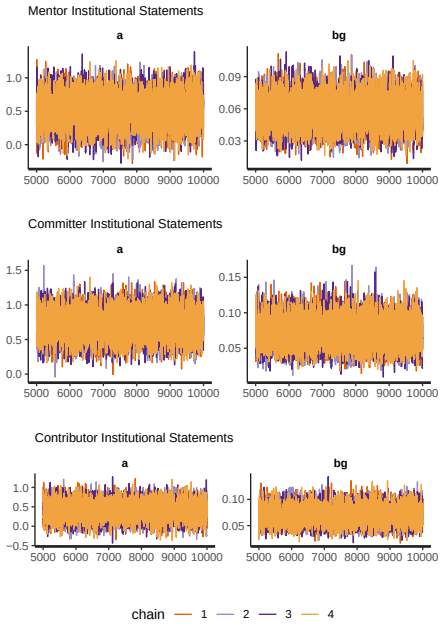


Figure E.7: These traceplots display the behavior of the MCMC chain when fitting the institutional statements model for mentors, committers, and contrbutors.

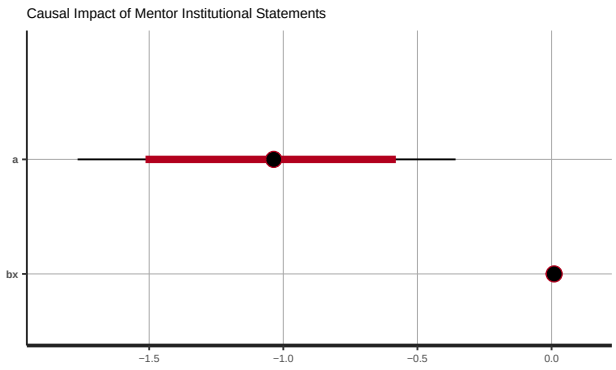


Figure E.8: Predicted log-odds causal effect on graduation of mentor institutional statements. Red lines indicate an 80% confidence interval and black lines indicate a 95% confidence interval.

Causal Analysis Likelihood

The likelihood for this model is a logistic regression with a linear combination of mean institutional statements as a predictor, with graph density, number of nodes, and weighted mean degree as controls held at the median level.

Causal Analysis Prior

Previous work evaluating the factors in the model found that each is associated with better graduation rates. We propose a relatively uninformative prior with all values drawn from a normal distribution with $\mu = 0$ and $\sigma = 1$.

Causal Analysis Prior Predictive Check

For my prior predictive check, I created a simulated dataset holding my controls at low values—the .05 quantile of number of nodes, graph density, and weighted mean degree. I examined what the priors I specified would look like using these simulated data versus the observed data. The idea here is to simulate how variations in the level of institutional statements from mentors might affect organizations which are otherwise not evincing strong performance (and hence whose success is in no way guaranteed). The prior predictive check for this model appears in Figure E.9.

Causal Analysis Posterior Predictive Check

The posterior predictive check for this model appears in Figure E.10.

Causal Analysis Diagnostics

To assess performance of this causal estimation model, we examine the effective sample size, traceplots of the chains, and the value of the $\hat{R}$ diagnostic.

The traceplots look healthy, exploring the space, and not getting stuck. The effective sample size for alpha is 1.1486×10^4 and the effective sample size for beta is 1.8157×10^4.

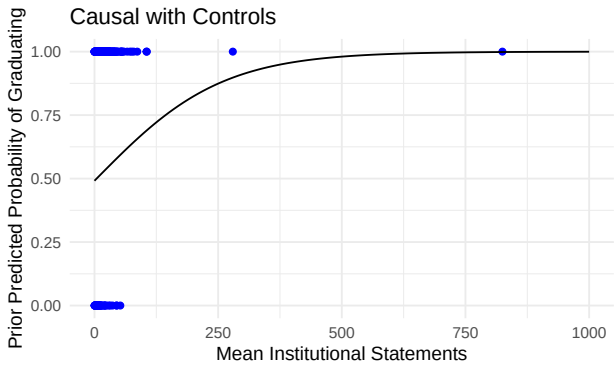


Figure E.9: The prior we set on our model for the causal effect of institutional statements generates the logistic regression curve seen in black, all controls set to the median of observed value. Red indicates data simulated from the provided prior. Actual data in blue.

