SOFTWARE ECOSYSTEM SUSTAINABILITY, A SOCIO-TECHNICAL PERSPECTIVE

by

ARMSTRONG TITA, FOUNDJEM

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
July, 2022

Copyright © Armstrong Tita, Foundjem, 2022

Abstract

The last decade has seen a plethora of large-scale software ecosystems (SECOs) developed by distributed teams and organizations dispersed across the globe. Despite the growing success of these SECOs, many questions remain regarding their long-term sustainability, in particular in terms of what socio-technical activities during a SECO's lifecycle are beneficial. This thesis aims to empirically study SECOs' sustainability across four major SECO lifecycle phases, i.e., onboarding, micro-collaboration, release synchronization, and cross-community feedback mechanisms.

SECOs' onboarding process aims to recruit long-term, high-quality contributors. Through an observational study of an OpenStack onboarding event and mining repositories, we found a significant correlation of onboarding with increased gender diversity and patch acceptance rate, while the time until the first commit and bug-proneness of contributions decrease.

Second, we study the so-called `micro-collaboration` practices of distributed development, in which multiple authors contribute to the same commit (instead of to the same branch). To understand the mechanisms, benefits and limitations of micro-collaboration, we performed a mixed-method analysis of collaboration practices in the OpenStack ecosystem through semi-structured interviews and a mining repositories analysis.

Third, open-source SECOs bring value by integrating software projects into a polished

product. However, since each SECO project may potentially have its own release cycle and roadmap, a SECO release team has to set up and enforce synchronization activities to sustain SECO releases. Qualitative analysis on OpenStack's release team's IRC-meeting logs allowed us to identify, catalogue, and document ten major release synchronization activities, which we further validated through interviews.

Lastly, SECOs continuously gather and process feedback from distributors, which customize a SECO product, and end users, for example in the form of bug reports and contributed features. We aimed to understand the effectiveness of these feedback mechanisms through a grounded theory study of unstructured interviews, a survey, focus groups, and mailing lists archives, leading to a theory involving 15 constructs. Validation of our theory using mining analyses suggests that distros contribute a substantial proportion of a SECO's feedback, and that influential distros in a SECO do not focus on one specific feedback type.

Acknowledgments

Most importantly, I am delighted to have had Prof. Bram Adams of the School of computing as my thesis advisor. He has always been a great mentor to my journey in research, technical abilities, and in TAing his engineering courses. I am thrilled and proud to have defended my doctoral thesis with him and in the famous MCIS laboratory. I want to thank the jury members who reviewed my thesis. I dedicate this thesis to my heroin (Tita Kuamen Julienne), my hero (Tita John Babila Foundjem), and most importantly, to the Foundjem family, especially to my sons John Paul and Babila Joel. Many thanks to Prof. Ndonkeu Tita Walter, Sungabi Veronica Tita, and Kwicheu Blandine Tita (my princess) for their constant moral and spiritual encouragement. Finally, I would also like to thank the OpenStack Foundation and Microsoft research for supporting my research.

Statement of originality

I, Armstrong Tita FOUNDJEM, hereby affirm the sole authorship of this thesis. All ideas and inventions attributed to others have been appropriately referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Contents

Abstract **i**

Acknowledgments **iii**

Statement of originality **iv**

Contents **v**

List of Tables **ix**

List of Figures **xi**

Chapter 1: Introduction **1**
1.1 Problem statement 4
1.2 Thesis organization 16

Chapter 2: Related work **17**
2.1 Onboarding 17
2.2 Micro-collaboration 19
2.2.1 Collaboration 19
2.2.2 Developer onboarding 22
2.2.3 Code review and quality 23
2.2.4 Pair programming 24
2.3 Release synchronization 25
2.3.1 Release Engineering 25
2.3.2 Software ecosystems 28
2.4 Cross-community collaboration 29
2.4.1 Grounded theory methodology (GT) 29
2.4.2 Collaboration at open source software communities 30
2.4.3 Cross distribution/projects bug 32

Chapter 3: Onboarding vs. Diversity, Productivity, and Quality – Empirical Study of the OpenStack Ecosystem **33**
3.1 Introduction . . . 33
3.2 Background and related work . . . 36
3.2.1 The SECO onboarding process . . . 36
3.3 Observational study of the top-level SECO onboarding phase . . . 37
3.3.1 Study design . . . 37
3.3.2 **PRQ1:** ***What (and how) are the topics taught during a SECO onboarding event*****?** . . . 40
3.4 Perceived challenges and benefits of SECO onboarding . . . 44
3.4.1 **PQR2:** ***What are the challenges involved with organizing and sustaining a SECO onboarding program*****?** . . . 44
3.4.2 **PRQ3:** ***What are the benefits of a SECO onboarding program*****?** . . . 45
3.5 Quantitative validation of perceived benefits . . . 47
3.5.1 Study design . . . 48
3.5.2 **RQ1:** ***Does onboarding correlate with SECO diversity*****?** . . . 53
3.5.3 **RQ2:** ***Does onboarding correlate with new contributors' productivity*****?** . . . 57
3.5.4 **RQ3:** ***Does onboarding correlate with new contributors' code quality*****?** . . . 60
3.6 Discussion . . . 62
3.7 Threats to validity . . . 64
3.8 Conclusion . . . 67

Chapter 4: A Mixed-methods Analysis of Micro-collaborative Coding Practices in OpenStack **69**
4.1 Introduction . . . 69
4.2 Background . . . 73
4.2.1 About OpenStack . . . 74
4.2.2 The OpenStack development workflow . . . 75
4.2.3 OpenStack contribution and attribution policy . . . 76
4.3 Setup of interviews with OpenStack contributors . . . 78
4.3.1 Selection and demographics of participants . . . 78
4.3.2 Interview recording and transcription . . . 79
4.3.3 Interview coding and reviewing . . . 81
4.4 Qualitative results of interviews . . . 82
4.5 Setup of quantitative study . . . 94
4.5.1 Extracting micro-collaborations . . . 95
4.5.2 Data cleaning . . . 97
4.5.3 Dataset alignment . . . 99

4.5.4 Analysing bug-introducing commits . . . 101
4.5.5 Validation of qualitative findings through multivariate analysis . . . 103
4.6 Quantitative results . . . 110
4.6.1 How do OpenStack contributors engage in micro-collaborative coding practices? (RQ_1) . . . 111
4.6.2 What are the benefits of micro-collaboration? (RQ_2) . . . 115
4.7 Discussion . . . 134
4.7.1 The good . . . 134
4.7.2 The bad . . . 135
4.7.3 The uncertain . . . 137
4.7.4 Validation with OpenStack experts . . . 139
4.8 Threats to validity . . . 141
4.9 Conclusion . . . 146

Chapter 5: Release Synchronization in Software Ecosystems 148
5.1 Introduction . . . 148
5.2 Methodology . . . 153
5.2.1 Subject system selection . . . 153
5.2.2 Communication channel selection . . . 154
5.2.3 Data extraction from IRC logs and etherpad . . . 156
5.2.4 Qualitative analysis of IRC logs and etherpad . . . 158
5.2.5 Preliminary analysis of the catalogue of activities . . . 161
5.2.6 Understanding the OpenStack release strategies . . . 165
5.2.7 Validation using interviews . . . 167
5.3 Catalogue of release synchronization activities . . . 169
5.4 Discussion and implications . . . 196
5.4.1 Discussion of our findings . . . 196
5.4.2 Implication of our findings . . . 204
5.4.3 Generalization of our findings . . . 210
5.5 Threats to validity . . . 220
5.6 Conclusion . . . 222

Chapter 6: A Grounded Theory of Open Source Cross-communities Feedback Mechanisms 225
6.1 Introduction . . . 225
6.2 Background and related works . . . 231
6.2.1 Subject system selection . . . 231
6.2.2 About OpenStack . . . 231
6.2.3 OpenStack community events . . . 233
6.2.4 Related work . . . 234

6.3 Qualitative analysis . . . 237
6.3.1 Data collection . . . 238
6.3.2 Data analysis . . . 243
6.3.3 Qualitative results analysis . . . 246
6.4 Quantitative study approach . . . 253
6.4.1 Approach for RQ2 . . . 254
6.4.2 Approach for RQ3 . . . 256
6.5 Quantitative results . . . 259
6.5.1 **RQ2:** ***How diverse is the feedback contributed by distros?*** . . . 259
6.5.2 **RQ3:** ***What roles do distros play in synchronizing SECO feedback?*** 263
6.6 Threats to validity . . . 267
6.7 Conclusion . . . 268

Chapter 7: Discussion and conclusion **270**
7.1 Discussion . . . 270
7.2 Conclusion . . . 272
7.3 Future Work . . . 275

Bibliography **277**
.1 Questions for guiding the OpenStack interviews on co-authoring . . . 323
.1.1 Demographics . . . 323
.1.2 Generic questions . . . 323
.1.3 Questions for OpenStack foundation members . . . 324
.1.4 Questions for OpenStack practitioners . . . 325
.2 Extended summary of the straussian GT data analysis . . . 326

Appendices **326**

List of Tables

3.1 Contributor-level$*$, SECO-level†, and/or company-level♯ metrics used in our study. 52

4.1 Demographics of interview respondents. 80

4.2 Research questions and associated thematic findings on micro-collaboration practices based on the transcribed qualitative data. For the findings highlighted in grey we will provide additional quantitative empirical evidence in Section 4.6. 84

4.3 Communication media used for collaboration, per number of respondents (n), in percentages (Pct). IRC and Gerrit-comments are the most used media with 53.7%. 87

4.4 Perceived benefits of micro-collaborative coding at OpenStack in descending order of respondents (n) per percentages (Pct). 88

4.5 Mechanisms to improve collaboration practices, per number of respondents, expressed in percentages (%). 93

4.6 Observed variables (metrics) for software quality (q), software productivity (p) and both (b). The productivity metrics marked as "p^a" apply to change sets, while those marked as "p^o" apply to contributors. 105

4.7 The latent variables for quality and their loadings. 107

4.8 The latent variables for productivity at change-set level and their loadings. . 107
4.9 Top 10 of most collaborative OpenStack projects in terms of proportion and number of collaborative change identifiers. 114
4.10 Recognition of collaborative activity of aligned change identifiers. 114
4.11 The presence of bug inducing changes (BIC) in collaborative versus individual coding. 119
4.12 Top 10 OpenStack projects most affected by bug-inducing changes (in terms of their proportion and number). 121

5.1 Overview of the release synchronization activities and their distributions across the Mitaka, Ocata, and Queens releases. Data for Cycle Highlights Management (A10) was not available (N/A) in our coding results since A10 is an activity that we identified after the studied Queens release. 163
5.2 Demographics of interviewed OpenStack experts. The experts either belong to the release or project teams and were **active** during the studied period. 168
5.3 Results of manual classifications to validate the regex matching in both Gnome and Eclipse release synchronization activities (A1, A2, . . . , A10.), we report the range of precision proportion (PP ± 15 %) per activity on a confidence level of 95% and confidence interval of 15 %. 213

6.1 Surveyed and focus group participants (D1-21), and upstream members, comprising Foundation members (Up1-5) and moderators (Mod1-3). 239
6.2 Influential distros in detected communities (DC). 266
6.3 Truck factor for each detected community (DC): NF, FB, RB, EM. 266

List of Figures

1.1 Thesis overview along the SECO lifecycle model with four socio-technical activities: onboarding (Step 1), micro-collaboration (Step 2), synchronization of releases (Step 3), and feedback mechanisms (Step 4). Based on the findings in Step 1, we conducted a follow-up study on onboarding at Microsoft Research (results not included in this thesis). 5

3.1 Materials taught during onboarding and their observed impacts on individual mentees (I), the SECO (E), and companies in the SECO (C). 41

3.2 Timeline of stratified categories used in our study. Cat-1 is our control group, while Cat-2 and Cat-3 are the experimental groups. Each group uses data of seven OpenStack releases (42 months). 50

3.3 (Left) Median GD (in %) of each category ((F)emale, (M)ale, and (N)on-binary) ; (Right) Hired vs. volunteering (Vols) contributors in Cat-1, Cat-2, and Cat-3. 54

3.4 Overview of technical diversity, showing the number of commits made across different numbers of projects (Left) and the number of projects people contribute to, per declared gender (Right). 54

3.5 The evolution of the number of companies (NoC, solid blue line) for each of the 7 studied OpenStack releases before (Cat-1, yellow) and after (Cat-2/3, green) the introduction of onboarding. The black dashed line represents the top NoC responsible for 50% of a release's commits and the red dashed lines shows the total commits per release cycle. 55
3.6 Comparison of commit density between Cat-2 and Cat-3. 58
3.7 Survival curves for time until the first accepted contribution per gender in Cat-2 (left) and Cat-3 (right). 59
3.8 Survival curves for the time until Cat-2 (left) and Cat-3 (right) contributors leave. 60
3.9 Comparison of patch acceptance rate between Cat-2 and Cat-3 61
3.10 An Overview of effort needed by Cat-2/3 contributors. 62
3.11 Likelihood of bug-inducing commits across Cat-2/3. 63
3.12 Radar chart of the studied metrics showing that onboarding (Cat-3) has significant differences and improvements over Cat-2. The metrics are those of Table 3.1: Bug-inducing-commits (SZZ), Effort (Eft), Time to first commit (TFC), Retention (Rt), Patch Acceptance Rate (PAR), Density (Den), Diversity: Gender (GD(f)), Technical (TD), and Corporate Diversity (CD). . 65

4.1 Courtesy of OpenDev 77
4.2 Number of collaborators in collaborative git commits in $\mathcal{C}$; and in Gerrit change sets in $\mathcal{P}$. 112
4.3 Annual evolution of collaboration frequency in OpenStack. Data points along the x-axis indicate the *start* of each yearly interval (e.g 07/2014). . . . 116

4.4 Kaplan-Meier survival curves with 99% confidence interval (shaded areas) for sliding windows of 4-year periods, for junior/senior OpenStack contributors according to their collaboration status. The activity duration is measured in months. 118
4.5 Distribution of #BICs for individual (No) and collaborative (Yes) commits. . 121
4.6 Path diagram showing the structural analysis for the quality SEM model. Factors are shown as circles, observed variables as rectangles, and loadings as solid arrows. 122
4.7 Overall productivity of collaborative vs non-collaborative contributors. . . . 124
4.8 Kaplan-Meier survival curves (with 99% confidence intervals) for the time between the first and the last patch set in a change set, focusing on the first year after initial patch submission. 127
4.9 Time (in days) to finalize change sets for different code churn categories. . . 128
4.10 Time (in days) to finalize change sets for different churn sizes. 128
4.11 Time (in days) to finalize change sets for different file categories. 129
4.12 Time (in days) to finalize change sets for different file sizes. 129
4.13 Path diagram showing the structural analysis for the change-set level productivity SEM model. Factors are shown as circles, observed variables as rectangles, and loadings as solid arrows. 131

5.1 Ecosystem release dependencies across project teams following different/multiple release models. Dev X .X means a given project team at some point in the six-month cycle is developing a milestone/intermediate release. V X.X implies that a project team released a specific version of a deliverable/milestone, etc., at some within the cycle. 151

5.2 Releases team IRC meetings 156
5.3 OpenStack releases 157
5.4 We use inductive coding to build the codebook and deductive coding to compute the IRR using Cohen's Kappa for categorical data. 158
5.5 Excerpt of a weekly meeting IRC log with labels (tags) applied on three sections/paragraphs of the text, and code hierarchy shown on the right. . . . 161
5.6 Affinity diagram showing five of the nine high-level abstractions of the IRC weekly chat logs. Complete diagram is available online 162
5.7 High-level dependency among release activities. Regular lines (Blue) indicate sequential activities, while the dashed lines (Blue) show social activities among A2, A4, and A5, meanwhile, the dashed lines (Red) show activities that are supported by Tools/Automation (A7) 164
5.8 Release team managed models 167
5.9 Cycle with release candidate 173
5.10 Cycle with intermediary 173
5.11 Ecosystem release with projects following different release strategies in a release cycle. 197
5.12 Excerpt from a "*.yaml*" configuration file, showing the Swift project on Rocky release following the cycle-with-intermediary release models. 198
5.13 Release team persona 211
5.14 GNOME release synch. activities 215
5.15 Eclipse release synch. activities 216
5.16 Eclipse time series analysis 218
5.17 GNOME time series analysis 219

6.1 Feedback Loop of OpenStack SECO enables cross-community collaboration between Upstream and Downstream communities 228
6.2 Methodology for qualitative and quantitative analysis. 238
6.3 Summary of GT data analysis, showing how (A) **Open coding**, (B) **Axial coding**, and (C) **Selective coding** generate a theory of how cross-community collaboration drives feedback mechanisms for SECO growth and sustainability. *(See enlarged figures in appendix .2).* 244
6.4 Types of feedback mechanisms. 247
6.5 Characteristics of feedback. 248
6.6 Benefits associated to cross-community collaboration 249
6.7 Challenges associated to cross-community collaboration 251
6.8 Feedback productivity contributed by distros, [L] %distribution of feedback, [R] Pareto CDF per feedback type. 260
6.9 At the top #Bug filed vs Resolved [L] at the bottom, shows a 4-D plot [R] of feedback contributed by distros. Circle size represents relative feedback quantity, and numbers represent distros (without pre-/suffix for visualization purposes). 261
6.10 Centrality algorithms' detected communities for each feedback type. 264

1 Extended summary of the Straussian GT data analysis, showing (A) **Open coding**. 327
2 Extended summary of the Straussian GT data analysis, showing (B) **Axial coding**. 328
3 Extended summary of the Straussian GT data analysis, showing (C) **Selective coding** 329

Chapter 1

Introduction

The last two decades have seen a proliferation of software ecosystems (SECOs) [1–11], from programming-language related ecosystems such as Maven and npm, to operating system distributions such as Debian and Fedora, to infrastructure-related ecosystems such as OpenStack, Linux kernel an Eclipse, and mobile app ecosystems like Android and iOS [12–19]. A SECO can either be proprietary (e.g., Microsoft 365, iOS), open-source (e.g., Apache) or hybrid (e.g., Android), and can span any domain, such as scientific (e.g., R), industrial (e.g., Zephyr), and business (e.g., AWS) communities.

The real strength of a SECO lies in the collective power of actively interacting projects, not just in throwing together a number of unrelated projects. SECOs such as Maven and npm involve a large set of products (libraries) with clear technical and social dependencies on each other, of which a subset has to be selected and installed by the end user directly. Other SECOs, such as the Linux kernel, OpenStack, etc., produce one final product for end users, which is the tight integration of dozens up to hundreds of smaller projects from the SECO. Since each ecosystem's project potentially has its release cycle and strategic planning, it takes considerable **coordination** and **synchronizations** effort at the ecosystem level to release a polished, well-integrated product to the end-user [20].

To achieve successful SECO products, it is essential for the SECO to be healthy. Biologically, the world health organization (WHO) [21] defines health as: "the state of ***complete*** physical, mental, and social well-being and not merely the absence of disease or infirmity" (WHO, 2019). To better grasp this notion of health, it is helpful to use a different metaphor from the domain of ecology [3,22], where SECOs could be likened to an ***environment*** in which ***Interactions***, coordination and collaborations happen between ***Organisms*** (contributors/developers/foundation staff) within their specific niche (project/cross-project teams) [7,23–31]. Moreover, similar to how natural ecosystems evolve, so do its organisms grow, multiply, and start competing with other "organisms" for resources (team members, etc.). Finally, the health of an organism (SECO) can be threatened by toxic behaviour of individuals or other organisms. Upon entering an organism, the impact of such a "poison" on the organism's health and survivability is determined (amongst other things) by the SECO's ability to absorb and block the poison from being distributed throughout the organisms, or even to transform poison into less toxic material (metabolism) [32, 33].

Based on the similarities between ecological ecosystems and SECOs provided by this metaphor, research has approached SECO health from an interdisciplinary dimension, as such benefiting from the rich scholarship (models and theories) in natural ecosystems. Mens et al. [7, 25] proposed three high-level SECO health characteristics with **`social`**, **`technical`**, and **`phenomenological`** components. Moreover, they emphasize the need to consider SECO health at different levels of granularity [25]. The micro-level would gain an insider's view on the socio-technical interactions/collaborations within/among the inter-dependent projects of a SECO (i.e., the phenomenology of each SECO), while the macro-level would study a SECO's health from an external perspective by modeling a SECO's resilience in terms of biodiversity concepts. Diversity and resilience have

been found to have a direct proportional relationship [25], which implies that an increase in diversity relates to an increase in a SECO's resilience.

In other words, by measuring the health of a SECO from a micro- and a macro-level perspective, we are measuring the SECO's ability to stay perpetual under changing environments, such as the evolving needs of stakeholders' requirements [34–37], which is also known as a "SECO's sustainability" [34, 38]. Venters et al. [34] also approach SECO sustainability from both a socio-technical perspective and natural ecosystem. It is generally agreed upon that SECO sustainability should ensure the health of SECOs for both present and future demands and requirements. For this reason, in this thesis the notion of SECO sustainability is considered to be a generalization of SECO health both at the micro- and macro-levels. The phenomenological component, on the other hand, relates to the unique experience of each SECO. For example, a SECO might be experiencing a state of maturity [39–41], while others might be experiencing an influx of new features or more projects requesting to join the SECO (growth and evolution) [42, 43], etc.