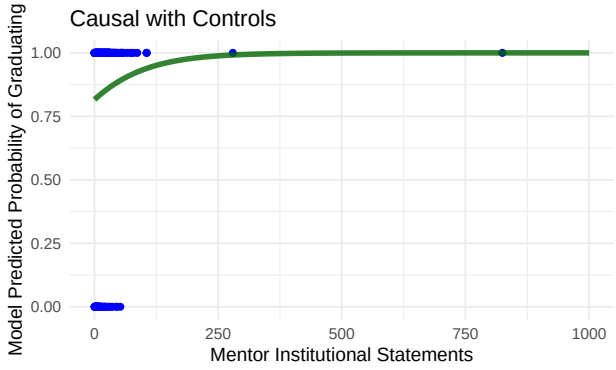


Figure E.10: Our posterior model for the causal effect of mentor institutional statements generates the logistic regression curve seen in green, all controls set to the median observed value. Red indicates data simulated from the provided prior. Actual data in blue.

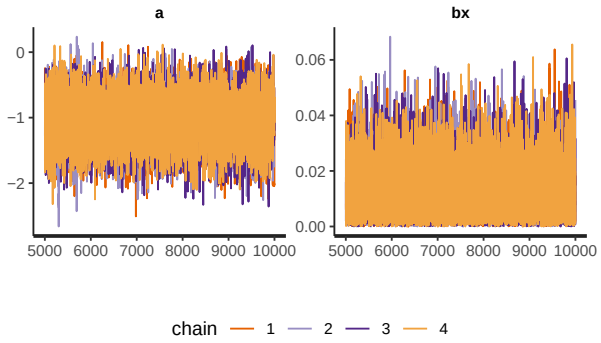


Figure E.11: Traceplot of the MCMC chains associated with fitting Causal estimation of Mentor Institutional Statements.

The values of $\hat{R}$ (alpha: 1.0001; beta: 1; σ 0.9999) are consistent with a converged chain.

Appendix F

ENGINEERING FORMALITY AND SOFTWARE RISK IN DEBIAN PYTHON PACKAGES

This appendix illustrates the use of underproduction as a dependent variable in understanding the implications of organizational structure and behavior. In it, we explore the risks that software communities face when they take on more formality. In particular, we find that underproduction risk is *higher* as the organization takes on more formal structure, and that power-sharing is associated with *lower* underproduction. We found no statistically significant relationship between formality of work process and underproduction.

These findings are important for my broader theory of social production failure because they illustrate the challenges of organizing and structuring a community in ways that can prevent or remediate these failures, and the necessity of thinking carefully about how these communities are to be organized and governed. This work suggests that although substantial literature on maturity and sustainability for organizations in general may suggest adoption of formal processes and governance, formality may hold substantial risks for these communities and should proceed with caution. To the extent that evolving governance models or attempts to avoid or abate underproduction may squeeze out some of the traits of commons-based peer production (e.g. by diminishing self-organizing and self-selection of tasks), we may find these interventions have negative unintended consequences.

This chapter was previously published as: I contributed substantially to the article, including in its conceptualization, framing, dataset development, and analysis.

Gaughan, Matt; Champion, Kaylea; Hwang, Sohyeon. (2024). Engineering formality and software risk in Debian Python packages. *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*

F.1 Introduction

Across global computing infrastructures, free/libre open source software (FLOSS) packages underpin the successful operation of widely used technical systems (Crowston et al., 2012). Yet this crucial software is often "underproduced" — that is, not as well-maintained as we might expect, given its importance (Champion & Hill, 2021). Underproduction can be consequential, leading to disruption across global networks. Two notorious such failures are the FLOSS security defects known as Heartbleed and Log4Shell. The Heartbleed vulnerability impacted the widely-used OpenSSL library. At the time the vulnerability was announced, OpenSSL was decaying from community abandonment, with no full time developers and only $2000 in annual donations. Heartbleed was eventually remediated by large-scale, formally organized developer engagement — the very kind which OpenSSL had previously lacked (Walden, 2020). The Log4j library is maintained by The Apache Software Foundation and a dedicated development team. In 2021, researchers identified the zero-day Log4Shell vulnerability, which enables attackers to inject malicious code into already-running Java programs ("Log4Shell 10 days later," 2021). While the Log4j team was able to respond quickly with a series of patches, the long-undetected nature of this vulnerability suggests that like OpenSSL, Log4j was underproduced.

With 96% of software in the 'critical infrastructure sector' containing FLOSS code, the United States Cybersecurity and Infrastructure Security Agency (CISA) established an "Open Source Software Security Roadmap" in September 2023 ("CISA Open Source Software Security Roadmap," 2023) highlighting the need to study the social and technical antecedents of FLOSS software failure. This national attention is timely: security research firm Sonatype found in their 2023 *State of the Software Supply Chain Report* that across four major engineering ecosystems (NPM, Maven, PyPi, nugent) only 11% of projects were actively

maintained (Krill, 2023). Heightened scrutiny of FLOSS engineering practices as well as escalating warning signs of underproduction make examining the causes of underproduction all the more urgent.

A promising direction to diagnose both causes of and remediation strategies for underproduction is examining a structure commonly employed in FLOSS projects: commons based peer production (CBPP) (Benkler, 2006). Benkler defines CBPP as a production model reliant on decentralized individuals who self-select tasks, motivated by a range of factors (Benkler, 2002). Yet Benkler also writes that in the face of "knowledge intensive, creative, and complex" problems, projects must organize to "[elicit] diverse pro-social motivations" from contributors (Benkler, 2017). Said simply, although CBPP is an extremely valuable way of collaborating, FLOSS organizations may find it difficult to ameliorate underproduction because requisite yet undesirable tasks may lay neglected when contributors work according to their own interests.

In this paper, we make three contributions: we offer empirical assessments of the relationship between (1) underproduction and overall governance formality, (2) underproduction and the concentration of developer responsibility, and (3) underproduction and the management of work products. We place our work in the context of previous work in §F.2 and describe our analysis in §F.3 before presenting results of our analysis in §F.4. We discuss the implications of this work in §F.5 with limitations noted in §F.6 before concluding in §F.7.

F.2 Background

F.2.1 Governance in CBPP

Governance refers to the totality of the systems in an organization which incorporate decision making, operational control, and incentives (X. Yin & Zajac, 2004). Thus, as an organization matures, institutionalization in governance can be understood as "the processes by which the social processes obligations, or actualities come to take a rulelike status in social thought or action" across subunit actors (in the case of FLOSS project engineering, software developers)

(Meyer & Rowan, 1977). In CBPP governance, project institutionalization entrenches either formality or informality in decision-making processes (Healy & Schussman, 2003).

The influential work of Ostrom on governing commons-based resources notes that communities prefer internally developed governance practices, which naturally leads to variations in the resulting governance arrangements (Ostrom, 2015). For example, an analysis by Hwang and Shaw of CBPP governance on Wikipedia found that communities with shared goals, technical infrastructures, organizational structures, and institutional trajectories end up producing diverging rule sets and deliberating at length over shared rules (Hwang & Shaw, 2022). At times, how CBPP communities self-govern can lead to counter-intuitive patterns of institutionalization that introduce bureaucratization and hierarchies and create later barriers to participation, following Robert Michel's "iron law of oligarchy." (Shaw & Hill, 2014). The full extent of the productive repercussions of the variety of approaches to CBPP remains unclear.

F.2.2 Governance of FLOSS Engineering

Similar to other CBPP communities, FLOSS projects adhere to a wide range of organizational forms; the management of functional processes (leadership elections, funding allocation, onboarding, conflict resolution) is often internally defined with governance documents such as project constitutions and charters (de Laat, 2007; Tourani et al., 2017). Core to FLOSS governance is the allocation of developer labor and permissions (i.e., code integration, and how work products are released). For example, in centralized projects, formal project leadership may retain engineering management power, exemplified by Linus Torvald's sole control of code merging into the Linux kernel (Jiang et al., 2013).