Adapted to the context of SECOs [25, 44, 45, 45], this definition suggests that the health of a SECO is a snapshot of its sustainability processes[1] throughout a SECO's life-cycle. In other words, the health of a SECO should therefore span across its socio-technical activities from the onboarding of new contributors and subsequent collaboration during development to the official, synchronized releases of SECO products to end-users, as well as the multiple flows of feedback from end-users and other stakeholders into future releases of the SECO. This snapshot view accounts for projects within a SECO that may become unhealthy under changing conditions at some particular point in time.

[1] these processes include measurable socio-technical activities such as the retention rate, the time to first commit, patch acceptance rate, the rate of reporting bugs vs. fixing bug, the flow of commits or new features, technical diversity, gender diversity, feedback synchronization, the flow of communication among (cross)project-team members, work culture, etc.

By approaching health from a holistic (micro/macro-level) standpoint, SECOs' sustainability becomes of utmost importance and more feasible, yet it is unclear what a SECO needs to ensure sustainability. One reason for this is the many different types of SECOs discussed earlier. Despite these differences, most SECOs feature a closed-feedback community enabling the collection and propagation of feedback. Such feedback is essential to enhance the scope, quality and overall development process of a SECO, and can span from sending email requests for help or reporting a bug all the way to fixing said bugs or even contributing new features [46–48]. This feedback involves the end users, distributors (distros) that package or customize a SECO's products, and the contributors of "upstream" SECO projects themselves.

Due to this common closed-feedback property, many SECOs have a similar lifecycle, hence it is possible/feasible to start empirically measuring sustainability across this lifecycle for a limited set of SECOs, while still deriving valuable lessons for other types of SECOs.

1.1 Problem statement

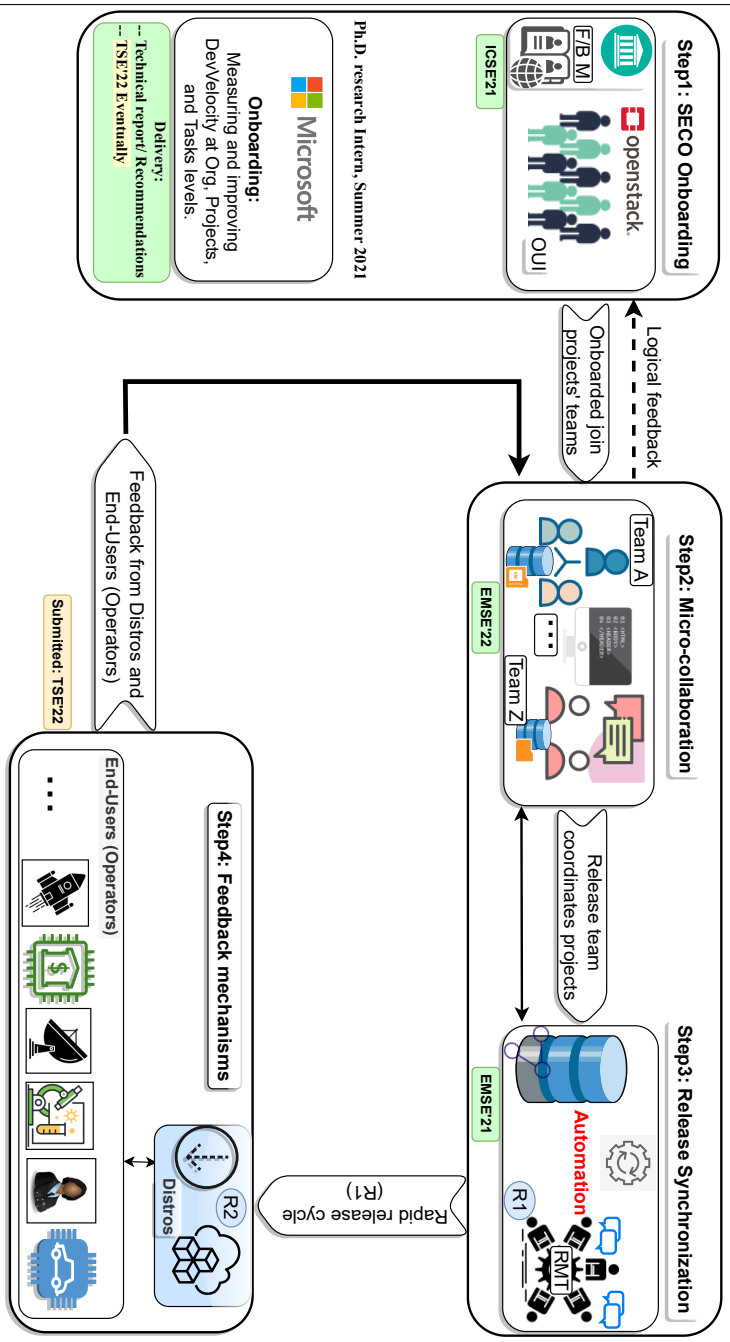
This thesis validates the following claim:

Thesis statement

The sustainability of a SECO relies on a number of essential socio-technical activities along its lifecycle.

In particular, for four major phases in the lifecycle of a SECO (see Fig. 1.1), we analyze the socio-technical activities being performed, then evaluate their effectiveness w.r.t. the SECO's sustainability. These four studies are discussed in a separate chapter each. The first three have been accepted for publication in top-tier venues in software engineering,

Figure 1.1 Thesis overview along the SECO lifecycle model with four socio-technical activities: onboarding (Step 1), micro-collaboration (Step 2), synchronization of releases (Step 3), and feedback mechanisms (Step 4). Based on the findings in Step 1, we conducted a follow-up study on onboarding at Microsoft Research (results not included in this thesis).



while the fourth one is under submission. Here, we discuss these studies in terms of their link with the thesis statement, as well as the more detailed research questions (RQs) that were formulated.

☞ (Step 1) — **`SECO Onboarding`**:

Title: *Onboarding vs. Diversity, Productivity and Quality — Empirical Study of the OpenStack Ecosystem. (Published at the 2021 International Conference on Software Engineering (ICSE)*

Rationale: When experts/senior members within the SECO leave/abandon a SECO (so-called "brain drain") without training new contributors to continue the development process, this might result into a disruption and, hence, an unhealthy SECO team/project. Therefore, stakeholders are encouraged to invest resources to formally train new contributors in terms of the work-culture, the programming style, tools, and processes by instituting a training program on the current SECO's best practices, processes, and tools. As a return on investment for the knowledge invested into the new contributors, SECOs will expect them to provide high-quality code contributions during a prolonged amount of time.

However, it is unclear to what extent setting up an onboarding process is able to achieve those expectations, or, in general, what are the real benefits and challenges of such processes.

Approach: We performed a mixed-methods empirical study to investigate the benefits, challenges, teaching content and strategies of SECO-level onboarding. In our qualitative study, we did an observation study of 72 new contributors participating in an onboarding event of the OpenStack SECO, followed by a quantitative analysis of the OpenStack codebase to find empirical evidence of the perceived onboarding benefits regarding diversity, productivity, and quality of contributors/contributions. To do so, we mined code changes, reviews, and issues of both contributors with and without OpenStack onboarding experience (total of 1,281 contributors) in a period of

over three years. Our study includes three categories of contributors: Cat-1 (control group) comprises people who contributed to the codebase before onboarding was introduced, while Cat-2 and Cat-3 are the experimental groups when onboarding was introduced, where Cat-2 contributors did not participate in any onboarding program, while Cat-3 did participate.

To empirically investigate SECO level onboarding, we guide our study by six research questions:

Preliminary Research Questions (PRQs):

– PRQ1: *What (and how) are the topics taught during a SECO onboarding event?*

– PRQ2: *What are the challenges involved with organizing and sustaining a SECO onboarding program?*

– PRQ3: *What are the benefits of a SECO onboarding program?*

Research Questions (RQs):

– RQ1: *Does onboarding correlate with SECO diversity?*

– RQ2: *Does onboarding correlate with new contributors' productivity?*

– RQ3: *Does onboarding correlate with new contributors' code quality?*

Outcome: Our results have implications for individual contributors, companies, and the SECOs, and we found that onboarding correlates with:

1. Increased contributor productivity;
2. Diversity (gender, technical and corporate);
3. Software quality, and less effort to have commits accepted;
4. A higher retention rate across the three genders (male/female/non-binary) in the SECO and a significant increase in the percentage of accepted change requests (time-to-first acceptable commits).

My unique contribution to this chapter: I performed all the work, under the guidance of my advisor.

☞ (Step 2) — **`Micro-collaboration`**:

Title: *A mixed-methods analysis of micro-collaborative coding practices in OpenStack. (Accepted in the journal on Empirical Software Engineering (EMSE), 17th March, 2022)*

Rationale: During the development life cycle of a SECO release, project team members collaborate to solve complex problems. Collaboration within and between individual project teams [25] usually involves contributors with identical or different levels of experience, for example, expert-expert, expert-novice and novice-novice working together towards a common goal. We aim to investigate if such collaboration relates to an overall improved sustainability of the SECO.

While "macro-collaboration" of developers who submit commits into a common branch is a well-understood practice [49, 50], we focus on an important subset of developer collaboration that we refer to as "micro-collaboration". This practice involves multiple authors contributing to the same commit (instead of the same branch), similar to the agile practice of pair programming, but by collaborators that are not necessarily residing in the same room and that (as we will see) might not even have collaborated in a synchronous way. While maco-collaboration is supported well using today's tool support (e.g., git), micro-collaboration is a practice for which only unofficial practices were available until September of 2017, when organizations like GitHub documented and announced the official adoption of **`Co-Authored-By:`** tags ("trailers") to document micro-collaboration in commit messages. As such, traceability of micro-collaboration activities is an issue, which makes it unclear to

what extent micro-collaboration is beneficial to a SECO.

For that reason, we aim to understand the mechanisms, benefits, and limitations of micro-collaboration practices in SECOs. Such empirical evidence can help practitioners and SECOs understand the value of micro-collaboration and the need for explicit tool support.

Approach: We conduct a mixed-methods analysis, starting with a semi-structured interview of 16 OpenStack practitioners, followed by thematic and content analysis to analyze the qualitative responses. Next, we quantitatively analyze 1M commits from Gerrit code review data (631K change sets and 2M patch sets) of 1,804 OpenStack project repositories over nine years.

This mixed-methods study is guided by three RQs:
– RQ1: *How do OpenStack contributors engage in micro-collaborative coding practices?*
– RQ2: *What are the benefits of micro-collaboration?*
– RQ3: *How can micro-collaboration be improved further?*

Outcome: We make the following contributions:

1. Ten qualitative findings describing the perceived practices, advantages, and shortcomings of micro-collaboration practices at OpenStack;
2. Empirical evidence validating five of these qualitative findings, notably confirming that micro-collaboration correlates positively with:
   - onboarding and retention;
   - a lower likelihood of introducing bugs;
   - more extensive and more code reviews, patch sets, and commits;

3. We observe that the large majority of micro-collaborative changes are not explicitly marked as such using standard git commit trailers **`Co-Authored-By:`**. Thus, we propose a mechanism to identify and re-document micro-collaborations that did not explicitly use the trailers.

My unique contribution to this chapter: I performed most of the work, including qualitative data collection and part of the quantitative data that we used as ground truth. Subsequently, I did both qualitative and quantitative analysis. In addition, I coordinated/led the write-up under the guidance of my advisor, except for the quantitative data collection and initial analysis, which was performed by the second author of this chapter.

☞ (Step 3) — **`Release synchronization`**:

Title: *Release synchronization in software ecosystems. (Accepted in the journal on Empirical Software Engineering (EMSE) 17th December, 2020)*

Rationale: A typical SECO comprises multiple projects, each having its own roadmap and release strategy. While each project can release different versions of the projects' intermediary releases during the development cycle, at set times the overall SECO needs to release a polished, integrated version to the SECO users (e.g., a new version of a Linux distribution SECO).

However, at the end of the SECO release cycle, it is unclear which version of each given project will be available (or shipped) to the end users. For example, constraints of a given project *P* on the API version it uses of another project *A* might not be compatible with the upcoming release of *A* (and its API), in which case the latter release cannot be bundled with the existing version of *P* in the upcoming SECO-level release. Or maybe project *P* could have been working on a new release compatible

with *A*'s new API, but it would only be ready after the SECO's release date.

Such situations, and others, can degenerate and render a SECO un-releasable, and hence not sustainable, unless a clear release synchronization strategy is specified and implemented. Such strategies exist, and typically are managed by a release team, which could either be centralized (managed by the SECO) or distributed (managed by the SECO projects). Existing work has explored the various strategies that are common in SECOs like OpenStack [20] , however it is unclear what are the main activities a SECO's release team has to perform in order to implement and achieve such strategies, in order to keep the SECO's release on track.

Approach: We did a mixed-methods empirical study of the release strategies that the centralized SECO release team uses to synchronize the releases of the projects in the context of the OpenStack SECO. We use qualitative analysis on the release team's IRC meeting logs that cover two OpenStack releases (study period of 1 year). Furthermore, we validate our findings through interviews with eight active OpenStack senior practitioners from the release and project teams. Next, we quantitatively analyze 22 OpenStack releases and all the project repositories to find evidence of release synchronization activities and why multiple release strategies co-exist in SECOs.

To identify the "how" and "why" of the release synchronization process at the ecosystem level, specifically, we asked the following RQ:

– **RQ1**: *What does it take for an ecosystem release team to manage synchronized releases across time successfully?*

Outcome: We make the following contributions:

1. Our results suggest that even though an ecosystem's power lies in the interaction of inter-dependent projects, release synchronization remains a challenge

for both the release team and the project teams;

2. We catalogue ten release synchronization activities that the release team performs in varying proportions each release cycle
3. We provide evidence (and rationale) of multiple release strategies co-existing within a complex SECO;
4. We constructed a persona for a typical release team in a complex SECO, which will enable managers and SECOs to hire the most suitable candidate for the challenging role of release team member.

My unique contribution to this chapter: I performed all the work, under the guidance of my advisor.

☞ (Step 4) — ***Feedback types/mechanisms***:

Title: *A Grounded Theory of Open Source Cross-communities Feedback Mechanisms. (Will be submitted to IEEE Transactions on Software Engineering.)*

Rationale: A SECO's products can reach end users either directly, or via customized products built and distributed by distributions (distros). A common example would be the Linux kernel, which can be downloaded and installed directly or via one of the many Linux distributions. However, getting a SECO release into the hands of end users is not the end goal, but actually the beginning, since this is when a variety of feedback types flow into the SECO, trying to find their way to the upstream project development of the SECO in some form: bug reports, emails with questions, complaints or feature requests, bug fixes, new features, etc.

While it is imperative for upstream and downstream communities inside a SECO to be in continuous synchronization, ensuring a sustainable SECO, to date not much is

known about how these feedback are synchronized between upstream and distros and among distros. Currently, there is a gap in the literature on a theory to explain the relationship between cross-community collaboration and feedback mechanisms. In particular, SECOs need a mechanism to identify influential distros/projects and the various feedback communities within a SECO, which is paramount for its sustainability.

Approach: Our mixed-methods study comprises a Straussian grounded theory methodology that analyzes qualitative data in-depth using four qualitative methods: open-ended interviews, survey, focus groups, and qualitative coding of textual mailing list data to understand how SECOs sustain cross-community collaboration. Next, we quantitatively explore the feedback of 21 distros contributed to upstream to 1,283 OpenStack repositories. This feedback comprises 142,914 filed bugs, 65,179 resolved bugs, and 4,349 new features from the issue trackers, and 140,361 emails.

We address the following RQs:

– **RQ1:** ***How do SECOs sustain cross-community collaboration****?*

– **RQ2:** ***How diverse is the feedback contributed by distros****?*

– **RQ3:** ***What roles do distros play in synchronizing SECO feedback****?*

Outcome: The outcome of our qualitative study (RQ1) is a theory grounded in data that we quantitatively validated (RQ2/3).

1. "Cross-communities collaboration Requires synchronization and communications on diverse feedback mechanisms to grow and sustain SECOs."
2. We found 15 constructs in the emerged theory, from four distinct categories, which suggests four benefits and seven challenges of growing and sustaining cross-community collaboration, and 16 types of feedback mechanisms in four

sub-categories, supporting nine change characteristics in two sub-categories.

3. Two of these 15 constructs describe the types of feedback mechanisms (T1) and the types of actual feedback made by distros (T2)
4. Moreover, we quantitatively validate our theory by providing empirical insights that suggest that:
   (a) Distros are essential and productive in cross-community feedback, contributing 25.6% of all new features in the OpenStack ecosystem, authoring 30.7% emails relating to technical topics, reporting and resolving 44.3% and 30.7% bugs respectively;
   (b) The feedback contributions from distros follow a Pareto distribution (power-law) [51] with 80% of all downstream changes coming from 20% of the distros;
   (c) While nine distros provide the four types of feedback in equal proportions, 12 distros are dominated by one type of feedback;
   (d) Wasted effort is a pain point for distros, since they spend an extra fix time of 12 days to resolve wasted effort;
   (e) Surprisingly, influential distros are not the most dominant in their respective feedback type communities; they play general-purpose roles in synchronizing feedback;
   (f) influential distros are associated with a higher truck factor (TF) ($\geq$3) in terms of produced feedback, which is unusually higher than for regular open-source projects.

My unique contribution to this chapter: I performed all the work, under the guidance of my advisor.

1.2 Thesis organization

The remainder of this thesis is structured along the four major phases of a SECO lifecycle studied by the thesis; for each, we empirically study the degree to which socio-technical activities in place relate to SECO sustainability. After discussing the state of the art related to the entire thesis in chapter 2, chapter 3 discusses SECO-level onboarding. Then, chapter 4 presents a detailed socio-technical analysis of micro-collaboration at SECO-level, and in chapter 5 we discuss our mixed-methods study on SECO release synchronization. Lastly, chapter 6 presents our socio-technical analysis on cross-community feedback mechanisms using grounded theory, social network analysis and influence maximization. Finally, we draw conclusions and discuss potential avenues for future research in chapter 7.

Chapter 2

Related work

This chapter discusses related work of the four major parts of this thesis, in particular SECO-level onboarding (chapter 3), SECO-level micro-collaboration (chapter 4), SECO release synchronization (chapter 5) and cross-community feedback mechanisms (chapter 6).

2.1 Onboarding

Prior studies mostly focused on the project-specific onboarding phase.

Sharma et al. [52] explored the relationship between successful (short-term) onboarding results and job satisfaction (contributors' intention to either leave or remain active with an organization). Their results suggest that job satisfaction is directly related to both onboarding success and turnover intention. However, they found no relationship in workplace quality. Our study identified eight benefits of onboarding at the SECO level and found that contributors who did onboarding stay longer in the SECO than those who did not.

Fagerholm et al. [53] explored onboarding in a pilot program organized and sponsored by Facebook (under the Education Modernization Program for OSS projects) in collaboration with universities across the globe. A study conducted with 120 students showed that

participants who were deliberately mentored during the entire onboarding process were more motivated and committed than their counterparts who did not follow the onboarding process. Our study also shows that contributors who did onboarding were self-motivated and more productive than those who did not do onboarding.

Viviani et al. [54] took a different approach and focused on onboarding in smaller companies that follow a fast software release cycle. They observed a stronger bond among developers, mainly due to close mentoring relationships between core reviewers and younger developers. Contrarily, our study focuses on large and complex SECO. However, we also found new contributors collaborating with mentors (expert-novice collaboration) and expert-expert and novice-novice collaboration.

Britto et al. [55] adopt a model to measure the state of onboarding in software organizations. Steinmacher et al. [56] qualitatively study systematic literature reviews and responses from various practitioners (including an interview study) across several OSS projects to understand the obstacles new developers in an ecosystem from actively contributing to projects. In our research, we found 13 challenges associated with SECO onboarding.

Using the GitHub ecosystem as a case study, Casalnuovo et al. [57] investigate the effects of socialization on a developer joining a new project, a process which the authors refer to as onboarding. They analyze the information of 1,255 developers contributing to a total of 58,092 GitHub projects. Their analysis shows that both the social and technical factors of prior connections and experiences that developers established with experienced team members of a new project have a lasting effect that substantially affects these new members' productivity. Our work found that contributors who participated in the mentoring program were more productive than those who did not participate.

Labuschagne et al. [58] studied the impact of the onboarding program at Mozilla and

found that onboarding does not relate to contributor retention. They did not control for prior experience or self-motivation of contributors. At the same time, we show that self-motivation and commitment are challenges SECOs should manage. The onboarding program correlates to a high retention rate, productivity, quality, and diversity.

On the other hand, Zhang et al. [59] studied how companies collaborate within OpenStack by measuring productivity at the release level (while we focus on the release before OpenStack introduces onboarding). Even though their work is not directly related to onboarding at the SECO-level, it, however, explores contributors' paid and volunteered productivity, which, in our case, refers to project-level mentoring for onboarding contributors.

Given that related work has focused mostly on project-specific onboarding, this chapter first studies in detail the top-level SECO onboarding phase through an observational study.

Onboarded SECOs' participants can start contributing to the codebase after obtaining both the (i) top-level and (ii) project-specific know-how. Thus, we quantitatively study the correlation between their later contributions (in terms of productivity, code quality, and diversity) and the *overall* onboarding process that they followed.

2.2 Micro-collaboration

2.2.1 Collaboration

Young et al. [60] identified four models for contributor attribution across open source projects. The authors suggest that these models, ranging from technical platforms such as GitHub to non-technical platforms such as the ad hoc model (e.g.for board members or sales persons) could identify all possible types of software development contribution, which usually have multiple dimensions. The authors used a standard "All Contributors" (AC) model to analyze the entire lifecycle of thousands of open source projects, in order

to capture a wide variety of contributions from diverse sources (outreach, finance, infrastructure, and community management) and identify the differences with traditional ways of measuring contribution. Their findings suggest that attribution systems that are designed with the community in mind allow to make contributions more visible. The authors also claim that models that require a more explicit attribution, in principle, discriminate against the morality of what is and what is not a contribution. Similar to Young et al., our work focuses on micro-collaborative coding practices including contributor attribution. In line with the previous claim, we provide quantitative evidence on contributions that OpenStack didn't explicitly attribute to contributors.