Code merge patterns—who gets their work merged, and who decides—are a key indicators of how a project is being governed. Even in technically flat communities, Onoue et al. frame participant engagement in FLOSS projects as a hierarchy, with differences in who can enact development actions such as commits, pull requests, comments, and issue events (Onoue et al., 2014). Thus, users who have the requisite permissions to merge code from peripheral

forks into the primary development branch are empowered in project governance over those who simply commit to derivative branches. Tamburri et al. and van Meijel use the hierarchy of merge permissions as evidence of project hierarchy (D. A. Tamburri et al., 2013; D. A. A. Tamburri et al., 2019; van Meijel, 2021). De Stefano et al. adopt this approach to conclude that formality in internal engineering governance structures is associated with diminished community contribution to project code bases (De Stefano et al., 2022).

Tools like project progress trackers also give insight into FLOSS development. The management of such work products are often disputed within project communities; Crowston et al. (2012) observes that that there is no common patterns to how FLOSS projects employ public releases. GitHub "milestones" are indicators to mark the current state of a project, and are used by communities to track progress on collections of tasks such as issues or pull requests[1]. While the milestone feature is platform-specific, GitHub remains the preeminent platform for FLOSS project hosting (Finley, 2019). Zhang et al. found a strong positive correlation between milestone usage and the count of other engineering measures, like commits and releases, in part due to the fact that older projects are more likely to use milestones than younger projects (Zhang et al., 2020).

F.2.3 Governance and Underproduction

Yin et al. observe that in canonical software engineering literature, the success of FLOSS software is primarily defined in two ways: functional development processes or community cohesion (L. Yin et al., 2021). Functional development processes include the management of project risk. In these terms, underproduction is a metric which considers potential for risk (developer neglect), its impact (user adoption), and the likelihood of risk resolution (code quality, bug resolution speed.) Champion and Hill identify underproduction by evaluating whether project quality is aligned with its importance when compared to a theoretical baseline of alignment, where importance and quality are aligned if their non-parametric rankings

[1]https://docs.github.com/en/issues/using-labels-and-milestones-to-track-work/about-milestones

are the same (Champion & Hill, 2021). For a set of packages from the Debian GNU/Linux distribution, Champion and Hill represent project importance through installation count and project quality as the average time to bug resolution (controlling for bug severity).

Prior work examining FLOSS governance formalization vary in their measures of success and subsequent conclusions; Crowston and Howison (2006a) even suggest that informality itself may be a metric of project success. For critical FLOSS systems, D. A. Tamburri et al. (2013) defines success as "24/7 availability in (a) fault-tolerant system". L. Yin et al. (2022) uses the advancement metrics of the Apache Software Foundation's incubator program to represent project success. De Stefano et al. (2022) focus on the frequency of product commits as a metric of community engagement and project success. As such, D. A. Tamburri et al. (2013) observes that as formality within software projects increases, projects may experience friction in the processes of developer engagement; L. Yin et al. (2022) conclude that isomorphic reproduction of formal governance structure may lead to project success; lastly, De Stefano et al. (2022) conclude that the more constrained a project's processes around merging are, the less engineering engagement it might receive.

Observations that formalizing FLOSS governance may lead to both project abandonment and project success are not contradictory but warrant further empirical study. Overall, these findings suggest that projects with higher levels of formality are more likely to be underproduced. Due to the observations of De Stefano et al. (2022) and D. A. A. Tamburri et al. (2019), we propose: **H_A: projects with higher formality are more likely to be underproduced**. Moreover, considering the findings from prior work on merge processes, we propose that **H_B: projects with less concentrated developer responsibility are more likely to be underproduced**. Given the argument in Yin et al. that the governance of project work assists the development process (L. Yin et al., 2022), we propose **H_C: projects with formal work process management are less likely to be underproduced.**

F.3 Methods

F.3.1 Empirical Setting

GNU/Linux distributions, includingDebian in particular, have a substantial history as settings for understanding software engineering communities (Spaeth et al., 2007), including their effective governance (O'Neil, 2009; Sadowski et al., 2008) and lifecycles (Nguyen & Holt, 2012). We examine a collection of projects packaged via the Debian GNU/Linux distribution, which is also a packaging source for Ubuntu and several other distributions. All projects were tagged by Debian developers as being implemented in the Python language, and all have repositories on GitHub. Therefore, these packages are part of one of the most widely-used GNU/Linux operating systems worldwide, with an upstream language and source code management platform in common.

F.3.2 Data

Our unit of analysis is the software project, n = 182. All projects meet three criteria: they are in the Debian GNU/Linux distribution, included in the dataset published by Champion and Hill (2021), and tagged by Debian maintainers as being written in Python. Due to the manual identification of upstream repositories, our data set was constrained to Debian packages written in the Python language; further research is necessary to study the Debian package ecosystem writ large. The median project in this dataset was 13 years old, with 44 members in their developer communities.

To identify the upstream repository of each package that was the same as the one used by Debian, we first examined Debian metadata: in the Ultimate Debian Database (Nussbaum & Zacchiroli, 2010), on the Debian package website[2], and inside the package as stored in Debian's GitLab instance[3]. If no upstream repository location was identified, we examined the files associated with the package (README.txt if available, and the documentation

[2]https://packages.debian.org/

[3]https://salsa.debian.org/

and license files if not).We then collected package commit and milestone history using the CHAOSS GrimoireLab Perceval tool (Dueñas et al., 2018) and the public GitHub API, respectively. We bounded our data collection within the range of 2/8/2008 - 11/09/2023; February 8, 2008 was the date of GitHub's founding.[4]

F.3.3 Measures

We operationalize project *governance formality* (H_A) using Tamburri et al.'s YOSHI formality score (D. A. Tamburri et al., 2013), one of six metrics addressing project governance (alongside community structure, geodispersion, longevity, engagement, and cohesion). YOSHI has been used extensively in prior empirical studies of FLOSS governance (Catolino et al., 2019; De Stefano et al., 2022; Mauerer et al., 2022; D. A. A. Tamburri et al., 2019; van Meijel, 2021).

$$\text{Formality Score} = \frac{MMT}{MS/LS} \tag{F.1}$$

This measure considers developer responsibility, work processes, and age as component variables to calculate overall project formality, shown in Equation F.1. *Mean Membership Type* (MMT) represents developer responsibility. This is further explained below with Equation F.3. MS refers to work processes, in our case GitHub milestone count, while LS refers to project lifespan in days. The potential range of scores is between 0 and 10,000. However, because this definition would filter out projects who do not use milestones entirely—a substantial portion of our sample— we develop an augmented calculation of the formality score where the MMT is divided by whether or not the project employs milestones (represented by a binary 1:2 classification) (MSE) per the project's age grouping (AG), which is included as a control variable given age may generally impact formalization. Projects were grouped in bins of (1) 0-9 years old (n=44), (2) 9-12 years old (n=44), (3) 12-15 years old (n=49), and (4) 15-16 years old (n=64). The breaks were selected to create bins of similar sizes.

[4]Our data and code are available on the Harvard Dataverse: https://doi.org/10.7910/DVN/WENTBH

Each project was coded with a corresponding group label of one to four. This augmented formality measure is shown in Equation F.2 and the resulting metric ranges between 0 and 10.

$$\text{Augmented Formality Score} = \frac{MMT}{MSE/AG} \tag{F.2}$$

To evaluate *concentration of developer responsibility* (H_B), we draw on a measure used in the overall formality score mentioned above: MMT, which represents the share of project developers who hold more responsibilities via their privileged roles in the engineering process, such as merge permissions. As seen in Equation F.3, this metric is a weighted average of collaborators in a given FLOSS community, with the community defined as all individuals whose committed edits to project files have been accepted in the main branch (D. A. Tamburri et al., 2013; D. A. A. Tamburri et al., 2019). Collaborators are developers who have authored a merge into the primary code branch and thus have the requisite permissions to do so, contributors are commit authors who have never merged into the primary code branch (van Meijel, 2021). Higher MMT averages may suggest flatter organizational structure, since it indicates widely diffused merge activities (Gousios et al., 2014).

$$MMT = \frac{1}{|M|} \sum_{m \in M} \begin{cases} 2 & \text{If } m \text{ is a collaborator.} \\ 1 & \text{If } m \text{ is a contributor.} \end{cases} \tag{F.3}$$