Zhang et al. [59] studied how collaboration happens between large open source ecosystems (including OpenStack) and contributing companies. Different clusters of collaboration emerge among thousands of contributing companies and the OpenStack projects. This qualitative study identified four recurrent roadmaps, which contributing companies have adopted as their business models to becoming dedicated contributors to OpenStack. Companies could collaborate intentionally, passively or choose to stay committed in isolation. Based on a collaborative network graph, thus, the authors show how the position of a company correlates positively to productivity within the ecosystem. In addition, Zhang et al. sheds light on the complexities of collaboration within such a large ecosystem. In contrast to studying collaboration at the company level, this thesis focuses on micro-collaboration between individuals in code reviews.

Whitehead [61] argues that software engineering is a collaborative process by nature, requiring strong coordination among many software developers to release large-scale software artifacts. The past two decades experienced a rise in web-based tools that advance collaboration in software development. Despite this, there is a lack of empirical evidence

on the benefits of such collaborative tools and processes. Our work provided such empirical evidence, for developer collaboration in the OpenStack ecosystem in particular.

Similar to Whitehead, Mens et al. [62] reflected on the collaboration and coordination are processes during social coding. They draw the attention of practitioners in large-scale software development on the importance of constant communication and interaction. However, they do not provide concrete implications of or reasons for collaborative coding. Our work fills in the gaps by providing both qualitative and quantitative evidence on why and how developers carry out micro-collaborations in a large open source ecosystem, OpenStack.

Avelino et al. [63] investigated developer collaboration in the Linux kernel. They found that a small portion of developers make significant contributions to the code base and that authors with a high number of co-authored files tend to connect with others with fewer connections. Avelino et al.consider as co-authorship instances cases where multiple authors touch the same file. In contrast, our work focused on the two different kinds of micro-collaboration instances (through co-authorship of git commits, or collaboration on code reviews).

Overall, previous work on collaboration has focused either at a high-level view of collaboration between companies, or at attribution models and tools. This article investigates collaboration on a fine-grained level and by doing so, uncovers potential challenges of micro-collaboration in practice. We shed light on micro-collaboration benefits and drawbacks for the development process, as well as for the tools supporting it. Future work should investigate how the collaboration practices that have shown to be beneficial to OpenStack can be transferred to other communities, and how the difficulties due to current tool support and lack of community awareness about micro-collaboration can be mitigated.

2.2.2 Developer onboarding

Foundjem et al. [64] used a mixed-method analysis to understand the process, benefits and drawbacks of OpenStack's onboarding program. The authors highlighted that onboarding comprises two levels: the ecosystem-level event (teaching overall processes and practices) and the project-level (project-specific technology and practices) onboarding process. They conducted an observational study to classify the ecosystem-level activities of 72 contributors aspiring to join OpenStack. The authors quantitatively validated the perceived benefits by comparing diversity, productivity and quality measures of contributors that participated in onboarding, and those that did not. Results showed that onboarding correlates significantly with increasing gender diversity, productivity and code quality. The authors also focused on project-level onboarding to verify whether collaborative coding helps to improve the onboarding process. They found a statistically significant correlation between micro-collaboration and prolonged activity, as well as evidence of micro-collaboration practices between novices and experts. The current chapter expands on these results about project-level onboarding at OpenStack by examining the relationship between micro-collaboration and other factors such as the quality and productivity of code contributions.

Casalnuovo et al. [65] mined git repositories for empirical evidence that prior social communication with project developers facilitates project onboarding. Focusing on 1,274 developers with a long GitHub contribution history, the authors examined the development history of 58,170 projects that these developers participated in. Pre-existing relationships were found to strongly affect developer migration to projects. It should be noted that the authors considered a single author per git commit, so they did not take into account possible co-authorships (such as those expressed in git commit trailer messages).

Steinmacher et al. [66] performed a systematic literature review on onboarding in open

source projects. They depict onboarding as a sustainability mechanism for open source projects. They identified two categories of contributors and the major barriers that these contributors face during onboarding. The most evidenced barriers were newcomers' previous technical skills, receiving response from community, centrality of social contacts, and finding the appropriate way to start contributing. While these barriers form a baseline, Steinmacher et al. highlight that further empirical evidence on newcomer onboarding is needed.

2.2.3 Code review and quality

Henley et al. [67] carried out a mixed-method analysis on the documented challenges and limitations that modern code review faces. The authors propose CFar, a tool to enhance human collaboration during code review. The tool forms an integral part of the collaborative software design process, incorporating an automated code review mechanism based on program analysis to generate feedback to the reviewer. Through deploying CFar at various professional sites, the authors measured the tool's effectiveness and reported that it correlates with an improved communication, productivity, and review quality.

McIntosh et al [68] quantitatively studied code review coverage and participation in the context of three software projects. They found significant correlation between software quality and code review coverage and participation. This provides empirical support to the intuition that badly performed code review negatively correlates with software quality in sizable projects that use modern code review tools. Our own study also suggests that software quality is a complex process with multiple facets to consider, moreover our SEM models found positive relationships between micro-collaboration and both *reviewProductivity* and *reviewActivity*.

2.2.4 Pair programming

Pair programming has been advocated by the agile software development community as a technique that can improve the productivity and transfer of knowledge within open source projects. Plonka et al. [69] highlight that knowledge transfer can happen at any level of collaboration, but it is particularly valuable in a novice-expert configuration. The authors carried out an observational study and interviews in four major software companies. Their findings suggest that knowledge transfer exist within pair-programming in six teaching strategies, ranging from incidental to straightforward suggestions.These strategies are an abstraction of major teaching methods that are encouraged in a cognitive apprenticeship. Cognitive apprenticeship is a concept that describe the knowledge transfer process where a skilled master teaches his skills to an apprentice.

In a similar study, Lui and Chan [70] aimed at comparing the productivity between novice–novice and expert–expert relations. The authors carried out a controlled experiment, called *repeat-programming*, that aims at facilitating the socio-technical factors that human beings experience with respect to efficacy. Experimental findings suggest that the productivity gain (in terms of time reduction and software quality improvement) is much higher for novice–novice pairs (compared the baseline of novice individuals) than for expert–expert pairs (compared to the baseline of expert individuals). The results align with prior studies, suggested that pair programming is effective in increasing practitioner productivity, especially between novices, or experts that need to solve programming problems outside their area of expertise.

Spohrer et al. [71] examined pair programming and peer code review in collocated teams of a global software firm. The authors performed on-site interviews to solicit information about perceived patterns of team collaboration, specialization and knowledge

coordination. They found that teams applied pair programming and code reviews in varying ways; the team's structure was theoretically linked to the structural properties of the technology underlying the techniques. Similar to our work, they investigated collaboration practices, but based on a quite different case study with different characteristics, and focused on the communication in pair programming and code reviews. The micro-collaboration practices investigated in this chapter align with the ones investigated by Spohrer et al., and the findings from our OpenStack case study enrich the body of knowledge on collaboration practices.

2.3 Release synchronization

2.3.1 Release Engineering

Adams et al. [72] studied the various activities involved in integrating packages within the Debian, Ubuntu, and FreeBSD open-source software distributions. Like our study, the authors performed a qualitative analysis to identify major activities of the studied process (integration). Yet, they focused on the package release notes instead of the actual communication logs between maintainers. While Adams et al. identified and documented seven major integration activities, typically performed by the maintainers of an individual the open-source project, we instead focus on the synchronization of releases *between projects and the overall ecosystem*.

Adams and McIntosh [73] discuss how modern software projects are released using continuous delivery strategies and the major research challenges in this area. However, they do not consider ecosystem-level releases. The authors suggest that empirical studies are vital to understand and (possibly) enhance release engineering practices, which our chapter aims to do for ecosystem releases. Khomh et al. [74] aims at evaluating the relation

between faster releases and software quality, which they found a link but at the project level. Nothing is yet known at the ecosystem level. However, our study has not yet evaluated a relationship between the nine release activities and software quality.

Other studies such as Franch and Ruhe [75] studied Release Planning and Tracking and the relation to software quality. Also, Rahman et al. [76] analyzed the relationship between the release stabilization activity and the quality of software, albeit at the level of individual projects. However, our study has not yet evaluated a link between the ten release activities and software quality. We plan to conduct this study in future work.

Poo-Caamano et al. [77] studied the release communication and coordination process of the GNOME ecosystem, which follows a time-based release strategy. Through an exploration of the different communication channels used by developers and the release coordination team to manage GNOME releases, we found four significant challenges facing the GNOME release team: (1) difficulties in coordinating project teams (2) problems in handling the build process, (3) challenges in tracking unplanned changes, and (4) testing the GNOME deliverables. To overcome these challenges, the authors suggest that release team members should have excellent socio-technical communication skills and diverse knowledge of ecosystem-wide concerns. However, these four challenges translate to three synchronized activities in our work (Planning and tracking, Dependency management, and Stabilization) [75], [76], and [78]. Poo-Caamano et al. interviewed 10 GNOME developers to triangulate their findings. Similarly, we also interview eight experts from the OpenStack ecosystem to validate our findings. Nevertheless, the authors did not generalize their work but advocate that future research considers comparing their results against other ecosystems through analytical generalization. Our work fulfills this request by comparing our ten activities against different ecosystems, including GNOME.

Teixeira et al. [79] synthesized online web resources to document the release process and infrastructure used in the famous OpenStack ecosystem. They explore a spectrum of release strategies for OpenStack's projects, from utterly the independent to the product-wide concern, which is fully managed by the central OpenStack release team. Moreover, they emphasize the time-based model at the expense of the other four models that the release team manages. While the release team is deemed essential to achieve 6-month releases of the OpenStack ecosystem product, Teixeira et al. did not focus on the specific synchronization activities and challenges experienced by the OpenStack release team. Mostly because the online web resources that they consulted do not provide information on release synchronization activities, they also extensively explore how OpenStack uses the release notes manager (Reno) across different project teams to automate release notes.

However, this online documentation is limited in capturing the real socio-technical challenges during each release cycle. Teixeira et al. have documented the "what" about the OpenStack ecosystem's rapid release process. They did not study the "how" about the release process. For example, they found evidence online of the different release strategies that co-exist in a complex ecosystem but did not determine why and how they exist in a complex ecosystem such as OpenStack.

Therefore, our study complements theirs by providing empirical evidence on why and how multiple release synchronization strategies co-exist in a complex ecosystem. We also found ten release activities that the release team performs each cycle to synchronize ecosystem releases.

2.3.2 Software ecosystems

Zhang et al. [59] studied large companies' involvement in open source ecosystems. The authors investigated the OpenStack ecosystem and the consortium of companies that collaborate to build a sustainable ecosystem. In particular, the authors identify networks of collaboration ("clusters") and four models that collaborative companies within OpenStack have adopted in engaging with the ecosystem. Consequently, they suggest that there are three ways that companies can participate within the ecosystem; "intentional" or "passive collaborations", or in "isolated fashion". Therefore, these collaborative networks help companies boost company productivity within the OpenStack. The works of Zhang et al. did not focus on release synchronization and did not explore the release strategies used by OpenStack. However, the authors were able to show that companies' massive involvement and global engagement increases the complexity of an already challenging task that the central release team faces while coordinating globally distributed teams of contributors/-collaborators with different time zones.

A substantial body of work exists on `Dependency Management`, the strategy that we identified within the weekly IRC logs. For example, Constantinou and Mens [78] investigate the dynamics and socio-technical evolution of the software ecosystem by measuring the impact of permanent changes in an ecosystem on the ecosystem as a whole, its developers, and their source code. Results show a significant impact, especially when contributors abandon a project or migrate to another ecosystem.

Decan et al. [80] empirically examine the inter-dependencies of packages in three programming language ecosystems over time and show the extent to which an ecosystem can deteriorate due to package dependency issues.

Bogart et al. [81] show that changes introduced in a package can ripple through an

entire ecosystem, which may cause a refactoring of the package. Their results show a design lag in building an ecosystem, especially regarding the policies and infrastructure. Also, the authors claim that building a transparent community can resolve conflicts and change-related costs.

Our research identifies the nine major release synchronization activities used in the thriving OpenStack ecosystem, which (similar to Bogart et al.) involves ecosystem governance and dependencies. OpenStack deals with the ripple-through problem by the use of centralized dependency specifications.

2.4 Cross-community collaboration

2.4.1 Grounded theory methodology (GT)

Strauss-Corbinian GT is gaining grounds in software engineering research as a common methodology for grounded theory analysis [82–90], due to its rich intersection of the social and technical dimensions.

For example, GT has been used extensively by Masood et al. [85], Shastri et al. [90,91], and Hoda et al. [92] to study agile software development. Masood et al. collected data from three sources: interviews, participant observation, and scrum guide (*by the book*) and found variations across three core categories between Scrum theory and how it is practiced. Hoda et al. use GT to investigate how Agile teams self-organize, with members repeatedly self-organizing themselves and functioning outside their boundaries, with one person playing multiple (different) roles at different time intervals as needed. The authors use interviews and participant observation to collect data from 58 Agile practitioners across 23 software organizations to generate a theory, which identifies how informal, implicit, transient, and spontaneous roles make Agile teams self-organized.

Palomba et al. [84] did a mixed-method study to understand developers' concerns and how those affect their decisions to eliminate or preserve code smells. The authors use Straussian GT qualitatively with data from a survey to study community smell and quantitatively, and mined nine open source repositories to show that community smells can influence code smells' severity.

In particular, our approach of GT is greatly inspired by Masood et al. We did a mixed-method study with an open source SECO upstream and a mixture of downstream communities to understand cross-community collaboration and feedback mechanisms. GT allowed us to focus on exploring how cross-communities collaborate to drive feedback mechanisms in a complex SECO and to identify four benefits and seven challenges.

On the other hand, Rodríguez et al. [88] take a different epistemological stand, by using a different variant of GT (Glaserian version —positivist approach) to understand the concept of value-based feature selection mainly when used in the software development process to select and prioritize features based on their values. The authors collected data from three large-scale software organizations practicing value-based decision-making using semi-structured interviews with 21 decision-makers. Their resulting analysis led to a theory "of value for value-based feature selection (software features) in software/software intensive product release planning" of value-based feature selection in software intensive product release planning that is explained by six constructs. However, Glaserian GT is not suitable for our **socio**-technical activities because of its strong positivist stand.

2.4.2 Collaboration at open source software communities

Prior works [93–95] have studied collaboration in open source such as OpenStack SECO, WebKit communities, etc., along various dimensions and levels of abstraction.

Zhang et al. [96] empirically studied upstream commercial participation in OpenStack projects, by mining projects' Git repositories and conducted a survey with OpenStack developers. Authors found eight contributions models that drives competing companies motivations to OpenStack, and argue that while competing companies contribute over 90% of commits to upstream as compare to volunteers, these companies can still collaborate while maintaining a balance between the SECO's long-term sustainability goals and maximizing business profit. The authors stated that a majority of these companies are hardware manufacturers such as Dell, Intel, etc., and other e-commerce platforms like Walmart, eBay, etc. Similar to our work, Zhang et al. found that distros ("full-solution providers") are key players among the eight models that define companies collaborate scheme with upstream projects. On the contrary, they reported that a single company contributes over 50% of commits to upstream projects, which contrast our finding and previous works [93, 97] and the upstream policy. Therefore, we call on the authors to verify this claim. Both Zhang et al., and Teixeira [94] use social network analysis to show collaborative patterns in OpenStack's upstream, and both lines of work agree that competitive companies can still collaborate upstream in the same project.

Our work differs from Zhang et al. in a number of major ways. Firstly, they focused mostly on upstream collaboration during the development cycle instead of considering significant roles that distros play in contributing feedback to upstream. We explore a wide varieties of data sources and venues to draw our conclusions. We also found and studied different types of collaborations (upstream-upstream [93], upstream-downstream and downstream-downstream), of feedback provided by distros and of feedback mechanisms used to collaborate. That said, Zhang et al.'s work provides an excellent base and inspiration for our work on SECO feedback mechanisms. Both works agree that dominating

companies risks the sustainability of a SECO if abandon.

2.4.3 Cross distribution/projects bug

Cross distribution [95, 98] SECOs are becoming increasingly popular, for example, in the Linux [99], Android [100], and OpenStack [95] distributions, etc., such SECOs and the fact that they keep on growing leads to various cross-distribution issues, some of which have been studied in literature Boisselle and B. Adams [99] quantitatively analyze cross-distribution issue report duplicates across the Ubuntu and Debian bug repositories. Using the within-project approach to detect duplicate in the context of cross-distribution duplicates, and obtain an optimal precision and recall of 43% for an 88% threshold. The authors estimate the amount of wasted time that contributors/users lost waiting for a fix that already exist: a median of 38-47 days.

Ding et al. [101] empirically analyze the scientific Python ecosystem (upstream/downstream cross-project bugs) to understand the distros' workaround characteristics of upstream fixes. The implications of this study guide practitioners to understand cross-project bugs in software ecosystems. Results show that workarounds have four patterns, three categories of workarounds bugs exist, and upstream fixes are significantly different in terms of LOC and code structure.

Our study found evidence of distros collaborating among themselves in reduced form and we also show that wasted efforts are pain-points to distros involved in cross-community collaboration, as it takes an extra fix time of 13 days to discover a redundant bug.

Chapter 3

Onboarding vs. Diversity, Productivity, and Quality – Empirical Study of the OpenStack Ecosystem

This chapter empirically studies the socio-technical activities in place for SECO-level onboarding, then validates the extent to which those activities relate to diversity, productivity and quality of contributors and contributions in a SECO, as concrete measures of SECO sustainability. This chapter has been published at the 2021 International Conference on Software Engineering (ICSE) [97].

3.1 Introduction

Substantial research conducted by both the academic and industrial sectors over the past two decades has attributed most of the success of open-source software (OSS) projects and ecosystems (SECOs) to the strong involvement of contributors, both volunteers and paid employees of involved companies [59, 102–104]. Apart from attracting and retaining talented contributors, another major challenge faced by software projects and SECOs is the practical training of new contributors [105, 106], specifically, the onboarding experience of new contributors.

Despite sharing similar goals, SECO-level onboarding programs differ from onboarding programs of individual projects since a SECO is not just the sum of its parts but also "a set of independent and interrelated OSS projects working together for a common objective" [102]. On the one hand, individual projects use different workflows and technologies (requiring different skill-sets) and have independent sets of features and release roadmap. On the other hand, projects have to collaborate with other projects that they depend on. Such cross-project coordination implies the need for onboarding to cover inter-disciplinary processes and tools, compared to the more domain-specific training individual projects provide. SECOs have to ensure that, despite differences in roadmaps, all of their projects can be integrated at set times and can achieve major SECO milestones such as a joint SECO release (e.g., Eclipse, OpenStack, Linux distributions).

Thus, SECO-level onboarding programs should enable new contributors to learn and master both the general SECO processes and concepts and the specific workflows and tools of the individual SECO project(s) in which they want to be active contributors. Several existing works have explored onboarding as an event within proprietary and open-source software communities [52, 106–108]. However, these studies focus on individual projects. Only a few studies have investigated the benefits and drawbacks of contributors' (one-time) onboarding event in large organizations [53,57]. Thus, little is known yet about the benefits and drawbacks of contributors onboarding in the context of SECOs.

Therefore, we aim at reducing the gap in current literature regarding **understanding the process and impact**[1] **of onboarding in/on open-source SECOs** by conducting an empirical study of the OpenStack SECO. We choose the OpenStack SECO among other contenders such as GNOME, the Apache foundation, Eclipse, CRAN, or the Linux kernel

[1] Any usage of the words "impact", "influence" or "enhance" refers to the correlation sense of these terms, and does not imply causality.

because it is one of the world's fastest-growing open-source software ecosystems [109]. OpenStack has over 100K community members distributed across 182 countries, managed by a consortium of about 693 supporting companies, and organizes two major onboarding events yearly in different geographical locations.

First, we follow a mixed-method research approach by first performing a direct observational study of 72 new contributors to identify the activities performed during a two-day OpenStack onboarding event and identify any perceived challenges and benefits of SECO onboarding. Next, we conduct a quantitative study of the submitted code changes, code reviews, and issues of 1,281 contributors of the OpenStack ecosystem to measure the correlation between onboarding experience and contributor diversity, productivity, and contribution quality.

Our findings show that the OpenStack SECO uses a wide variety of content and strategies to train new contributors during SECO-level onboarding, trying to address 13 challenges involved in SECO onboarding. We also identified eight benefits expected by SECO onboarding stakeholders. Our quantitative validation of three of these expected benefits shows that participating in onboarding correlates with (amongst others):

1. 65% more gender diversity (both female and non-binary);
2. a median of 14% less buggy code contributions;
3. a median increase of 61% in the average code churn;
4. a median 45% (35%) shorter time to get code contributions accepted for female (other) contributors;
5. a 35% (10%/4.5%) longer average retention rate for female (male/non-binary) contributors in the SECO;

6. a median 13.5% higher pull request acceptance rate.

3.2 Background and related work

3.2.1 The SECO onboarding process

Given that SECOs constitute a complex set of inter-dependent project/cross-project teams working together for a common goal [102], a SECO's onboarding program is a "continuous" process that usually has two phases [110]: (i) top-level training, and (ii) (more traditional) project-specific training [52–54].