We measure *formal work process management* (H_C) via projects' use of GitHub milestones (D. A. Tamburri et al., 2013), again drawing on the formality score. However, some projects in our dataset are hosted on platforms which lack the milestone metric or do not use it entirely, with prior work indicating only around 20% of projects on GitHub used the milestone tool (Zhang et al., 2020). Thus, we employed two metrics of milestone usage. One was the numeric count of milestones that a given project was using; the other was a binary classification of whether or not a project used milestones. Around 25% of packages studied in this data set used milestones. To measure the *underproduction factor* of a FLOSS project, we use the mean underproduction factor estimates published in Champion and Hill (Champion &

Hill, 2021). As previously described in F.2, the underproduction metric measures a project's engineering activity against the project's importance.

F.3.4 Analytic Plan

Linear regression is an established approach of identifying associative relationships between two continuous variables. In our evaluation of the continuous measures of $\mathbf{H_A}$ (formality score), $\mathbf{H_B}$ (responsibility concentration), $\mathbf{H_C}$(work processes), and project age, we used linear regression models to evaluate the relationship with the project's underproduction factor and evaluate significance at the $p < .05$ level.

Given our constrained data sample, we also employed a statistical power analysis to evaluate the impact of data sample size on our results when relevant (H_A, H_C).

F.4 Results

F.4.1 H_A: Formality Score

We found a statistically significant relationship between a project's score and underproduction factor ($p < .005$). However, the relationship between formality score and underproduction was relatively small, where a unit increase in formality score was associated with only a 0.17 increase in underproduction factor. This result is evidence in favor of our hypothesis H_A, higher formality is associated with increased risk of underproduction.

The analysis reported in Table F.1 uses the augmented formality calculation in Equation F.2, which enabled us to include projects who did not use GitHub milestones, giving us a larger sample size (n=182).

Using the original formulation of the formality score in Equation F.1 would have limited our analysis to projects whose milestone counts are greater than zero; the sample size for this model was 44. Testing H_A using this alternate metric found no statistically significant relationship between the score and a project's underproduction factor. Given our small sample, we conducted a power analysis to assess the impact of sample size in detecting an

effect size of at least $\beta = 0.00017$ (this is the significant effect size of the augmented formality score, but scaled commensurately with the original score's range.) To simulate our data, we assumed that the formality score encapsulated other metrics such as MMT, milestones, and age and that formality score data follow a beta(α:1, β:3) distribution. After running 1000 simulations on data sets of size 75, we are able to reject the null hypothesis. The power analysis provides evidence that, to the extent that this simulated pilot data is representative of the population, our sample size is insufficient to conclude the original formality score's relationship to underproduction, and lends validity to our decision to use the augmented formality calculation displayed in Equation F.2.

F.4.2 H_B: Concentration of Developer Responsibility

A linear regression model fit with MMT values indicated strong statistical significance for a negative relationship between MMT and mean underproduction values, suggesting that as the share of project developers who assume privileged roles (MMT) increases, the factor of underproduction decreases. The results are statistically significant ($p < .002$), where a unit increase in MMT is associated with a 1.38 point decrease in underproduction risk. This result is contrary to our hypothesis H_B, and instead provides evidence that increasing dispersion of responsibility is associated with lower underproduction risk, rather than higher. The effect size (-1.38) is relatively large, given that underproduction scores for our data only range between -5.05 and 2.81, suggesting that this association is also practically significant. Figure F.1 displays the relationship between projects' MMT and mean underproduction factor.

F.4.3 H_C: Formal Work Process Management

A linear regression model fit with projects' mean underproduction scores and project usage of GitHub milestones did not point to a statistically significant relationship between the two measures ($p = 0.09$). However, as with formality score, a lack of significance may be

due to our relatively small sample size (n = 182). Thus we conducted a power analysis simulation to assess whether we had sufficient observations to detect an effect size of at least $\beta = 0.40$; a 5% impact in underproduction factor within our data set. To simulate our data, we transposed MMT into a 0-1 range, modeled the measure's established relationship of -1.38 units of underproduction factor, and assumed MMT data follow a beta (α:5, β:1) distribution. From initial analysis of collected data, we also assumed that milestone data follow a binomial distribution with probability 0.247. Running 1000 simulations of data sets of n=300 we were able to reject the null hypothesis. This provides evidence that, to the extent that our pilot data and simulation are representative of the population, our sample size is insufficient to conclude there is no meaningful relationship between formal work process management (GitHub milestones) and underproduction.