First, the top-level training introduces new contributors to the SECO's overall complexities, such as its organization, overall workflow, SECO-wide tools, processes, etc. Such activities also provide networking opportunities between newcomers and mentors across the SECO's sub-projects. Then, newcomers move to (sub-)project-specific training, under the guidance of a personal mentor, to learn the ins and outs of a specific sub-project in the SECO. The expected outcome of the overall SECO onboarding process is that new contributors can make their first accepted contribution.

For example, the OpenStack SECO has a dedicated OpenStack Upstream Institute (OUI) [111] responsible for organizing its onboarding process. OUI is necessary since OpenStack ranks among the largest open-source collaborative communities globally with a codebase size of over 20M LOC and produces a new SECO release every six months [79]. Due to its vast diversity in projects (with over 2,000 project/sub-project, technical standards, and social norms), new contributors may experience difficulties understanding the roadmap of OpenStack, which can significantly slow down contributions to the codebase.

The OUI organizes the OpenStack onboarding process in two phases — a two-day

physical top-level training event, followed by several months of one-to-one online mentoring. The physical event serves to share knowledge on the cross-project processes (planning and dependencies) and tools such as ZUUL (for CI/CD) and Storyboard (for issues tracking) designed to coordinate SECO-level activities. Likewise, the online mentoring phase focuses on processes and tools specific to sub-projects, as well as each project's own work culture. Since OpenStack SECO is distributed across different geo-locations, the OUI has to balance the in-person top-level training event's location and time to be equally accessible across new contributors.

3.3 Observational study of the top-level SECO onboarding phase

3.3.1 Study design

To understand how a regular, top-level onboarding training is organized in a SECO, we conducted an observational study of 72 new contributors at an OpenStack onboarding event held in Berlin, Germany, on November 11-12, 2018.

In particular, we aim at investigating the following preliminary research questions:

- **PRQ1:** ***What (and how) are the topics taught during a SECO onboarding event*****?**
- **PQR2:** ***What are the challenges involved with organizing and sustaining a SECO onboarding program*****?**
- **PRQ3:** ***What are the benefits of a SECO onboarding program*****?**

We describe this observational study's design and present the results of the PRQ_1 below. Meanwhile, we will discuss the results of PRQ_2 and PRQ_3 in section 3.4.

Participant selection. Participants for our observational study consist of the pre-registered individuals who completed the two-days onboarding event in Berlin. All participants signed a non-disclosure agreement (consent form) with OpenStack, willfully granting OpenStack the permission to record and document all activities during the entire onboarding event.

These participants command good programming skills in at least Python, have formal college/university education in Computer Science or a related field, and no prior experience with OpenStack or similar SECO. Their average age was 25±5 years, and they exhibited a high demographic diversity in terms of continents and gender (male, female, and non-binary). Participants were asked questions such as: "How do you self-identify your gender?" Participants had an option not to answer if they choose not to disclosed their gender. All 72 participants responded. While those who identified as male or female were explicitly categorized in those respective categories, the non-binary category does not only contain participants who self-identified outside male/female categories, but also people who said they are confused about what it is to be truly 'male' or 'female,' or participants declaring that they should be free to identify their gender at any given moment and how they feel about it at that particular instance. Finally, 31.94% used this field to report being gay, bi-sexual, lesbian, etc.

We obtained this confidential demographic information data either from the participants themselves before the observation study started or from the OpenStack D&I working group's private records of contributors' identities [112], to which we obtained access.

Study procedure.

The observational study involved 72 participants (P1, . . . , P72) and 13 mentors (M1, . . . , M12), including the observer (OB1; first author).

At any given instance, each of the 12 tables has six participants and a mentor, with at least one mentor leading a task or an exercise. Participants are encouraged to choose their seats and team members freely. Besides the high-quality audio-visual equipment that OpenStack provided, OB1 also used field notes to document mentors' and participants' activities during the entire onboarding event.

To understand the participants' various activities, OB1 used an observational approach with a low degree of interaction with participants but a high Hawthorne effect [113]: all the 72 participants were aware that they were under observation. Moreover, as mentors assign new tasks to participants, OB1 would randomly ask a participant to describe the actions taken during the task using the think-aloud protocol on 24 (2x12) randomly selected participants of the 12 tables.

Qualitative data analysis The first author of this thesis initially transcribed audio-visual recordings and field notes of all the 72 participants. The first and second authors in this chapter of the thesis used a combination of inductive and deductive coding at sentence/paragraph level [114–117] to analyze the transcribed text to find patterns and themes relevant to the three PRQs. These themes are further grouped/regrouped to form a hierarchical structure known as an affinity diagram [118], which enables us to visualize how concepts of high-level themes are emerging from basic low-level codes/labels.

Inductive coding With no pre-conceived themes/patterns, the first and second authors of this chapter independently apply inductive coding on 15% of the transcriptions in the first iteration to create an initial coding scheme. At the end of this iteration, the coders had 66 and 200 codes, respectively. After several discussions and three more iterations of coding, more informative codes emerged, and we merged low-level codes. Both authors agree on a set of 128 codes and a three-level hierarchical structure of code categories.

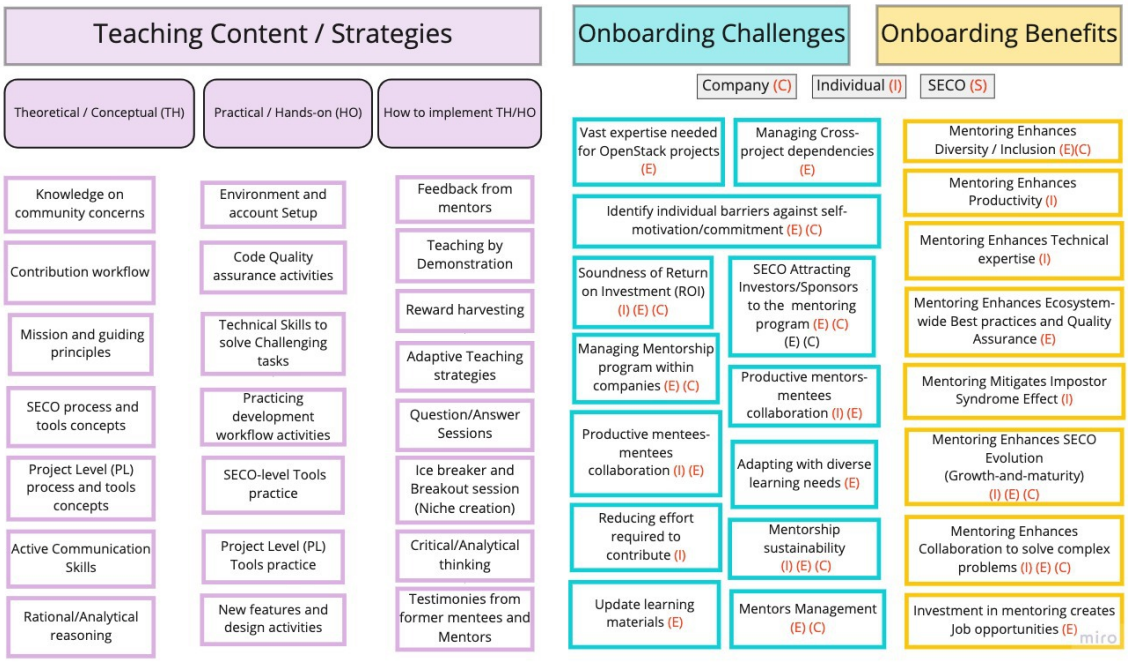
Deductive coding In this step, two coders independently apply the existing codes (from the inductive coding step) on the entire transcribed text to identify code examples. Then, calculate the inter-rater reliability (IRR) score using the Cohen kappa coefficient [119]. We perform three iterations of deductive coding and achieved IIR scores of 51%, 62.6%, and a final score of 100%. These iterations involved merging five existing codes, renaming or moving codes to fit different categories, and splitting up some code categories. The result of our coding is available online [120], and we present the final abstraction of high-level codes in the affinity diagram in Fig. 3.1.

3.3.2 PRQ1: *What (and how) are the topics taught during a SECO onboarding event*?

We grouped the Teaching Contents (TC) under THeoretical material (TH), Hands-On content (HO), and the strategies used to implement both the TH and HO, see (Fig. 3.1). Our online repository [120] contains a detailed set of activities and tasks that participants performed. Using the observational study's transcripts and notes, we could also determine the relative weight of the three groups of TC based on the allocation of time and resources to their corresponding content.

Mentors dedicate 40% of the training materials to TH, which aims to establish a solid foundation for understanding the OpenStack community and the major concepts involved in making open-source contributions to the SECO. Examples of TH contents are `knowledge on community concerns`, `mission and guiding principles`, and `contribution workflow`, but also more personal skills like `active communication skills` (why it is crucial to develop this skill, and later on, practice on these skills) and `rational/analytical reasoning`; participants are encouraged through puzzles (training archives) [111] to develop critical thinking abilities [121].

Figure 3.1 Materials taught during onboarding and their observed impacts on individual mentees (I), the SECO (E), and companies in the SECO (C).



An essential part of this training material focuses on the specifics and differences of SECO-level (`SECO process and tools concepts`) and project-level (`project level (PL) process and tools concepts`) communities and workflows. For example, the need to synchronize each project's release cycle with that of the SECO, stimulate cross-project collaboration, and deal with different workflows and tools (e.g., Storyboard issue tracker at SECO-level vs. Launchpad in several individual projects). Participants reacted to the **TH** differently: *"I am now getting more confident with my understanding of Zuul and rechecks, especially when M6 explained the concepts a few minutes ago; that was a great explanation!"*(**P51**). Yet, another participant appreciates the mentors' efforts: *"I think a load of materials has been too overwhelming, but the mentors make it look too easy for me to follow the concepts."*(**P29**)

Mentors dedicate 60% of the training materials to HO, which involves hands-on

training (50%) and deep-dives into challenging (hackathon) tasks (10%).

The HO component provides participants with a walk-through of typical real-world scenarios and tasks that OpenStack contributors face regularly. The HO component starts with the necessary steps of creating accounts with the OpenStack foundation, Gerrit (code review tool), storyboard (issue tracker), mailing-list, and IRC channels (for communication). Mentors also guide participants to install and configure their (virtual) working environment, which comprises a VirtualBox with possibly a Ubuntu image pre-installed, a copy of the OpenStack development environment (aka DevStack on Sandbox), issue trackers such as Launchpad and Storyboard, the code review environment (Gerrit), and git. Moreover, the OpenStack Sandbox environment (repository) provides virtual servers for testing OpenStack projects/functionalities in an isolated environment. Also, mentors ask participants to perform tasks of varying difficulties covering technical areas. Such as documentation, implementing new features, tracking issues (using storyboard/Launchpad), reviewing source code, best practices on commit messages and code quality, and CI/CD. **OB1** asked a participant to think aloud while performing a HO task: *" I want to run several unit test cases and an integration test. I use the 'tox framework' to run unit testing, so I call the 'tox' command on my terminal [typing . . .] "*(**P7**)

Mentors use a variety of teaching strategies that facilitate collaboration and competitiveness among participants throughout the training sessions (Fig. 3.1). These strategies enhance participants' understanding of the teaching content by making the sessions interactive. The most observed strategies include the following:

`Ice-breaker and breakout session`. Training sessions begin with an introductory activity by both mentors and participants to create an atmosphere of familiarity that facilitates collaboration among participants (novice-novice collaboration) and mentors (novice-expert

collaboration). Breakout sessions during the event further strengthen this collaboration.

Expert-novice feedback. Mentors usually use this strategy to teach practical skills that require a "trial-and-error" approach. Therefore, they allow participants to make several attempts, while the mentors keep providing constructive feedback until the participants arrive at the answer.

Teaching by demonstration. Mentors often demonstrate how things work while explaining the underlying concepts; this approach enriches participants with confidence towards the mentors and the ecosystem.

Reward harvesting. Mentors use reward strategies to motivate participants to be competitive and work in a group while completing challenging exercises within an allocated time frame. The first participant to figure out the best solution to a task within that time-frame is **rewarded** with a token, a swag, or a sticker. This strategy required participants to apply critical and analytical thinking.

Novice-novice collaboration. Mentors encourage participants to work in small groups of two people at each table and discuss their problems/solutions table-wise.

Participants were mostly positive regarding the strategies, which mentors used. **P48** said: *"I like the hands-on section most and, of course, the sticker prizes."*, besides, other participants appreciated different strategies differently: *"The testimony on mentoring was great! I love it."*(**P15**) Meanwhile, **P31** congratulates the strategy and know-how of the mentors: *"Mentors were great inspirations and knew their stuff well."* Also, mentors use real-life scenarios to explain difficult concepts: *"I admired the explanations of different projects and how they form an ecosystem."*(**P1**)

3.4 Perceived challenges and benefits of SECO onboarding

Based on the observed onboarding activities shown in Figure 3.1, 13 challenges and 8 benefits emerged. During our observation, **3/13** challenges and **5/8** benefits encountered substantially more and deeper discussions than others, leading to significantly more words in the transcriptions of the audio-visual recordings. Below, we discuss in detail these three challenges (PRQ_2) and five benefits (PRQ_3).

3.4.1 PQR2: *What are the challenges involved with organizing and sustaining a SECO onboarding program*?

Challenge 1: `Vast expertise needed for SECOs`

Onboarding at the SECO-level has several challenges beyond the project-level onboarding. In particular, since a SECO is not just the union of hundreds of smaller projects but involves the collaboration of hundreds of cross-project teams with diverse interacting technologies (see the `cross-project dependencies` challenge). Given that the onboarding participants do not know the different SECO projects, the initial onboarding event cannot make any assumptions. It should target the overall SECO contribution process. To cover a wide variety of topics and tools (see PRQ_1), this also implies that mentors should have polyvalent skill-sets to guide the participants: *"Be prepared with the 'deep dives' exercise. Usually, participants have very different levels of knowledge and skill-set,"*(**M2**) (which in turn impacts `mentorship sustainability`. Furthermore, there should be ongoing communication between the SECO-level onboarding process and the onboarding process within individual projects of the SECO (see `mentorship within companies`), for example, to `update learning materials` to project-level developments.

Challenge 2: `Self-motivation and commitment`

It is challenging for SECO to identify individual barriers against self-motivation/commitment. Therefore, active participation in an onboarding experience is tantamount to a successful outcome, hence every stakeholder should be fully involved and committed. *"Successful mentoring requires active commitment both from the mentor and the mentee."*(**M9**), also, another mentor advocates *"People learn in different ways at different speeds, which means a commitment to active mentoring requires more than a handful of quick IRC chats."* (**M7**) This challenge has direct links to the `adapting with diverse learning needs` challenge.

Challenge 3: `Mentorship sustainability`

SECOs and companies face challenges finding available mentors to guide mentees. This is partly because of challenge 1 above, and partly because mentoring requires substantial effort to prepare and keep material up-to-date. Constrained companies may prefer to prioritize their experts' time on tasks that will bring more financial profit to the company, at the detriment of supporting mentees. At the observed onboarding event, participants were briefed that *"If there aren't enough mentors on every table, ... float around the room checking in on people, especially during exercises."* **(M1)**

3.4.2 PRQ3: *What are the benefits of a SECO onboarding program*?

Benefit 1: `Mentoring positively correlates with diversity`.

Gender diversity (GD): out of the 72 participants at the observed onboarding event, 17 (23.6%) declared themselves as female, 23 (31.94%) as non-binary, and 32 (44.44%) as male. Moreover, for **corporate diversity (CD)**, we found evidence of different companies involved with OpenStack and sponsoring events, and hiring both Cat-2 and Cat-3 contributors.

We also observed that mentors and participants had diverse technical skill-sets that cut across different project/cross-project teams. Such, **technical diversity (TD)** brings value to the SECO since it *"drives cross-project teams forward through more mixed reviews, contributions, and viewpoints. By expanding that diversity, we're able to develop a variety of opinions for the open infrastructure project as a whole, ultimately"*. (**M9**)

Benefit 2: `Mentoring positively correlates with productivity`.

During onboarding, mentors assign real-life exercises and tasks to participants, such as creating patch sets, fixing bugs, testing and CI/CD (Zuul), and submitting new features and documentation. All 72 participants actively participated in the coding activities and successfully submitted acceptable commits. This not only trains the participants in the field, but also encourages them to adopt a collaborative workflow (often by themselves), both with other participants (novice-novice) and with mentors (experts-novice). OB1 observed how *"mentors were pairing participants to work on exercises, i.e., P33 and P35 seated on table/group 10, were exchanging ideas constantly throughout this exercise."* Moreover, M11 asked participants to: *"run different test cases in each project that you cloned. 'If you need help, mentors are seated on your tables, and they will assist you in running the test cases.'"*

Benefit 3: `Mentoring positively correlates with SECO QA / best practices`.

Mentors presented various techniques and best practices related to quality assurance (e.g., test-driven development, CI/CD, code reviews) and asked participants to practice those skills. Also, mentors presented a couple of bad and good examples of code that respect OpenStack standards. Some of these best practices include writing good commit messages and proper code documentation. **M9** *"shows a couple of bad examples of commits that reviewers rejected because they violated the best practices, which OpenStack enforces."*

Benefit 4: Overcoming imposter syndrome effect.

New contributors to an ecosystem often feel overwhelmed and inadequate, preventing them from collaborating freely with the other contributors in the ecosystem perceived as having more talent [122]. Thus, it is important for SECOs to take measures to ensure that they help participants to identify and start fighting/eliminating the imposter syndrome [123, 124]. *" As a new developer fresh out of college, coming into any new team can be very intimidating. Everyone around you knows so much more than you, and you feel that you're an imposter with so much to learn ... "*(**P1**). The onboarding program is aware of the effects of the imposter syndrome and sensitizes participants to overcome those, especially by letting mentors and past mentees share their experiences.

Benefit 5: Evolution of Ecosystem

As mentors transfer skills to mentees, they produce a larger pool of talent and enable the perpetual growth of the SECO (growth-and-maturity). In turn, previous mentees return to the onboarding program as mentors to help encourage participants to grow within the SECO: *"M7 mentored me during my last year of college, and I have been very fortunate to work with *them* and continue being *their* mentee. ... mentoring helps manage immature skill sets required to grow into a senior engineering role in the future."*(**M3**).

3.5 Quantitative validation of perceived benefits

In this section, we empirically evaluate the extent to which onboarding can achieve the three major perceived benefits identified in PRQ3. We could quantify and measure these three benefits by studying 84 months of historical contributions (code changes, issue reports, and code reviews) in the OpenStack SECO. Indeed, we measure Diversity, Productivity, and Quality. Specifically, we investigate these three research questions:

– **RQ1:** ***Does onboarding correlate with SECO diversity*****?**

– **RQ2:** ***Does onboarding correlate with new contributors' productivity*****?**

– **RQ3:** ***Does onboarding correlate with new contributors' code quality*****?**

3.5.1 Study design

Categorization of contributors. OpenStack's onboarding program is publicly advertised, with free training events (travel support is available) taking place in different countries. Hence, anyone is encouraged to do onboarding, not just people who could afford the travel expenses. Therefore, to measure the impact of onboarding on the OpenStack SECO, we considered three categories of contributors in our study (see Fig. 3.2). The first category (Cat-1) constitutes contributors who joined OpenStack before onboarding events were introduced and could not benefit from any official onboarding. The second category (Cat-2) comprises new contributors who did not participate in any onboarding event, even though the onboarding program did exist when joining OpenStack. Finally, the third category (Cat-3) are contributors who participated in the onboarding program. Both Cat-2/3 have equal opportunity/access to the information concerning the onboarding training program. Cat-2 contributors are similarly self-motivated to contribute to OpenStack as Cat-3 participants but choose not to participate in any OUI program.

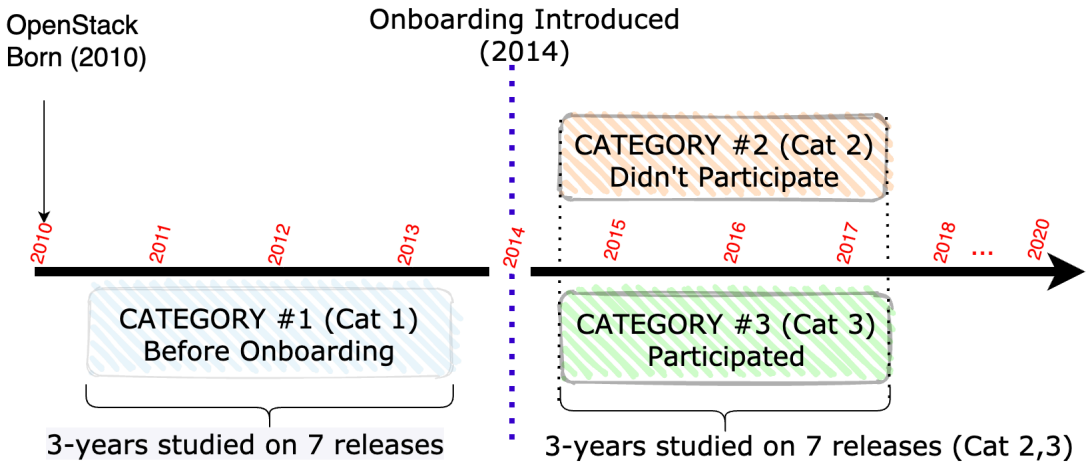
Each of the three categories plays an essential role in our study. In particular, for each RQ and metric, we first compare the distribution of the metric values between Cat-2 and Cat-3. If significant differences exist, we perform a second comparison between Cat-1 and Cat-2 to control for any confounding factors such as changes in the development process that were put in place simultaneously when OpenStack introduced the onboarding program. Only if no significant changes exist between Cat-1 and Cat-2 (both of whom consist of

contributors who did not do onboarding) can we correlate the differences between Cat-2 and Cat-3 with the introduction of onboarding.

Data collection. Given the three categories of contributors (Cat-1, Cat-2, and Cat-3), we first use the clustered random sampling technique [125] to randomly select Cat-3 first-time contributors who joined through the OUI onboarding program from different geographic areas, reflecting the distributed nature of SECOs in our sample. This yielded 427 Cat-3 participants across seven OpenStack releases, from Juno to Pike (Fig. 3.5). We make sure that only new contributors (e.g., from college or similar training program) without any prior history of contribution to OpenStack codebase are selected in Cat-3. Then, we used stratified random sampling to select an equal number (427 each) of individuals in Cat-1 and Cat-2. For those two categories, we made sure to exclude any OpenStack contributor who later on (after making contributions) decided to participate in onboarding (720 exclusions). In this thesis design, we use Cat-1 as our control group, similar to Cat-2, whose participants, did not do any onboarding training (since there was no onboarding program at the time of Cat-1). Then we use both Cat-2 and Cat-3 as our experimental groups. Finally, we mapped the 1,281 (3x427) selected contributors across all three categories to their activities in the following OpenStack repositories: Gerrit (code review system), git, and Launchpad/Storyboard (issues trackers). Based on this integrated information, we extract contributors' activities related to commits/patch-sets, bugs reported, reviews, blueprints, declared gender, and affiliation for each category's studied period. All experimental data and relevant materials are hosted online [120] for replication or third-party reuse.

Metrics and statistical tests. We adapt existing metrics from the CHAOSS project [126], Meyer et al. [127] and Jansen [128] (see Table 3.1) to measure the extent to which expected benefits of onboarding are achieved at OpenStack.

Figure 3.2 Timeline of stratified categories used in our study. Cat-1 is our control group, while Cat-2 and Cat-3 are the experimental groups. Each group uses data of seven OpenStack releases (42 months).



Our study analyzes these metrics at the individual contributors' level, then aggregates them to the SECO-level, split across the three categories of contributors. Some metrics are general, while others (like Technical and Corporate diversity) are SECO-specific. Note that there is only a weak Pearson correlation of 0.324 between Effort and TFC, i.e., they measure different phenomena.

We then analyze and compare contributor activities among the three categories using several statistical tests. We use survival analysis [129] to measure the amount of time it takes for an event, such as making the first commit, to occur. A (non-parametric) log-rank test is further used to compare the survival curves of multiple groups. If $\rho < \alpha(0.001)$, the tested survival curves are non-overlapping.

For other metrics, we use the Kruskal-Wallis H-test (KW-H) [130] to compare metric distributions of the three contributor categories at once. In case of a statistically significant difference ($\rho < \alpha(0.01)$), a Dunn (posthoc) test [131] is used to identify which of the three categories has a different distribution of metric values. As such, Dunn evaluates Cat-1 vs. Cat-2, Cat-1 vs. Cat-3, and Cat-2 vs. Cat-3. Finally, we measure the effect size (Cliff's

delta) [132], which quantifies the effect of significant differences. As explained earlier, we expect that if onboarding correlates with a change in, say, a productivity metric, then Cat-1 (the control group) and Cat-2 (treatment group) should have no statistically significant difference. In contrast, there should be a statistically significant difference between Cat-3 and Cat-2 (and, hence, Cat-1).

Table 3.1: Contributor-level∗, SECO-level†, and/or company-level♯ metrics used in our study.

RQs	Metrics	Description
RQ1: Diversity	Gender$^{\propto}$ (GD)†	Proportion of new contributors who self-declare as Male (m), Female (f) or non-binary (n) [112].
	Technical (TD)∗	The number of different project teams (technology) new contributors are involved in [133].
	Corporate (CD) ♯	The number of sponsoring companies that contribute commits to the SECO [59] [126].
RQ2: Productivity	Density (Den)∗	Commit density, i.e., the median proportion of contributed churn over the submitted commits [134].
	Time to first commit (TFC)∗	Number of days it takes for contributors to have their first commit accepted and merged into the codebase. [126]
	Retention (Rt)∗	The proportion of contributors, per category, still contributing to the codebase after N days [52] [126].
	Patch Acceptance Rate (PAR)∗	Probability of a contributor's contribution (‡pull-request; PR) to be accepted (higher values are better) [126]: $$PAR = \frac{\#Accepted_PRs}{\#Submitted_PRs} \quad (3.1)$$
RQ3: Quality	Effort (Eft)∗	A measure of the number of ‡pull request versions (attempts) necessary before a contribution is accepted (lower values are better; minimum value of 1) [126]: $$Eft = \frac{Median_\#Attempts}{\#Actual_Commits} \quad (3.2)$$
	Bug-Inducing-commits (SZZ)∗,♯	Percentage of submitted commits that introduce bugs [135].

‡Pull-request (GitHub) or change-request (Gerrit)

$^{\propto}$To measure Gender diversity at OpenStack, contributors' identity is not publicly available for confidentiality purposes.

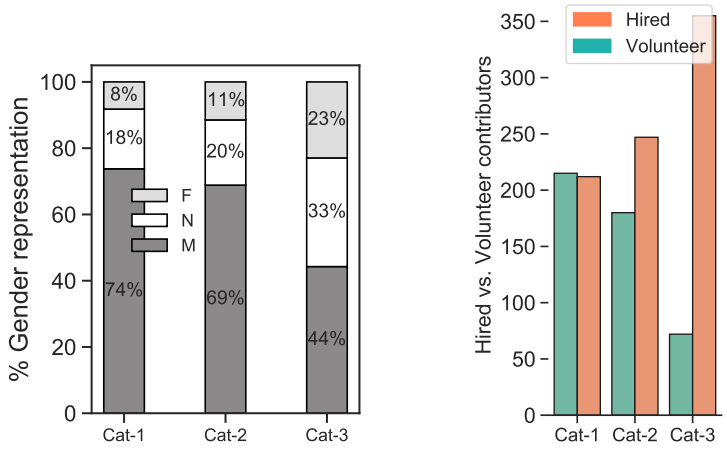
3.5.2 RQ1: *Does onboarding correlate with SECO diversity*?

This RQ aims to understand the correlation between onboarding and (i) gender representation (gender diversity) within the OpenStack SECO, (ii) the distinct skill sets of contributors (technical diversity), and (iii) the degree to which different corporations/organizations contribute code or sponsor events (corporate diversity).

Gender diversity (GD)

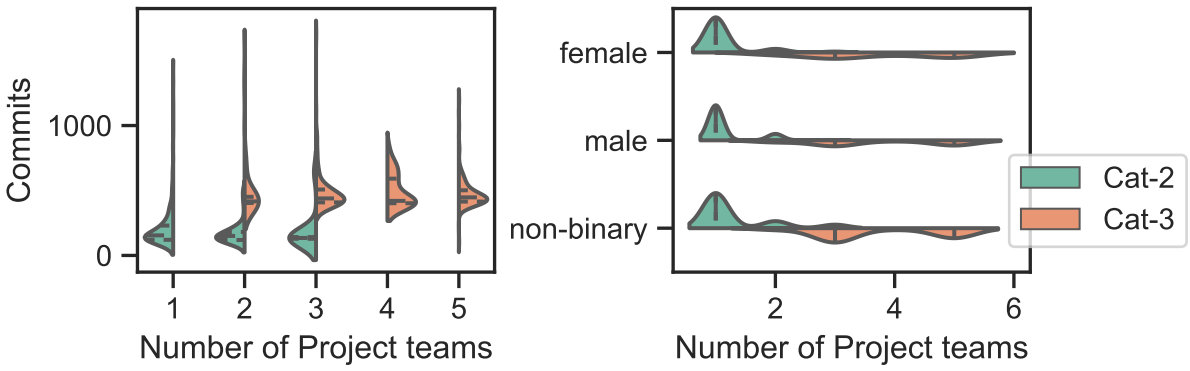
We observed a statistically significant increase of 65%, with large (L) effect sizes, in terms of contributors declared as either female or non-binary within Cat-3 (compared with Cat-2), at the expense of contributors who reported male gender [112]. Fig. 3.3(a) shows how the percentage of contributors who declared themselves as female increased to 33% compared to the 18% (20%) values for Cat-1 (Cat-2). Similarly, for contributors who declared themselves non-binary, the percentage significantly increased from 7% (10%) to 23%. The main reason for these increases seems to be the fact that a significantly smaller proportion of contributors explicitly declared themselves as having male gender, which thus far has been the over-represented gender in open source development [136]. There are different interpretations possible. The most likely explanation, supported by the fact that we did not find a significant difference in gender between Cat-1 and Cat-2, is that onboarding helped to attract a larger proportion of contributors of female gender, while providing confidence to others to declare themselves as non-binary instead of sticking to a binary gender. Self-disclosed gender at OpenStack [112] is not open to the general public; it is available in internal profiles for confidentiality purposes. However, there could still be confounding factors. For example, maybe contributors with male gender are less likely to participate in onboarding events. More research is needed to better understand this.

Figure 3.3 (Left) Median GD (in %) of each category ((F)emale, (M)ale, and (N)on-binary) ; (Right) Hired vs. volunteering (Vols) contributors in Cat-1, Cat-2, and Cat-3.



(a) Gender Diversity (GD) (b) Hired vs. Vols

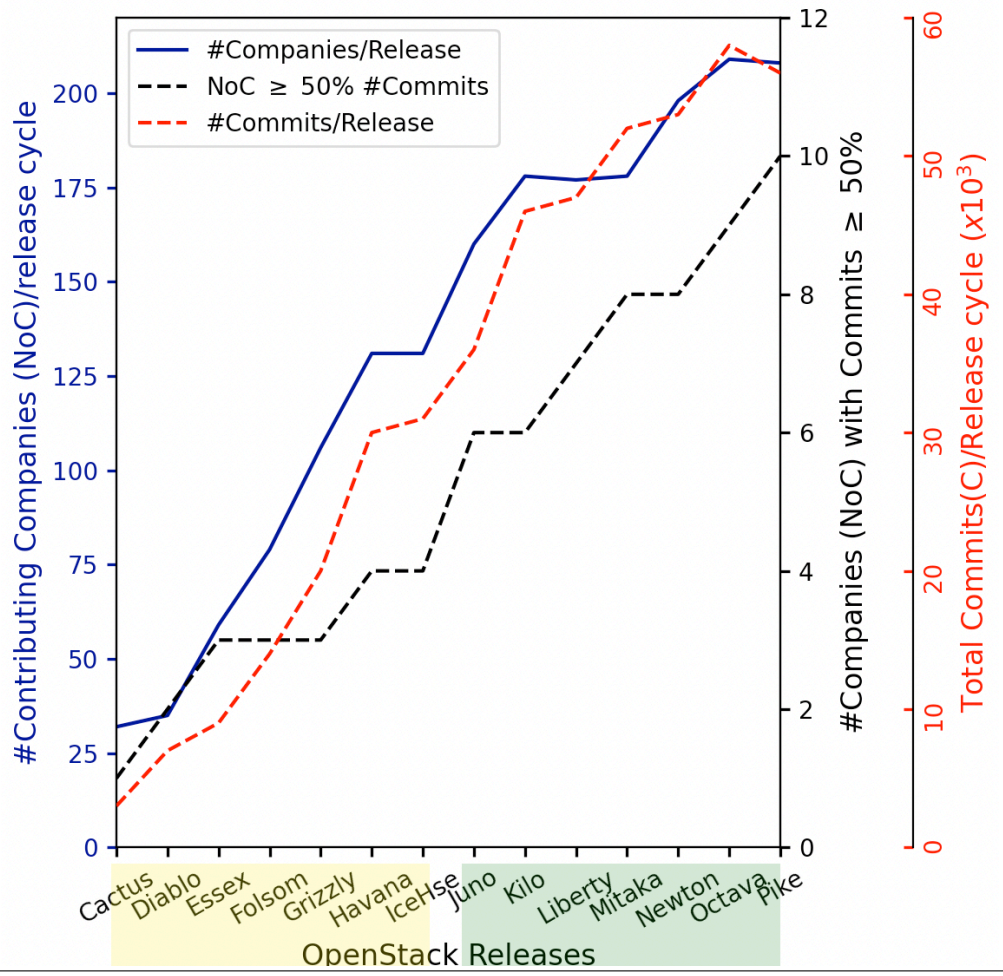
Figure 3.4 Overview of technical diversity, showing the number of commits made across different numbers of projects (Left) and the number of projects people contribute to, per declared gender (Right).



Technical diversity (TD)

People who followed onboarding (irrespective of gender) are more polyvalent than other contributors. Technical/code diversity measures the number of distinct projects (modules) to which a developer contributes source code. Fig. 3.4 shows that people who joined OpenStack without onboarding (Cat-1, not shown, & Cat-2) contribute to at most

Figure 3.5 The evolution of the number of companies (NoC, solid blue line) for each of the 7 studied OpenStack releases before (Cat-1, yellow) and after (Cat-2/3, green) the introduction of onboarding. The black dashed line represents the top NoC responsible for 50% of a release's commits and the red dashed lines shows the total commits per release cycle.



three projects, whereas people who joined through onboarding (Cat-3) often are contributors in more than three projects. For example, in Cat-2, 82.7% of individuals contribute to only one project, 16.6% contribute to two, and only 0.7% contribute to three projects; only contributors with non-binary or male gender contributed to two or more projects in Cat-2. On the other hand, in Cat-3, 52.7% contribute to three core projects, 31.9% contribute to

four projects, and 15.5% to five or more projects; contributors who declared male or non-binary gender are mostly contributing to three and four projects, while contributors who declared female gender are even contributing to five or more projects (significant difference between female and other genders). This supports our earlier findings about gender diversity (Section 3.5.2).

Furthermore, we find a statistically significant difference (large effect size) between Cat-2 and Cat-3 in terms of TD, and the number of commits made by Cat-3 contributors is significantly higher than those by Cat-2 contributors (median of 150 compared to 375).

Corporate diversity (CD)

refers to the way in which the Cat-2 and Cat-3 contributors who contribute code to a SECO are distributed across companies. It also measures if a particular company has a monopoly of over 50% or more of these contributions, which could influence the work culture of the SECO or, in the worst case (departure of key contributors), could cripple the SECO [137–140].

Studies [59, 108] show that companies contributing to the OpenStack codebase have an uneven distribution of commit across those companies. **Also, we found that none of the sponsoring companies (NoC) had a disproportionate amount of contribution either by Cat-2 or Cat-3 contributors. Furthermore, 83% of Cat-3 contributors were hired by companies compared to 51% of Cat-2 contributors**, and this difference is statistically significant with a ρ-value of $3.006x10^{-40}$ and a large (L) effect size. We also observed that no single company has a dominating share of contributors (and contributions).

Furthermore, Fig. 3.3(b) shows how the number of new contributors that remained volunteers instead of being hired dropped substantially from 48% in Cat-2 to 17% in Cat-3.

In other words, onboarding seems to be correlated with higher chances of being hired by OpenStack companies.

Only 13% of Cat-3 contributors were hired by the companies that sponsor the onboarding events 70% of the 83% hired Cat-3 contributors were employed by companies within OpenStack that do not sponsor onboarding (median days to hire for Cat-3 is 33.0 vs. 212.0 for Cat-2). While, overall, the high percentage of 83% is positive for the ecosystem as a whole, the sizeable proportion of contributors hired by non-sponsoring companies could be interpreted as a form of "brain drain" and "low return of interest" for the companies organizing the onboarding training.

On a positive note, though, we observe that seven of the top[2] 10 Cat-3 contributors in the SECO were hired by sponsoring companies, which improves their onboarding ROI. On the other hand, Cat-3's hired contributors switch more easily from one company to another. This could indicate that the expertise of Cat-3 contributors is useful and sought-after in different contexts, or could be due to technology transfer between ecosystem companies. For example, one given contributor started contributing to the Nova project in the Pike release cycle with *IBM*, switched to *Huawei* and later to *Futurewei*, all between February 20th – August 1st (2017).

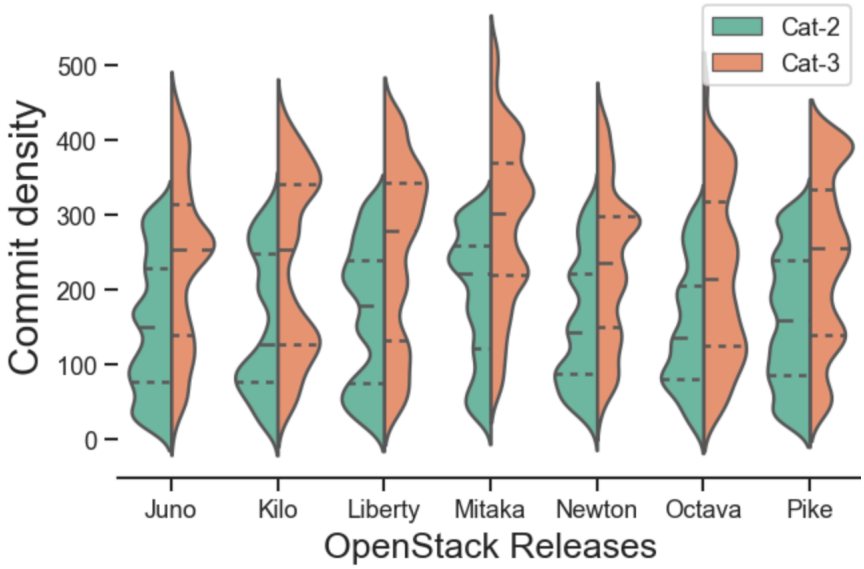
3.5.3 RQ2: *Does onboarding correlate with new contributors' productivity*?

Commit density (Den)

Onboarding positively correlates with contributor productivity. Fig. 3.6 shows a 61% increase in the median density of Cat-3 contributions compared to Cat-2 contributions,

[2]We used rankdata [141] on *TFC*, *SZZ*, and *Effort* to rank and sort the vectors of contributors in ascending order according to each of these three metrics separately. Since a contributor can be better in one metric but worse in the other, rankdata then aggregates the scores to identify the top 10 contributors.

Figure 3.6 Comparison of commit density between Cat-2 and Cat-3.



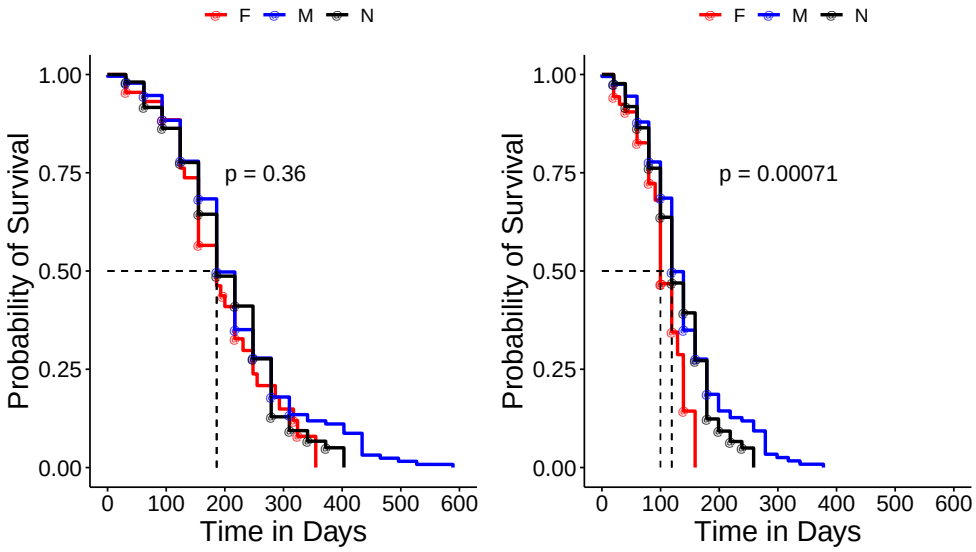
which is a statistically significant difference with large effect size (while no difference was observed between Cat-1 and Cat-2 contributions). This indicates that people who did onboarding consistently produce a higher average churn across their contributions.

Time to first commit (TFC)

Onboarding correlates with a median 45% lower time to first commit for female and 35% for both male and non-binary contributors.

Fig. 3.7 shows the survival curves [142] (with ρ-values obtained using the log-rank test) for the time until first commit (in number of days) for the three categories of contributors, split across the three genders. Only for Cat-3, we obtained statistically significant results among the genders. Furthermore, we obtained a significant difference with large effect size between Cat-2 against Cat-3, across all three genders. It takes 100 (120) days for half of the female (male/non-binary) contributors in Cat-3 to make their first commits, while in Cat-2, it takes at least 185 days for any contributor to get their first commit accepted.