F.5 Discussion

Overall, our results paint a nuanced picture of the relationship between formalization and underproduction in FLOSS projects. Our analysis of formal structure (H_A) using the augmented formality score showed that overall, more formal structures were associated with a minor increase in underproduction risk. These results support arguments from prior work from D. A. Tamburri et al. (2013) and De Stefano et al. (2022) that formal governance concentration is related to increased project risk.

However, our analysis of the distribution of privileged roles across developers (H_B, using MMT) showed that an increase in project developers with privileged roles is associated with decreased underproduction risk. Examining the relationship in Figure F.1 suggests that this effect occurs when MMT is especially high, given the relatively flat relationship when MMT ranges between 0-1.75. One reason higher MMT may be associated with lower underproduction risk may be that the diffusion of privileged roles across more developers may indicate higher commitment to the project overall. For example, cursory analysis of project age shows a small positive relationship between age and a project's risk of underproduction. As a package ages, not only is there a higher likelihood of software adoption, there is also

a higher likelihood of community abandonment. While we cannot make any causal claims, our results suggest the diffusion of project responsibility may help reduce risk of community abandonment and thus, risk of underproduction.

Finally, our analysis of formal work processes as captured by project milestone usage did not yield statistically significant results. However, our power analysis suggests that this was due to sample size, and we believe this is a valuable direction future work to better understand the effects of formalization.

F.6 Limitations and Future Work

In this project, we only examine Python projects which are within the Debian distribution and for which an underproduction factor measure was available. Our analysis was further limited to projects which are hosted on platforms supporting our data collection approach (GitHub/GitLab); while these platforms are widely used hosting platforms for FLOSS projects, these limitations restricted our sample size (n=182). Although our work offers insight into risk around the widely-used Debian distribution, expanding this sample is an important direction of future work. Moreover, the metrics we employ do not account for longitudinal changes in project governance; this remains a topic of further research.

F.7 Conclusion

FLOSS organizations face numerous challenges as they seek to mobilize volunteers to produce secure, high-quality software. We examined three hypotheses about the impact of formality on underproduction risk, finding that project formality may be a relevant indicator of higher software risk, that the diffusion of engineering responsibility was associated with lower risk, and that work product management is an uncertain predictor. Taken together, these results suggest governance informality and broader sharing of responsibility are beneficial to FLOSS projects. Given the importance of FLOSS across software ecosystems, unlocking the complex relationship between FLOSS governance and underproduction offers a new avenue for addressing the cybersecurity risks facing digital infrastructure.

Acknowledgment for Appendix F

This work is indebted to the volunteer developers producing FLOSS who have made their work available for inspection. We also gratefully acknowledge support from the Sloan Foundation through the Ford/Sloan Digital Infrastructure Initiative (Sloan Award 2018-113560 and the National Science Foundation (Grant IIS-2045055). This work was conducted using research computing resources at Northwestern University.

Table F.1: This table displays the relationships between underproduction and three project metrics: formality, MMT, and milestones. Models for H_B and H_C include a control for age with factor variables; 0 to 9 years old is the baseline category.

	(H_A) formality	(H_B) MMT	(H_C) milestones
(Intercept)	-0.78^*	1.65^*	-0.89^*
	$[-1.27; -0.29]$	$[0.06; 3.25]$	$[-1.32; -0.45]$
Augmented formality	0.17^*		
	$[0.05; 0.29]$		
MMT		-1.38^*	
		$[-2.21; -0.54]$	
Age 9-12y		0.07	0.27
		$[-0.53; 0.67]$	$[-0.33; 0.88]$
Age 12-15y		0.60^*	0.79^*
		$[0.01; 1.18]$	$[0.20; 1.38]$
Age 15-16y		1.15^*	1.53^*
		$[0.55; 1.74]$	$[0.96; 2.11]$
Milestone count			0.01
			$[-0.13; 0.15]$
R^2	0.04	0.22	0.17
Adj. R^2	0.04	0.20	0.15
Num. obs.	182	182	182

* Null hypothesis value outside the confidence interval.