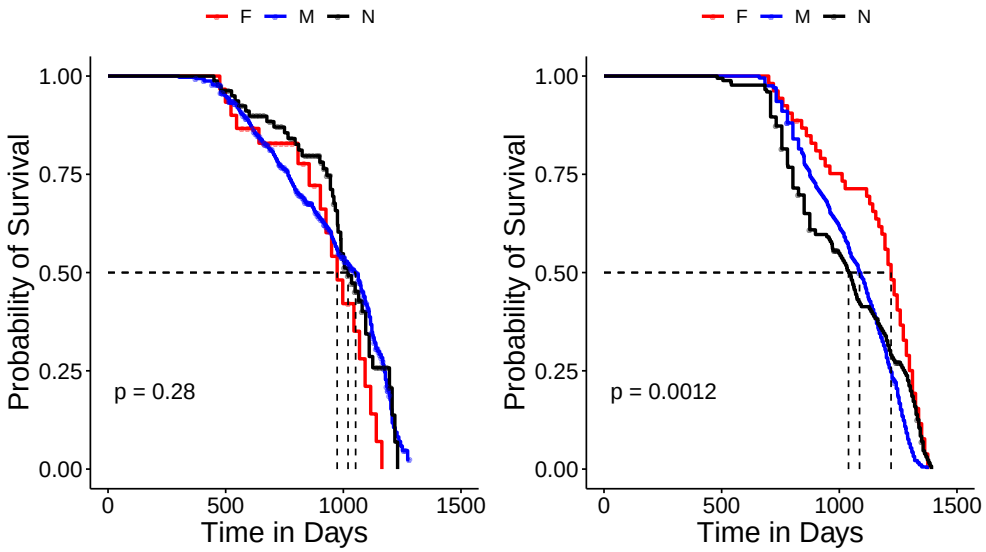
Figure 3.7 Survival curves for time until the first accepted contribution per gender in Cat-2 (left) and Cat-3 (right).



Retention rate

Onboarding correlates with a 16% longer average retention rate across the three genders in the SECO, i.e., Cat-3 contributors are active much longer than Cat-2 (and Cat-1) contributors, which is beneficial for the sustainability and cohesion within a community. We observed from the survival analysis chart (Fig. 3.8) that 95% of contributors were active for 450 days in Cat-2 and 750 days (four SECO release cycles) in Cat-3. While there is a 50% probability of Cat-1 contributors (either gender) abandoning the SECO/project within 1,000 days (not shown), this retention period is 1,100 (1,000) days for Cat-3 (Cat-2) non-binary contributors, 1,150 (1,100) days for males, and 1,290 (950) days for females. Therefore, contributors, on average, were productive for a significantly longer time in Cat-3 than in both Cat-1 and Cat-2 (large effect sizes), with self-declared female contributors with onboarding experience persisting longer than any other declared gender.

Figure 3.8 Survival curves for the time until Cat-2 (left) and Cat-3 (right) contributors leave.



Patch acceptance rate (PAR)

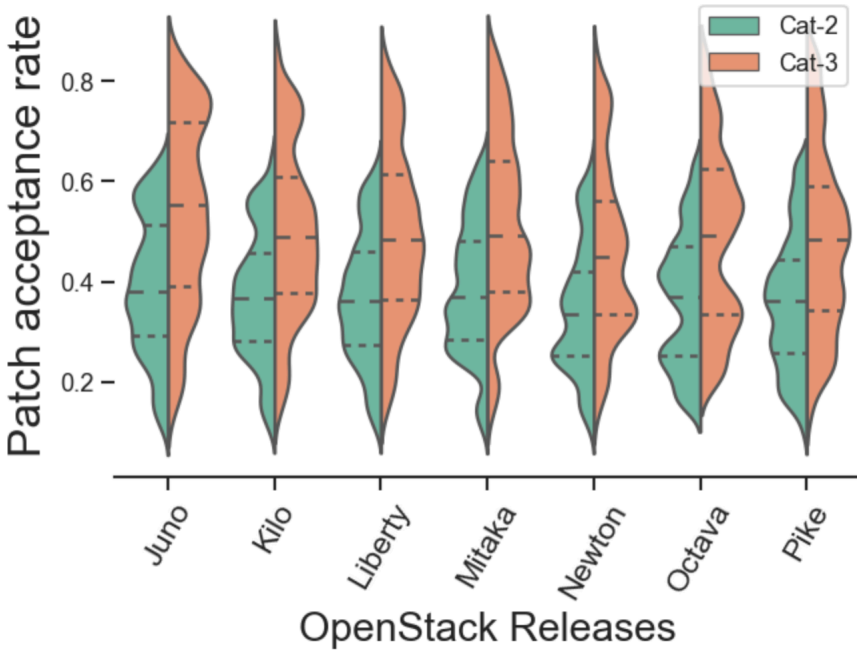
Onboarding correlates with a significant increase in the percentage of accepted pull requests (i.e., Gerrit "change requests"), i.e., contributors are more successful in getting their patches accepted. Fig. 3.9 (top) shows that the median PAR for Cat-3 contributors is 35.7% to 49.2% times higher compared to Cat-2 contributors. Our evidence suggests that contributors self-declared as female outperformed the other genders (not shown), in both Cat-2 and Cat-3, in terms of PAR (large effect size).

3.5.4 RQ3: *Does onboarding correlate with new contributors' code quality*?

Effort

Cat-3 contributors require less effort to have their commit accepted. Based on our observation and results in Fig. 3.10 , contributors who joined the ecosystem without an onboarding training (Cat-1 & 2) on average require significantly more attempts to get their

Figure 3.9 Comparison of patch acceptance rate between Cat-2 and Cat-3



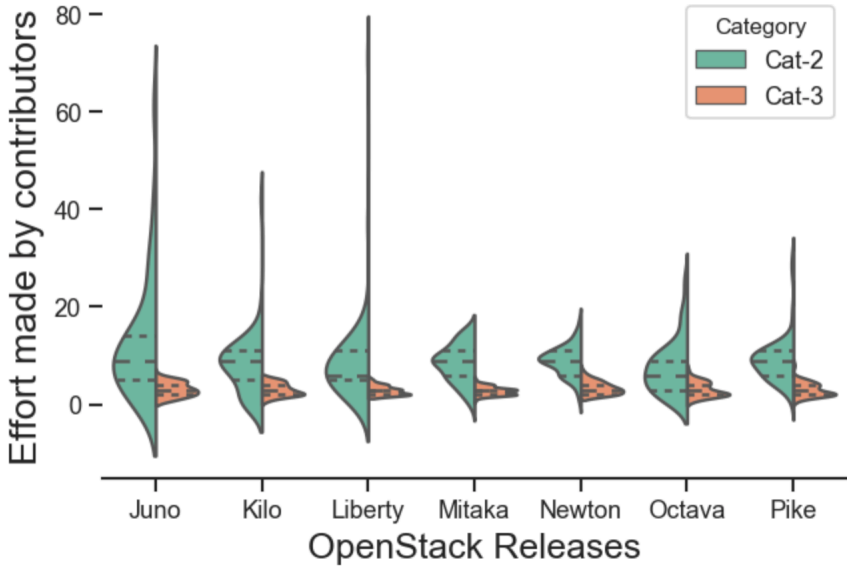
contributions accepted than those who were onboarded (Cat-3), with ρ-value of $6.621x10^{-77}$ and large effect size.

Since this observation only holds for Cat-3, this provides initial evidence for the hypothesis that onboarding enables contributors to better master the codebase, workflow and guidelines of an ecosystem. More research is needed to further validate this claim.

Bug-inducing changes

Contributors who did onboarding produce code that is 14% less likely to introduce bugs. Using the PyDriller [143] implementation of the SZZ algorithm [144–148], our results show that the median probability of a commit introducing a bug is 25% for Cat-2 compared to 14% for Cat-3 (Fig. 3.11). In other words, accepted patches are less buggy for Cat-3, even though Cat-3 contributors submit a higher quantity (with more complexity) of code changes than contributors from the other categories (as previously discussed in RQ1

Figure 3.10 An Overview of effort needed by Cat-2/3 contributors.

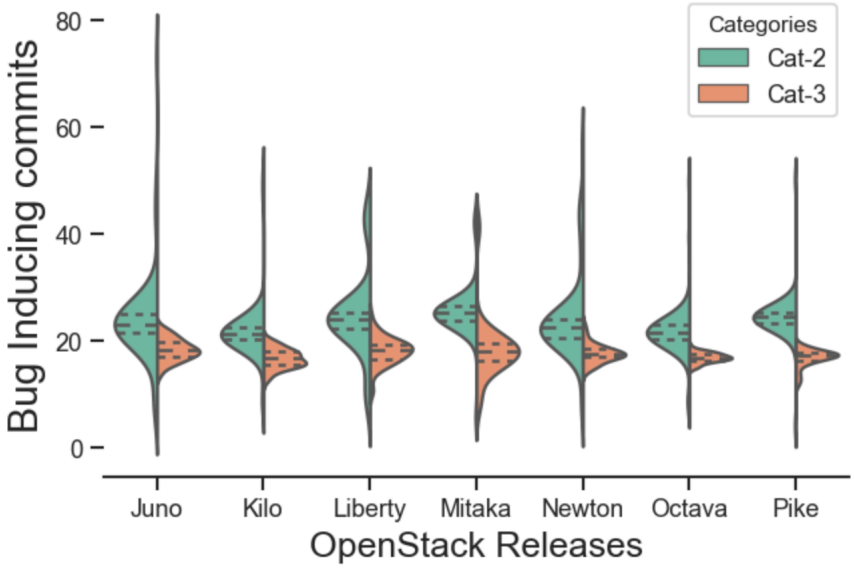


for TD). These differences are significant with a ρ-value of $4.290x10^{-57}$ and a large effect size. Not only are patches of Cat-3 contributors less buggy, they also required less attempts to be accepted (see previous metric).

3.6 Discussion

Based on the observational study findings (Fig. 3.1), we notice how the themes in the affinity diagram form a holistic set of socio-technical activities relevant to onboarding in a complex SECO. Such onboarding is more than giving a tutorial on creating a feature branch or running a test suite. Mentors spent substantial effort explaining the interactions and differences between the OpenStack SECO and the individual projects inside the SECO. `Knowledge on community concerns` is another essential pillar of the teaching content, as well as activities to train participants' `active communication skills` and `rational/analytical reasoning`. Combining such topics with the more technical hand-on activities requires (i) the use of a host of engaging teaching strategies, as well as

Figure 3.11 Likelihood of bug-inducing commits across Cat-2/3.



(ii) a continuous (online) onboarding process that goes well beyond the initial onboarding event.

While such an onboarding process requires an investment, both financially and in terms of in-kind, SECOs expect that the process can boost new recruits' productivity and the quality of their contributions and foster an inclusive and diverse community, able to sustain the SECO.

In particular, we observed that as the community grooms new contributors, they later become resourceful to the community by impacting other new contributors' growth by becoming mentors themselves. The idea is that the community evolves; mentees become mentors, and contributors stay longer within the community.

Our quantitative evaluation found evidence that some of these major expectations indeed seem to hold. The radar chart in Fig. 3.12 shows the extent to which the diversity, productivity, and quality of onboarded contributors (Cat-3) differ from contributors without onboarding (Cat-2). For each metric, the chart plots the median values for Cat-2/3 at

the contributor-, SECO- (GD) or company-level (CD, SZZ), using log-scale.

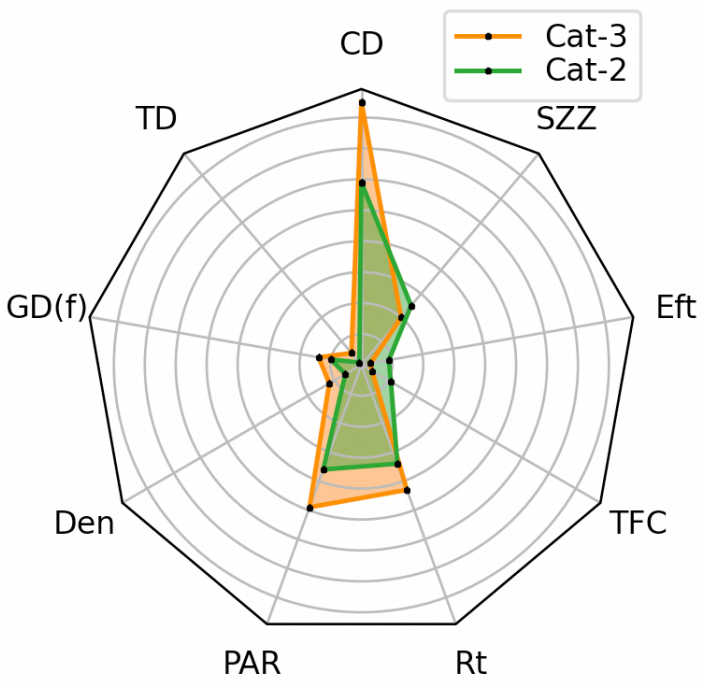
In particular, onboarding correlates with improved diversity (GD(f), TD, and CD) and productivity (TFC, Rt, PAR, and Den) metrics, since contributors in Cat-3 recorded significantly higher values in these metrics against Cat-2 contributors. However, onboarding correlates with reduced bug-inducing commits (SZZ) and efforts (Eft). Given that Cat-3 contributors seem to spend less effort in making quality code changes (commits). On the other hand, Cat-2 spent more time making their first accepted contributions (TFC) in terms of productivity; they also expend more effort, which are more likely to be bug inducing. Onboarded contributors stay longer in the SECO and make diversity more visible, but not necessarily within one SECO project or company or with a company sponsoring the onboarding process. Other potential benefits still need to be empirically evaluated.

Finally, several challenges could potentially complicate or even inhibit the onboarding process. A substantial amount of these challenges relate to people management—notably, the steady supply of motivated participants and capable mentors. While successful onboarding could yield new future mentors, both the SECO and academia should monitor this continuity carefully not to overload the same group of experts. At the same time, the latter have to keep on reinventing their teaching strategies to effectively teach the minimum material covering as much as possible the workflow and requirements of both the overarching SECO and the individual projects to be productive as fast as possible. Future research should explore and address these challenges.

3.7 Threats to validity

Construct validity. This study uses existing diversity, productivity, and quality metrics from the literature [10,59,127,149] and open source communities such as CHAOSS [126].

Figure 3.12 Radar chart of the studied metrics showing that onboarding (Cat-3) has significant differences and improvements over Cat-2. The metrics are those of Table 3.1: Bug-inducing-commits (SZZ), Effort (Eft), Time to first commit (TFC), Retention (Rt), Patch Acceptance Rate (PAR), Density (Den), Diversity: Gender (GD(f)), Technical (TD), and Corporate Diversity (CD).



However, concerning gender, we relied on the self-declared gender available in OpenStack's internal profiles [112]. Furthermore, we observed an onboarding event and mined readily available data from version control, issue reports, and code review repositories but did not have access to the private online communication between mentors and mentees after the onboarding event.

Another threat relates to the impact of the participants' awareness of our observation study on their behavior. To mitigate this, we observed selected people on a given task. We watched the onboarding event's video recording to validate how other participants performed the same activity when not directly observing them.

Internal validity. Confounding factors may have been responsible for some of the

observed differences between Cat-2 and Cat-3 contributors, i.e., factors other than the introduction of onboarding could explain some of our findings. Our study design included the Cat-1 control group, which, similar to Cat-2, consists of participants that did not do onboarding to mitigate this threat. Hence, if, for a given metric, no changes are observed between Cat-1 and Cat-2, the likelihood of confounding factors reduces (but not to zero). None of our quantitative analyses observed statistical differences between Cat-1 and Cat-2.

Another threat concerns the effect of unreported bugs on the result of the SZZ bug-inducing commit analysis, which uses an implementation of the original SZZ algorithm [144]. To mitigate this, we run SZZ on the entire history of OpenStack's official issue tracking systems (Launchpad/Storyboard). Also, our study window spans 14 releases (7 for Cat-1 and 7 for Cat-2/3), which gives ample time for contributors to make active contributions. We base our study on the assumption that participants/contributors had no prior experience with any SECO. However, since some educational institutions introduce their students to open-source development concepts and practices as part of their learning path, this could be a confounding factor that could affect our results. Since "generic" development concepts form only a minor part of the onboarding process, we believe this threat is minimal.

External validity. While OpenStack is a representative modern SECO, our results may not generalize to other ecosystems. That said, the methods that we use in our observational and quantitative studies are ecosystem-agnostic. Hence, practitioners and researchers could use our methods to identify and evaluate the impact of any ecosystem's onboarding program. As a side note, the post-Covid-19 era fosters a culture of online collaboration that could disrupt the dynamics of in-person [150] events. Even though Rodeghero et al. [151] studied onboarding during the Covid-19 pandemic at the project-level, it is still too early to

understand the impact of this on the top-level SECO training events or the SECO onboarding process as a whole. For example, the recent OUI training event on October 22-23, 2020, was virtual, yet the turnout was much lower (8 mentors and 11 participants) than previous events. Future research is necessary for the new reality of in-person vs. virtual training events in OSS communities.

Reliability validity. Except for confidential participant information, we provide the necessary description and resources (OSS tools and dataset) [120] needed to replicate our research.

3.8 Conclusion

This chapter provides the first large-scale, mixed-methods empirical study on onboarding in SECOs and is amongst the first empirical studies in the domain of software engineering onboarding in general. Though previous research has been conducted on onboarding within software projects, these works did not provide a deeper understanding of the overall SECO onboarding process, which involves an initial, top-level onboarding phase followed by one-to-one project-specific mentoring. Hence, this chapter aimed to (1) understand the onboarding process at SECO level, as well as to (2) quantitatively validate the impact of SECO-level onboarding in terms of expected benefits regarding diversity, productivity and quality of contributions.

Our observation study of a top-level OpenStack onboarding event yields a catalogue of six conceptual and eight hands-on categories of socio-technical onboarding content, eight teaching strategies used, eight expected onboarding benefits, and 13 onboarding challenges. Furthermore, our quantitative analysis of OpenStack contributors and contributions shows that contributors who followed the onboarding process spend less time and effort to get

their first commit accepted and produce larger, less bug-inducing commits. Moreover, we observe a strong correlation between onboarding and an increase in the gender and technical diversity of the OpenStack SECO. We provide our data set online [120].

The implications of this study are manifold and impact different stakeholders differently: (1) developers have empirical evidence that onboarding could be beneficial for them, since it correlates with increased productivity and chances of being hired by a company of the SECO; (2) (prospective) mentors have an overview of the relevant topics and strategies they should prepare for; and (3) organizations and SECOs as a whole have empirical evidence that investments in onboarding correlate with increased productivity, diversity and quality, while they also have a list of challenges they should be aware of while mounting or operating an onboarding program.

Chapter 4

A Mixed-methods Analysis of Micro-collaborative Coding Practices in OpenStack

Having studied SECO-level onboarding in the previous chapter, this chapter focuses on how SECO stakeholders interact using micro-collaboration practices towards producing value for the SECO, and the extent to which this practice relates to a sustainable SECO in terms of productivity, quality of contributions and retention of contributors. This chapter has been published in the journal on Empirical Software Engineering (EMSE), 18th June, 2022.

4.1 Introduction

In open-source software development projects, it is common for multiple contributors to collaborate on a wide range of activities, from social activities such as submitting a forum question or bug report comment, to technical activities such as developing and reviewing code changes. This chapter focuses on one of these activities, i.e., code contributions, challenging the assumptions that people have made thus far about the way in which collaboration processes exploit distributed version control systems.

Traditionally, collaborations between open-source developers take the form of so-called *macro-collaborations*, which focus on large-scale collaborations by entire teams on complex features for extended amounts of time, typically on dedicated feature branches [50, 152–155]. While developers do interact during such collaborations, their contributions tend to be nicely modularized in their own commits for which they are recorded as sole author by version control systems like git.

While macro-collaboration involves multiple developers contributing one or more commits, in contrast *micro-collaborative* coding is a more fine-grained type of collaboration that involves multiple developers contributing *to the same commit*. Hence, instead of working asynchronously, micro-collaboration involves synchronous technical activities on a shared code base. This practice is claimed to have various benefits within teams, such as solving complex problems [156–162] and improving onboarding [64]. At the same time, micro-collaborations are essential both in in-person settings (cf. traditional pair programming) and when working in an online, global software development environment (as has become the norm since Covid-19), when the required contributions by the different developers are impossible to decompose cleanly into separate commits.

Despite the many advantages of micro-collaborations, and their similarity to agile practices such as pair programming, modern version control systems for online collaboration like git unfortunately do not provide built-in support to track such collaborations, simply because, by design, a git commit can only have one author and one (possibly different) committer. The author is the one who created the content, and the committer is the one who committed it to the repository. Similarly, code review systems such as Gerrit[1], or reviews integrated in GitHub's pull request mechanism, only allow one individual to be the author of a code change.

[1] `https://www.gerritcodereview.com`

As such, for a long time developers have had to come up with workarounds to still attribute micro-collaboration changes to all responsible collaborators, which is essential for accountability, copyright, etc. purposes. For example, in November 2007, the Debian community raised awareness[2] for git to support multiple authors for a commit, and in March 2012, the Eclipse developer community expressed the need to attribute multiple authors for a commit: *"In the case, for example, of pair programming, we may have a situation where multiple developers should be credited with a commit ... there is no current mechanism to do this. It might be cool to be able to specify multiple values in the 'author' field or multiple occurrences of the author field"* (Wayne Beaton).[3]

Only recently, a *de facto* approach to acknowledge all co-authors of a patch has been integrated in GitHub[4], GitLab[5], etc., basically requiring developers to add specific Co-Authored-By: trailers at the end of their commit message, each one listing the name of one of the main author's collaborators. According to the OpenStack wiki[6], the use of such trailers is encouraged. It is even the only mechanism for micro-collaborative coding that is advertized publicly as *"a convention for recognizing multiple authors, and our projects would encourage the stats tools to observe it when collecting statistics."* Moreover, according to GitHub, *"the new feature responds to a growing need in organizations where collaborative coding is the norm to speed up onboarding of junior developers, to improve code quality through peer programming or code reviews, etc."* [7].

Thus far, there is little empirical evidence about the prevalence of micro-collaboration

[2] https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=451880#12
[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=375536
[4] https://github.blog/2018-01-29-commit-together-with-co-authors/
[5] https://gitlab.com/gitlab-org/gitlab-ce/issues/31640
[6] https://wiki.openstack.org/wiki/GitCommitMessages
[7] https://www.infoq.com/news/2018/01/github-multiple-author-commit/

practices in large open-source development communities, and the expected and actual benefits of such practices. Worse, there is no data about the extent of micro-collaboration before commit trailers were established, nor to what extent those older practices still prevail. To address this lack of knowledge, this chapter carries out an exemplar case study of micro-collaboration practices. Such exemplar case studies are a well-accepted, yet underexploited, method in empirical software engineering to gain fine-grained insights and understanding of specific phenomena [163].

This chapter studies as an exemplar case the OpenStack ecosystem[8], a popular open-source platform for cloud computing governed by the Open Infrastructure Foundation (OIF). To do so, we follow a mixed-methods research methodology [164] combining qualitative semi-structured interviews of OpenStack contributors with quantitative statistical analysis of historical data extracted from version control and code review environments.

First, we perform an interview study with 16 participants to answer the following research questions:

RQ_1 : **How do OpenStack contributors engage in collaborative coding practices?**

RQ_2 : **What are the benefits of micro-collaboration?**

RQ_3 : **How can micro-collaboration be improved further?**

Second, we perform a quantitative study on more than 900k commits and 600k Gerrit change sets of more than 1.8k OpenStack projects to validate the key findings of the qualitative study. Furthermore, we validate our qualitative and quantitative results with nine experts, all members of the Technical Committee at OpenStack. As a result, we provide the following main contributions:

[8] https://www.openstack.org/

1. Ten qualitative findings describing the perceived practices, advantages and shortcomings of micro-collaboration practices at OpenStack;

2. Empirical evidence validating five of these qualitative findings, notably we confirm that micro-collaboration correlates positively with:

   (a) onboarding and retention;

   (b) a lower likelihood of introducing bugs;

   (c) more and larger code reviews, patch sets and commits;

3. We observe that the large majority of micro-collaborative changes are not explicitly marked as such using commit trailers.

The chapter's main message is that micro-collaboration matters in distributed development, hence it makes sense for open-source communities to promote and support this practice. Yet, one cannot only rely on the presence of co-author trailers in commit messages. Open-source communities could adopt the various heuristics used in our study to identify micro-collaborations hidden in Gerrit code reviews. A replication package containing all necessary details of the qualitative analysis of the interviews, as well as the data and scripts of our analyses is available [165][9].

4.2 Background

The goal of this chapter is to understand the phenomenon of micro-collaborative coding practices in open source software projects that are part of a larger software ecosystem. This requires a case study containing multiple interdependent project teams with many

[9]The replication package can be found on Zenodo: `https://doi.org/10.5281/zenodo.5759968`

distinct contributors of diverse nature. Moreover, the community should be open to micro-collaborations, and have established mechanisms and tools to support it. In addition, the ecosystem should be sufficiently long-lived, and contain traceable and accurate data logs of its software development history. OpenStack satisfies all of these criteria, which is why it was selected for our case study.

4.2.1 About OpenStack

OpenStack is an open-source ecosystem for cloud computing that was jointly developed by NASA and Rackspace in 2010. It is available under the Apache 2.0 license, and follows a 6-month official release schedule, with releases ordered alphabetically starting with "Austin" in October 2010[10]. While this first release included only two initial projects (Nova and Cinder), OpenStack has been growing steadily over time, and currently comprises over 60 core projects[11].

As of March 12th, 2021, OpenStack's code base of 20M lines of code involved code contributions by over 100k community members including volunteers and developers employed by over 710 companies[12] (including large multinationals like Microsoft, Facebook, Huawei and Red Hat), spread across 187 countries. There is also a large base of companies and software ecosystems (SECOs) relying on OpenStack services, such as Apache Software Foundation, etc. OpenStack development involves over 2k projects/sub projects [64], 63 of which are so-called core projects [166]. Zhang et al. [59, 167] have studied how a large-scale complex system such as OpenStack is developed by various companies that are collaborating with different OpenStack projects. They found eight models of collaboration

[10] https://releases.openstack.org
[11] https://governance.openstack.org/tc/reference/projects/
[12] https://www.openstack.org/annual-reports/2020-openinfra-foundation-annual-report/

among these companies and OpenStack that differs in their objectives and contribution performance, despite their differences, these companies work together as an ecosystem with common goals, which is to release OpenStack. The Open Infrastructure Foundation (OIF), formerly known as OpenStack Foundation, was founded in 2012 with the objectives to promote, empower and protect the OpenStack software and its community. The Foundation staff Members (FM) are the custodians of the OpenStack trademark, controlling the flow of budget and defining the goal of the Open Infrastructure Foundation. The Technical Committee (TC) members are responsible for all technical matters concerning the foundation, and they control all the upstream OpenStack projects [168], such as the official core projects.

4.2.2 The OpenStack development workflow

The overall OpenStack collaborative development process is comparable to what one expects to find in large collaborative open source communities. The specific implementation of this workflow at OpenStack relies on a variety of tools, including: (i) a distributed version control system (git) that hosts the code base of all projects in its public repositories; (ii) a code review system (Gerrit) for patch sets and new features across all projects; (iii) a task and issue tracker (that has migrated from Launchpad[13] to Storyboard[14]); (iv) a continuous integration (CI) system (Zuul[15]).

Figure 4.1 shows the common development workflow that OpenStack project contributors use. Each stage is numbered chronologically. In step ①, as soon as a git account is set up, the contributor can select a desired upstream project and create a branch on his local environment to clone the project into. In step ②, the contributor should switch from

[13] https://launchpad.net/openstack
[14] https://storyboard.openstack.org
[15] https://docs.openstack.org/infra/system-config/zuul.html

the master branch to their local branch to make all necessary changes to the source code. Next ③, the contributor needs to run unit tests against the changed code and commit it to a staging area. Then ④, the change set can be submitted to the Gerrit code review system [68, 169] and can be iteratively amended ⑤ by code reviewers and the original contributor(s) through a collaborative process. When the code reviewers eventually approve the change set, the latest patch set is sent to the CI tool (Zuul) for automated testing and compilation ⑥. If the CI tests are successful, the changes are merged into the upstream project ⑦.

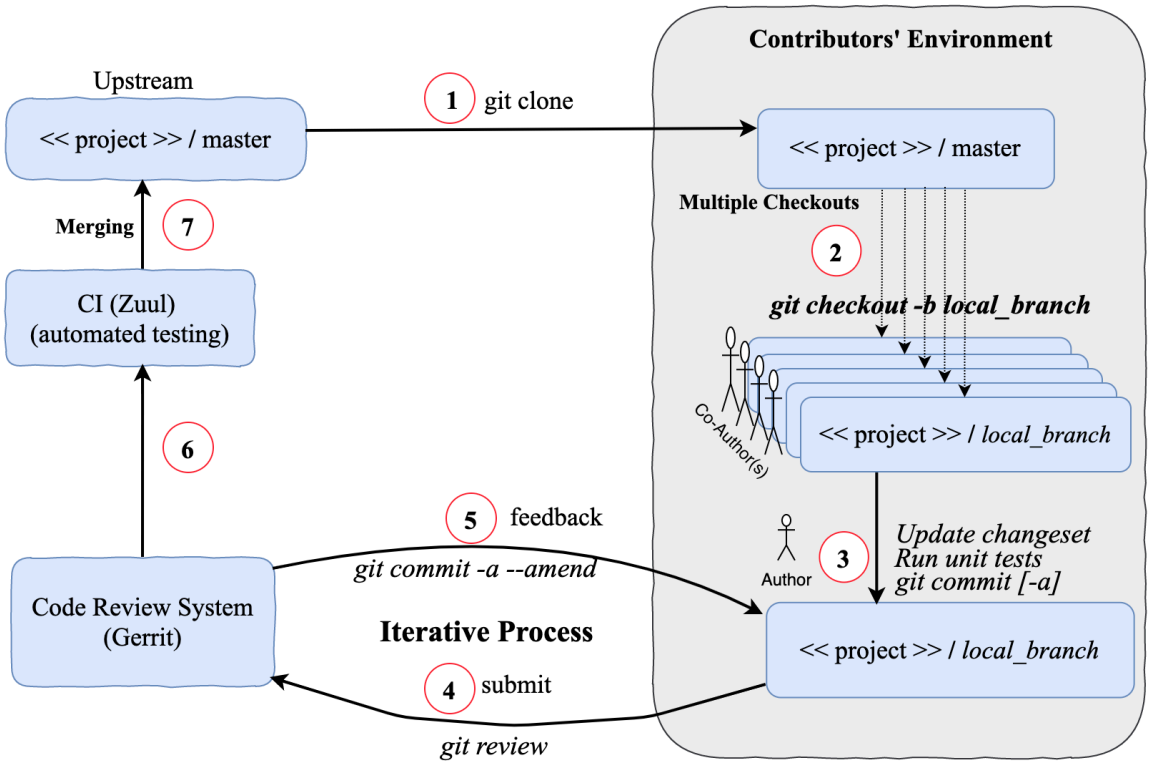
Terminology-wise, a *change set* in Gerrit corresponds to a given feature or bug fix for which a contributor submits an initial *patch set* [170] for code review. The iterative code reviewing process leads to a series of subsequent patch sets, corresponding to updates or fixes to the initial patch set. This series of patch sets belongs to the same change set with a unique Change-Id identifier. These identifiers will be referred to as *change identifiers* henceforth. If the code changes within a patch set are co-authored, OpenStack recommends that the commit message can be used to indicate that multiple people have been working on a particular patch, using the git commit message trailer Co-Authored-By: <name> <email>[16].

4.2.3 OpenStack contribution and attribution policy

People can contribute to the OpenStack community[17] in many different ways and there are also different types of rewards[18]. Contributions can be technical[19], social[20], or administrative[21]. Technical contribution happens through an upstream project, for example by

[16] https://wiki.openstack.org/wiki/GitCommitMessages
[17] https://tinyurl.com/49xw7r37
[18] https://tinyurl.com/4tfsbu73
[19] https://wiki.openstack.org/wiki/AUCRecognition
[20] https://superuser.openstack.org/articles/auc-community/
[21] https://wiki.openstack.org/wiki/Community/AmbassadorProgram

Figure 4.1 Gerrit code contribution process in OpenStack (adapted from *).



*https://docs.opendev.org/opendev/infra-manual/latest/gettingstarted.html

contributing to its documentation or code base, or by participating as a chair for any of the project's technical conferences and summits. Social contribution can occur by engaging in the social life of the community, for example, by planning community events such as summits, project team gatherings, forums, etc. Contributions can also be of administrative nature, for example through participating in the OpenStack Ambassadors' program or serving the foundation through the Technical Council (TC) or Project Team Leads (PTL). Besides, since OpenStack does not discriminate[22] among these types of contributions, but instead gives equal importance to contributors, every single contribution should count towards reward and recognition. This chapter will only focus on micro-collaboration to the code base, thus, we will only investigate attribution and co-authorship in the code base.

[22] https://docs.openstack.org/contributors/code-and-documentation/introduction.html

4.3 Setup of interviews with OpenStack contributors

The research questions raised in the introduction aims to understand and document OpenStack's practices for micro-collaboration. Are git commit message trailers used consistently? What other collaborative coding mechanisms are used frequently? What are the benefits of micro-collaboration? To address these and other related questions, we have conducted semi-structured interviews [171] with OpenStack practitioners during the Open Infrastructure Summit (OIS) in Berlin, Germany from 13 to 15 November 2018.

4.3.1 Selection and demographics of participants

For the interviews, we required OpenStack practitioners to share their view on the potential added value that collaborative coding brings to the ecosystem. To gain different perspectives on the problem, we sought to obtain the opinion of Foundation staff Members (FM), as well as Active Technical Contributors (ATC) having contributed to at least one OpenStack project and having at least three years of experience within OpenStack.

In November 2018, we carried out an initial quantitative analysis on OpenStack's git repositories to identify code collaborators in terms of git co-authored trailers. Based on this, we extracted a list of contributors having co-authored at least one patch set, and we sent customized emails to three senior contributors that the author of this thesis knew personally. For broader coverage and response we drafted a recruitment screener [172] and sent it to the OpenStack developers' mailing list as well through personal emails, inviting interested OpenStack practitioners to participate in the interview.

In total, we received 32 prospective participants for the interviews, of which 24 ATC and 8 FM (6 confirmed upfront through email confirmation, and 2 more were recruited onsite through snowball sampling [173] upon suggestion by a senior FM). During the summit,

as the interviews progressed, we noticed after the 16th participant that no additional new information was provided on top of what previous participants had said; we thus reached a point of saturation [174,175], similar to how previous studies [176,177] reached saturation at 16 and 10 interviews, respectively. In qualitative studies, saturation usually happens during data collection and analysis when no new theme emerges from any additional data collected, i.e., in our case from the interviewed participants. Therefore, we stopped the interview session and discarded the redundant interviews.

The demographics of the 16 retained respondents is shown in Table 6.1, with seven interviewees self-identifying as female and nine as male. The respondents were spread across different geographical regions. Only three of the respondents were contributing to OpenStack on a volunteer basis, while the other 13 were hired (9 were hired directly by companies participating in the OpenStack consortium, whereas 4 got hired through OpenStack events or summits). Respondents T_1 to T_{11} were all active technical contributors (ATC) to project teams, while respondents B_1 to B_5 were additionally involved in a more organizational role at OpenStack, such as being an FM or TC member, project team leader (PTL), or member of the infrastructure (Infra) team. Their experience in OpenStack ranged from three to nine years, and they were involved in between 1 and 11 different projects.

4.3.2 Interview recording and transcription

All interviews were semi-structured and performed by the author of this thesis during the 2018 OpenStack Summit in Berlin. The 16 recorded interviews lasted a median of 17 minutes. Before the interview, each respondent was asked to fill a consent form. Most of the questions were open-ended. The questionnaire is shown in Appendix .2. The interview guide was designed to include three main series of sections:

Table 4.1: Demographics of interview respondents.

ID	Region	Gender	OpenStack role	Status	#projects	#years
T_1	North America	♂	ATC	hired	5	6
T_2	Europe	♀	ATC	hired	4	5
T_3	Asia	♀	ATC	hired	3	3
T_4	Europe	♂	ATC	hired	3	4
T_5	North America	♀	ATC	volunteer	1	4
T_6	North America	♂	ATC	volunteer	11	6
T_7	Asia	♂	ATC	hired	4	4
T_8	Africa	♀	ATC	volunteer	1	8
T_9	North America	♂	ATC	hired	5	9
T_{10}	Europe	♀	ATC	hired	2	3
T_{11}	Asia	♂	ATC	hired	8	9
B_1	North America	♀	PTL/TC	hired	7	3
B_2	Asia	♂	FM/Infra	hired	4	5
B_3	Europe	♂	TC/Infra	hired	5	9
B_4	South America	♀	PTL/TC	hired	2	9
B_5	North America	♂	PTL/FM	hired	3	9

1. general questions aimed to understand the background and demographics of each participant;

2. questions targeted to respondents of type B_k (having an organizational role in OpenStack) to gain more insight in the mechanisms and modalities OpenStack governance has put in place to encourage the practice of collaborative coding; and

3. questions targeted to respondents of either type T_i or B_k to capture the technical perspective of collaborative coding.

After the summit, the recorded interviews were distributed equally over the last three authors of this chapter for transcription. The interviewer double-blinded the respondent names before providing the recordings to the transcribers. Each interview transcript was stored in a file labelled by a unique identifier associated to the respondent. Each transcriber received instructions to further anonymize the transcripts by hiding any personal data, in

order to anonymize any privacy-sensitive information.

4.3.3 Interview coding and reviewing

Our qualitative methods consisted of: (1) a thematic analysis [178–180] from the transcribed data to uncover themes such as the expected benefits, challenges and communication mechanisms at OpenStack; and (2) a content analysis [181, 182] to understand the prevalence of emerged themes that practitioners discussed about during the interview. To uncover themes from the transcribed corpus, we started with a qualitative coding process. We identified how identical codes form categories of high-level themes. We kept track of the prevalence (i.e.popularity) of the emerged themes as they appear in each code category. Below, we provide more details.

After transcription of the 16 interviews, which form our data points at document-level, we started a two-phase process consisting of (1) *coding* the transcripts; and (2) *reviewing* the coding to reach mutual agreement. The *coding* phase started with an initial *inductive coding* [183] conducted by one author who used two transcripts (corresponding to $10-15\%$ of the total set of transcripts [184]). He assigned labels to the transcribed text, without any predetermined theory, structure or hypothesis. As the coding progressed, common themes started emerging, that were classified into code categories. The coding process continued until all transcribed text was coded. This process resulted in an *initial codebook* [185].

Next, all authors were involved in a *deductive coding* process [183] using this initial codebook as a guide. Each author independently coded the interview transcripts that were initially assigned to them. Whenever a new theme emerged during this coding, the authors would discuss, resolve and manually harmonize the resulting codebook as described below.

In the *reviewing* phase, each author was assigned for coding a different set of four

transcripts that had already been coded in the first phase by another author of this chapter. The purpose was to assess whether both coders reached a sufficient rate of agreement on the performed coding. For each code found in each transcript, value 1 was assigned if both coders agreed on the code category, and value 0 otherwise. Based on the categorical structure of the data, and the use of two raters, we used Cohen's κ [186] to calculate inter-rater reliability (IRR) [187]. We used Landis' agreement levels [119] to interpret the IRR as *poor*, *slight*, *fair*, *moderate*, *substantial* or *excellent*.

To iteratively improve the IRR, we performed three rounds of negotiated agreement [187]. The first round of coding already yielded a *substantial* IRR of $\kappa = 0.69$. During the second round the raters negotiated the disagreements that were encountered during the first round using a combination of online and in-person discussion. After sorting out these differences the improved IRR became *excellent*, with $\kappa = 0.81$. In the third round, we rearranged and merged certain code categories that were considered contextually similar. This lead to a final coding structure that satisfied all raters, with a perfect coding agreement of $\kappa = 1$.

The resulting codebook, provided in our online replication package [165], enables us to understand practitioners' perspectives on collaborative coding in a complex ecosystem. Ten themes emerged from the transcribed text that depict the advantages and benefits of collaborative coding in OpenStack. Section 4.4 reports on these themes derived from the qualitative results together with their frequencies.

4.4 Qualitative results of interviews

This section reports on the findings that we derived from analysing the interview transcripts. They provide qualitative evidence to answer the research questions outlined in Section 4.1. Numbered from **F1** to **F10**, the findings are summarised in Table 4.2 and discussed in the

following subsections.

RQ_1 How do OpenStack contributors engage in collaborative coding practices?

> **F1** While OpenStack uses git feature branches for *macro*-collaboration, there are two main mechanisms for *micro*-collaboration: (a) Co-Authored-By: commit trailers; and (b) Gerrit code reviews. Even though OpenStack and GitHub encourage the use of commit trailers, it is not considered to be the most frequently used approach for micro-collaboration.

We asked the interviewees about the collaborative coding mechanisms they were aware of, or had personal experience with, in OpenStack projects. All different mechanisms they mentioned are discussed below. Some of them correspond to macro-level collaboration, while others correspond to micro-level collaboration. In the remainder of this chapter, we will narrow down our focus on micro-level collaboration between individuals.

(a) Co-Authored-By: trailers in git commits. The **micro-collaborative** coding mechanism that was mentioned by all 16 respondents consists of appending trailers of the form Co-Authored-By: <name> <email> to git commit messages to indicate the contributors that collaborated on a particular patch (see Section 4.2.2). While all respondents were aware of such commit message trailers, two of them (B_1 and T_5) did not participate in co-authoring activities within OpenStack, while three respondents (B_1, B_5, and T_{11}) only experienced this technique outside of OpenStack.

Three respondents stated that they used such trailers when it was important to list all co-authors in a code commit that is a product of collaboration, and six participants said to

Table 4.2: Research questions and associated thematic findings on micro-collaboration practices based on the transcribed qualitative data. For the findings highlighted in grey we will provide additional quantitative empirical evidence in Section 4.6.

RQ	Finding	Emerging theme on micro-collaboration
RQ_1	**F1**	There are 2 main mechanisms for micro-collaboration.
	F2	Most frequent communication media are IRC and Gerrit.
RQ_2		*Collaborative coding* ...
	F3	... encourages teamwork.
	F4	... improves onboarding.
	F5	... enhances learning and understanding.
	F6	... improves software quality.
	F7	... improves productivity.
	F8	... enables recognition and accountability.
	F9	... facilitates solving complex problems.
RQ_3	**F10**	There is a need for automation, tracking and awareness.

systematically use the trailers during collaborative coding. Three other respondents said that they sometimes forget to add the trailers because they are not interested in the perks coming with the collaboration (e.g., credits to attend OpenStack events). Two respondents said they only use the trailers for non-trivial changes. When asked how frequently trailers were used across the OpenStack projects, only three respondents claimed that it was extremely common. The other 13 reported that the mechanism was used even though it was not a frequent practice.

(b) Gerrit code review collaboration. The second most mentioned technique for collaborative coding is through Gerrit code reviews, and could support either micro- or macro-collaboration.

Respondents mentioned the following two code review mechanisms that can be considered as **micro-collaboration**:

1. Five respondents mentioned the practice of co-authoring by uploading a new patch

set version of someone else's patch set, under the same Gerrit change id. As such, one can download a patch under review, modify it, then re-upload a modified version that supersedes the previous patch version. This approach is easy to use and preserves the original co-authoring information. It also registers the information of the new author(s) in the final commit. Three of these respondents mentioned the use of Gerrit's web interface for *inline patch editing*[23], allowing to quickly generate a new patch set for minor edits such as fixing typos.