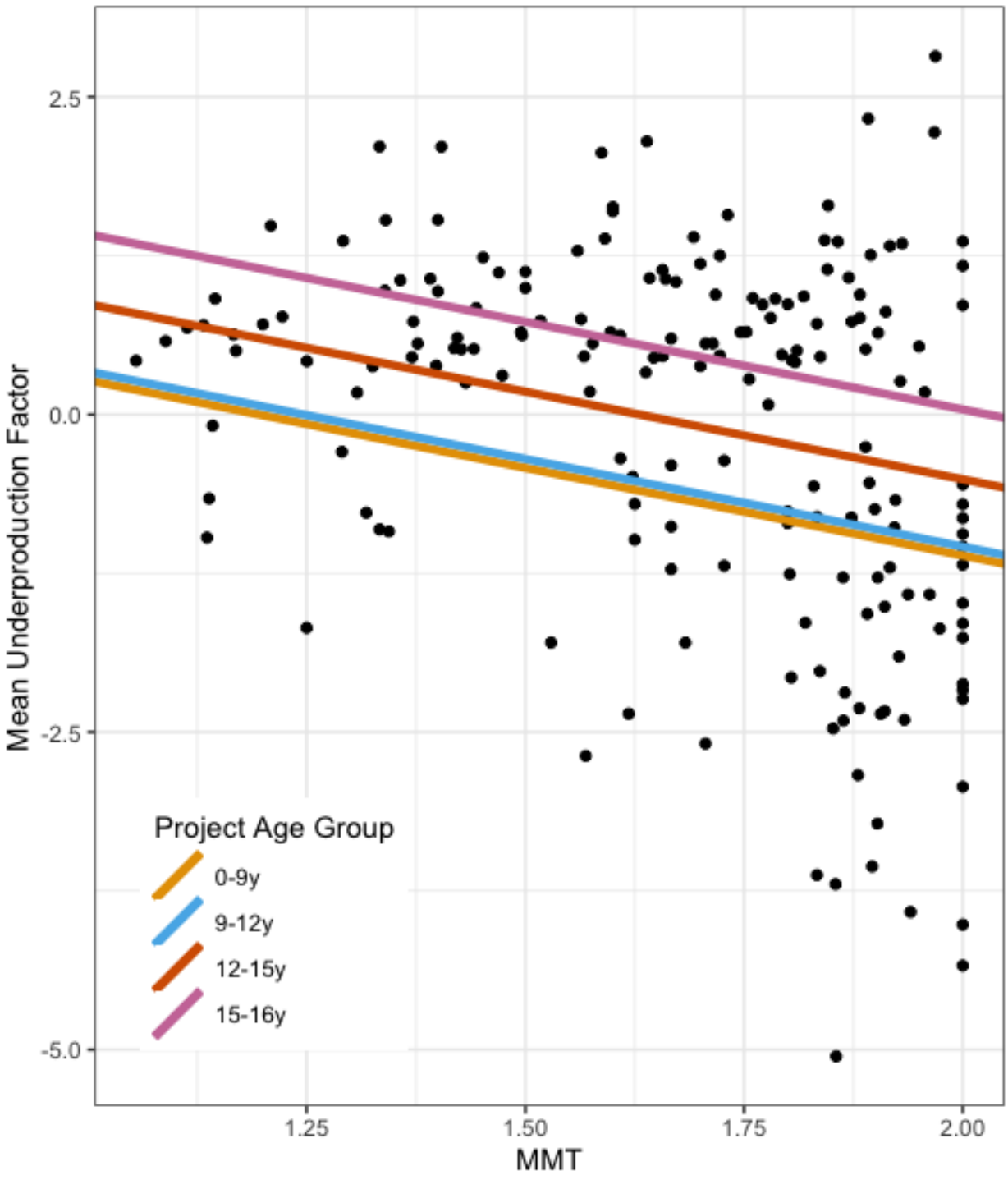


Figure F.1: Plot showing the relationship between projects' MMT and mean underproduction factor. Plotted lines are drawn from the results of our model for H_B.