2. Four respondents considered the act of providing *code review comments* as a form of code collaboration, typically when they expect no credit for their contributions in the form of co-authorship. This is the most loose interpretation of micro-collaboration that we encountered, and we did not study this further.

In terms of **macro-collaboration**, four respondents discussed the practice of writing a separate change set dependent on someone else's change set, such that the combination of both change sets achieves the intended functionality. The advantage of this approach is that both authors get full credits, since they each submit their own change set (and hence commits). The disadvantage is that it requires the code contributions to be cleanly decomposable, and that it introduces strong dependencies between patches. This approach is often used when the original contributor is no longer available or interested to work on the patch.

(c) git feature branches. Four respondents mentioned the **macro-collaborative** coding mechanism of feature branches in git. Development of certain complex features at OpenStack requires a large-scale, macro-level collaboration among cross-project teams. Such features usually take longer than expected to develop (otherwise the patch set approach

[23] https://gerrit-review.googlesource.com/Documentation/user-inline-edit.html#editing-change

mentioned previously could be used). As such, development takes place in a *feature branch*[24] with several iterations of commits. When development is completed, these features are later merged into the master branch. B_3 explains: *"You just set an official feature branch where you land those various patches and then once the branch is ready you can squash it as a single commit and merge it back to the main, with all the co-authors. So that would be: the lightweight ... spontaneous ... and the industrial way of doing it."*

> **F2** IRC is the most frequently reported communication medium for joint contributions, followed by code review comments.

While joint contributions could potentially involve many different communication channels, the responses highlighted that some of them were more common. Table 4.3 reports six different communication channels that are used for collaborating within OpenStack. IRC is the most common, followed by code review comments, communication through mailing lists and git commits. More traditional forms of private communication, such as personal emails and phone calls, appear to be less common or were simply not mentioned.

RQ_2 What are the benefits of micro-collaboration?

OpenStack contributors perceive benefiting from micro-collaborative coding for a diverse set of reasons. Based on an analysis of the contents of the interview transcripts, Table 4.4 summarises in decreasing order of frequency the benefits that respondents perceive from micro-collaborative coding. The frequency of each benefit indicates how prevalent contributors perceive its value in collaborative coding. **F3**, **F4**, **F5** and **F6** are considered the most frequent benefits, totaling 75.4% of all responses. Each perceived benefit is discussed in

[24] https://docs.openstack.org/infra/manual/drivers.html#feature-branches

Table 4.3: Communication media used for collaboration, per number of respondents (n), in percentages (Pct). IRC and Gerrit-comments are the most used media with 53.7%.

Communication channel	n	$Pct.(\%)$
IRC	13	31.7
Code review comments	9	22.0
General mailing list	8	19.5
Git commits	6	14.6
Personal email	4	9.8
Video/phone conferencing	1	2.4

detail below.

F3 Co-authoring encourages teamwork.

14 respondents consider co-authoring as a mechanism that both enables and exploits the benefits of collaboration and teamwork. For example, co-authoring was said to facilitate the planning and actual development of code contributions, to provide more confidence about the quality of a contribution, and to speed up development in areas with high complexity, even of larger contributions. T_4 affirms that it is *"nice to know that it wasn't just one person thinking through the design of it and developing it but that it was multiple people collaboratively doing it"*.

Co-authoring encourages diversity amongst contributing developers (in terms of thinking, skills, expertise, seniority, etc.) and provides a common platform for developers to share expertise; divide and conquer. By awarding credits for OpenStack events to all co-authors, co-authoring provides an additional incentive to make developers collaborate. Finally, T_5 signaled more implicit cases of co-authoring, where a contributor is *"... picking up things that have just been abandoned or left or people don't have time for ..."*

Table 4.4: Perceived benefits of micro-collaborative coding at OpenStack in descending order of respondents (n) per percentages (Pct).

Preceived benefits of micro-collaboration	*n*	*Pct.*(%)
F3 Encourage team work	14	24.6
F4 Improve onboarding	14	24.6
F5 Enhance learning and understanding	8	14.0
F6 Improve software quality	7	12.2
F8 Enable recognition and accountability	5	8.8
F9 Facilitate solving complex problems	5	8.8
F7 Improve productivity	4	7.0

F4 Collaboration is associated with a higher likelihood of onboarding success.

All respondents emphasized the importance and benefits of collaboration and mentoring for onboarding newcomers. For example, B_3 reported that collaboration makes it easier for new contributors to submit their first patch: *"You notice that they might be too shy to author completely a patch, but if they can be counted as a co-author, or if they wrote the design of it or the documentation of it that does not really show in the code, then you can credit them for their work. So, I feel like it is a positive system."*

Eight respondents reported that they started co-authoring as junior developers, whereas five claimed they were already experienced. Only four respondents mostly did co-authoring with less experienced people (typically as mentor), while the other twelve respondents typically performed co-authoring while being mentored by more experienced contributors. Hence, new contributors as well as more experienced ones seem to benefit from learning from more senior contributors, allowing a better understanding of the design and deeper technical details of projects.

While one respondent claimed that the person who is doing most of the work is recorded as lead author, fourteen other respondents instead claimed that the lead author would just be the person doing the final submission of the patch.

Eight respondents mentioned both the social and technical aspects that can influence contributor's retention/abandoning/turnover. Four respondents stated that the TC is doing its utmost to improve OpenStack's work culture, besides just being aware of unpleasant experiences that contributors have had in the past. For example, B_3 said *"we try to keep it fun and engaging, ... make sure that people are happy contributing to OpenStack and that they want to contribute to OpenStack."* OpenStack has also put in place an onboarding program to attract new contributors and keep them engaged [64]. Moreover, T_9 mentioned the benefits on feedback during collaboration: *"... like the onboarding sessions that we do here attempt to get you familiar with the specifics of a project but the toolset in general. But I think that's also a good way for feedback in regard to the retention; for feedback where things might not be working well. That's how we find out."* Collaboration creates an atmosphere for contributors to share immediate feedback on their work progress, since collaborators are accountable to one another. This allows to address obstacles timely before they may lead to abandonment.

F5 Collaboration correlates with learning and understanding.

Eight respondents indicated that collaboration enhances learning and understanding of the code base, since it (i) encourages junior developers to learn the development process faster; (ii) helps to better understand other people's context, component, expertise and problems; and (iii) facilitates better comprehension of the scope and complexity of the overall project.

F6 Collaboration is associated with a higher likelihood of improving software quality.

Seven respondents stated that collaborative coding fosters higher-quality contributions, for example in terms of (i) coding style and clarity and (ii) code correctness. Since every contributor brings in her own specific expertise, the resulting contribution becomes more than the sum of its parts. To sustain this claim on collaborative coding, B_1 asserts that *"... to ensure that the code has the style that matches the rest of the library code for that particular project and that it is readable and simple and accurate, correct. So, all the usual reasons that you do code review."* In addition, B_2 advocates *"... When more than one person contributes, it helps the quality of the code. "*

F7 Collaboration is associated with a higher likelihood of improving productivity.

Four respondents highlighted that collaboration increases productivity, since there is no more need to wait for other people to fix something. Instead, one can just collaborate to make things move quicker. At the same time, the fact that a group of people is working together on a contribution allows to bring larger changes faster, while everyone only needs to perform a part of the effort (less work for everyone involved). This was highlighted by B_6: *"... most projects do encourage that, it's a good way that people can work together and come up with changes ... than one person or they are able to get more work done than one person can do by themselves.""*

Two respondents provided another perspective on productivity improvement, namely that co-authoring stimulates developers to not just reject other people's contributions (with a high chance the rejected contributions would never make it), but instead encourages them

to collaborate themselves on the broken patches. As such, a small fix could still allow a broken patch to go in, rather than the project losing out on it or stalling while waiting for improvements by the patch author. As such, this mechanism leads to an improvement of the code review culture. Related to this are the aforementioned implicit collaborations, where contributors can effectively pick up an abandoned patch or a patch people do not have time for anymore (with their permission, if needed).

F8 Collaboration enables recognition and accountability.

Five respondents declared that co-authoring brings about individual recognition. It allows acknowledging everyone involved in a joint contribution, so that even the smallest contribution gives visibility to its contributors. Six respondents uphold that one major way of thanking joint contributors is by providing credits to them for each contribution. Such credits enable contributors to obtain a rebate on the registration cost for OpenStack events, such as a project team gathering or summit. In addition, credits allow contributors to vote or stand in elections.

Visibility and recognition also implies accountability, since in the case of a bug or other issues the responsible can be tracked easily. This is an important aspect of joint contribution, yet not everyone follows these guidelines consistently. Especially the more experienced developers would forego being recorded as a collaborator, since they already have enough visibility in the community or do not need the extra credits (they have direct access to all OpenStack events). By doing so, the accountability aspect is lost.

Furthermore, two respondents mention that the recognition provided by joint contributions positively reflects on the public image of the contributor's company. This is why many companies actively track their employees in order to encourage such contributions.

T_1 highlights that it makes *"... certain sense for some companies that employ contributors that having a name attached to a patch is important because it gives that company recognition in addition to the individual contributor."*

Related to **F6**, the provisions for accountability also provide more confidence in the quality of joint collaborations, since, in the event that one of the developers of a project leaves, the other collaborators still share a sizeable amount of knowledge and expertise about the contributed code. Hence, the organization can continue to work without being impacted.

T_5 emphasises accountability during collaborative coding: *"Usually most people try adding* co-authored-by *... so at least they have tracked accountability. And very rarely do we run in cases where people actually completely pull a patch down, recommit it without."* T_3 claims collaborative coding is important to credit or encourage collaboration: *"It gives credit or credit is due if many people are involved in an idea then they should all get credit for the idea."*

Accountability of co-authorship can also be important because of legal obligations. This was highlighted by respondent T_3: *"... giving credit is important for legal reasons ... someone created some code but if there is no attribution then that causes legal problems."*

F9 Collaboration facilitates tackling more complex problems.

Five respondents stressed that joint contributions enable a project to deal with complex and inter-dependent multi-person patches. These are patches that cannot be decomposed into separate parts, but need to be submitted as one. Similarly, co-authoring can help to break down complex tasks for contributors with different expertise. For example, B_5

reported that *"The reasons in which I contributed was mostly on complex task as we applied the divide and conquer technique."*

RQ_3 How can micro-collaboration be improved further?

When respondents were asked about whether they were satisfied with how OpenStack supports collaboration and co-authoring, 15 respondents said they were generally happy, and 11 could not think of any specific drawback. When asked more specifically about things that could be improved in the processes or tools for co-authoring, responses were more diverse, as shown in Table 4.5. Automation, tracking and promotion account for 71% of the total demands.

Table 4.5: Mechanisms to improve collaboration practices, per number of respondents, expressed in percentages (%).

Things to improve	# respondents	%
Automation	10	32.3
Tracking	7	22.6
Promotion	5	16.1
Enforcing "Co-Authored-By:"	3	9.7
Documentation	3	9.7
Communication	2	6.5
Licensing	1	3.2

F10 Better automation, tracking and awareness of joint contributions are the most commonly mentioned requests for improvement.

The respondents raised various concerns regarding the process and tools used for co-authoring commits, and suggested mechanisms that could improve joint collaboration. For

example, as Table 4.5 shows, ten respondents mentioned the importance of more automated tools involved in co-authored commits, for example to automatically insert trailers. Seven respondents asked for better ways for tracking (co-)authorship, since for example the Gerrit-based workflows require manual effort to look up all versions (sets) of a patch to identify

all co-authors, and contributions other than source code are not tracked that well. Moreover, automating the tools should also facilitate the process of adding co-authors, standardize the information provided in git commit messages, and speed up collaborations.

Five respondents stated that the practice of co-authoring commits should be promoted more actively, to create more awareness and hence achieve more collaboration. In addition, three respondents said that the usage of a Co-Authored-By: commit trailer should be enforced when co-authoring commits. This might partly be due to lack of clear documentation of the joint collaboration process (two respondents).

Two respondents emphasized the need for better communication means during collaborative coding activities. Finally, respondent T_5 advocated that collaborative coding could result in licensing violations [188], since more people are involved and might (accidentally) bring in source code protected by different licenses: *"... There is a lot of licensing problems that might occur ... You have to be generally careful. If it is already in the repo than naturally it is licensed, and they should be honoring our licenses."*

4.5 Setup of quantitative study

This section explains the setup of our quantitative study on OpenStack aiming to find statistical support for the key qualitative findings of RQ_1 and RQ_2 that were highlighted in gray in Table 4.2. We will empirically evaluate these qualitative findings on the basis of the

quantitative study outlined in this section.

4.5.1 Extracting micro-collaborations

The qualitative analysis of Section 4.4 revealed two important sources of micro-collaboration, namely, co-authorship trailers in git commit messages, and patch sets for Gerrit change sets to which multiple individuals have contributed. This subsection presents the datasets used in our quantitative analysis.

C_{init} **= initial dataset of git commits.** OpenDev is the integrated collaborative open source platform hosting the OpenStack code base. On the 22nd of July 2020 we cloned all 2,219 OpenDev git repositories with a total of 1,870,705 commits, in order to identify those contributors that are most involved in co-authorship activities. To gather a history of exactly nine years of OpenStack activity, we removed any activity before the 1st of July 2011 (66,015 commits) and after the 30th of June 2020 (5,503 commits). We selected July 2011 as the start for our commit data set to better align with the Gerrit dataset (described below), since Gerrit was introduced in the OpenStack workflow in July 2011[25].

We ignored 343 repositories and their 97,630 commits that corresponded to retired projects[26], as indicated by their latest commit's message. This left us with 1,701,557 git commits for 1,869 OpenStack projects, covering all branches including feature branches. From these, we removed another 210,462 commits corresponding to the activity of the openstack/openstack repository as it does not contain actual content, rather pointers to other repositories. *As such, the initial commit dataset C_{init} consists of 1,491,095 git commits in 1,868 repositories.*

[25] http://lists.openstack.org/pipermail/openstack/2011-August/022939.html
[26] https://opendev.org/opendev/puppet-ansible

To determine co-authorship in this commit dataset, we proceeded as follows. For each git commit we extracted its author and committer, as well as any contribution that could be retrieved through commit message trailers signaling co-authorship. Officially, co-authored commits should use the Co-Authored-By: commit message trailer (see Section 4.2.2). However, we manually observed and confirmed the presence of many variations of this trailer in the OpenStack git repositories, which we also included as valid signals of co-authorship. More specifically, after conversion to lowercase we found and considered at least one instance of each of the following alternative trailers: author:, co-author:, authored-by:, co-authored:, co-author-by:, co-authored-with:, author attribution:, also-authored-by: and co-authored-by:.

$\mathcal{P}_{init}$ **= initial dataset of Gerrit change sets.** The second data source of micro-collaborations in OpenStack is based on the Gerrit code review system. Micro-collaboration in Gerrit is more subtle than in git since, to signal co-authorship, we had to consider all change set instances where at least one individual other than the initial patch set submitter, modified and uploaded a patch set.

We gathered all change sets through OpenDev's Gerrit REST API[27]. For each change set we extracted all comments, all patch sets and their comments and files touched. The extracted data contains relevant information about which contributor performed which action, such as who commented on a change set or patch set, and who authored and submitted the patch set. On the 22nd of July 2020, we fetched the change sets of all 2,219 git repositories of OpenDev with a total of 733,465 change sets. Similar to the git commit dataset $\mathcal{C}$ we considered a 9-year observation period from the start of July 2011 (the introduction of Gerrit in OpenStack) to the end of June 2020 and removed change sets of retired projects

[27] https://review.opendev.org/Documentation/rest-api.html

and the openstack/openstack repository.

The initial review dataset $\mathcal{P}_{init}$ consists of 675,159 Gerrit change sets and 2,098,269 patch sets in 1,818 projects. The new, merged and abandoned change sets correspond to 13,890, 538,381 and 122,888 change sets, respectively.

4.5.2 Data cleaning

The initial datasets $\mathcal{C}_{init}$ and $\mathcal{P}_{init}$ presented in Section 4.5.1 need to be cleaned in order to merge developer identities and remove bots.

Merging identities. To avoid false positives of collaborative coding and report only true collaborative coding instances, we applied developer identity merging [189] on the datasets $\mathcal{C}_{init}$ and $\mathcal{P}_{init}$. $\mathcal{C}_{init}$ contains 20,312 distinct identities, composed of a name and email address, corresponding to authors, committers and persons mentioned in the commit message trailers. $\mathcal{P}$ contains 18,392 distinct identities that typically contain a name, email and username information[28]; 5,089 of the extracted identities were missing the username.

Prior to merging identities, we manually analysed the combined dataset $\mathcal{C}_{init} \cup \mathcal{P}_{init}$ to recover generic names and emails that are not associated with identities (e.g.root, etc, Your Name, Author Name, root@localhost.localdomain, none@none); we recovered 20 such labels in total. The list of terms was enhanced with an initial invalid name list based on the suggestions of Goeminne et al. [189]. These names and emails where excluded when comparing terms during identity merging.

Then, the author of this thesis acquired a partial ground-truth dataset containing the list of active contributor accounts, with names and username(s) of each contributor mapped

[28]There are identity pairs in $\mathcal{P}$ that contain the same name-email pairs, but the username information only appears in one of them. If we account identities in $\mathcal{P}$ as name-email pairs, the number of identities corresponds to 18,081.

to all their email addresses, as recorded in OpenStack's internal database. This partial dataset did not contain inactive or deleted accounts and was used as the first step to merge identities.

After this merging step, identity names, emails and usernames in the $\mathcal{P}_{init}$ were normalized following the approach of Bird et al. [190] and identities with at least two common tokens were manually inspected to decide whether they needed to be merged. The manual inspection was based on three criteria: (1) name/email/username labels, (2) activity statistics of each identity; and (3) recovery of OpenStack webpages for each identity to verify if the person corresponds to the identities. In case one criterion was not sufficient, then the following one was used to make a decision. For example, the identities Tom Mens <tom.mens@domain.com> and Mens Tom <tom@mens.com> do not share the same email address, but the normalized terms (Tom, Mens) in both identities indicate that they should be merged based on the first criterion. This identity merging process resulted in a final set of 17,195 merged identities in $\mathcal{C}_{init} \cup \mathcal{P}_{init}$.

Removing bots. The second cleaning step consisted of excluding bot activity. Bots are tools performing automated processes and their accounts are disguised as real identities. In the context of identifying collaborative coding of OpenStack contributors, such automated bots should not be considered as collaborators and thus be excluded from our analysis.

To identify bots, we relied on an official OpenStack list of Gerrit accounts corresponding to bots[29]. We enhanced this list in three ways. Firstly, we identified OpenStack system names[30] and manually verified identities that contain terms stemming from the system names. Secondly, we manually identified Gerrit identities having the same name as the

[29] https://wiki.openstack.org/wiki/ThirdPartySystems
[30] https://docs.opendev.org/opendev/system-config/latest/systems.html

official bots used by continuous integration tools; this was achieved using variations of the search term "CI" in their name or username. In addition, the most active Gerrit accounts for different activity types (change set ownership, change set or patch set comments, authored or uploaded patch sets, patch set approvals, reviewing) were manually inspected to recover very active bots that might have been missed by the first two bot identification steps. Using the above process, 322 distinct bot accounts were detected in $\mathcal{C}_{init} \cup \mathcal{P}_{init}$.

Throughout the process, we discovered 14 *mixed profiles* of combined human and bot activity [191,192]. For example, the fictitious merged identity of John Doe <john.doe@domain.com> and John Doe CI <john.doe@domain.com> would be marked as a mixed profile, as the first identity is marked as human while the second is marked as bot.

All activity corresponding to bots and all bot-specific activity of mixed profiles were excluded, thereby effectively removing 572,544 commits from $\mathcal{C}_{init}$, and 43,713 change sets from $\mathcal{P}_{init}$. Moreover, 218 change sets are removed as all their patch sets were authored or uploaded by bots. *This resulted in cleaned datasets $\mathcal{C}_{clean}$ of 918,551 commits in 1,832 projects, and $\mathcal{P}_{clean}$ containing 631,223 change sets in 1,817 projects.*

4.5.3 Dataset alignment

The cleaned datasets $\mathcal{C}_{clean}$ and $\mathcal{P}_{clean}$ still need to be *aligned*. We need to carry out *project alignment* to ensure that we only consider projects that exist in both datasets. Otherwise, we might overestimate the collaboration in one of the datasets if there are highly collaborative projects that are only present in that dataset. Second, in order to compare both datasets we need to carry out *commit alignment* by mapping git commits to the corresponding Gerrit change sets.

Project alignment. There are 1,804 projects in common between the commit dataset $\mathcal{C}_{clean}$ (corresponding to 1,832 projects) and the review dataset $\mathcal{P}_{clean}$ (corresponding to 1,817 projects). *Restricting ourselves to these 1,804 projects, we obtain a final commit dataset $\mathcal{C}$ with 917,970 git commits and a final review dataset $\mathcal{P}$ with 631,201 Gerrit change sets. These final datasets will be used for the quantitative analyses of Section 4.6.*

Commit alignment. To align the individual commits in $\mathcal{C}$ with the merged code reviews in $\mathcal{P}$, we cannot simply rely on commit hash values, since commits can be rebased or squashed [158]. As a first heuristic, we use the change identifiers (change-id trailer in the commit message) that persist when squashing, according to the OpenStack documentation[31]. This heuristic allowed us to align 474,156 git commits and 490,860 Gerrit change sets corresponding to 442,509 change identifiers.

As a second heuristic, we use commit hash values to match commits in the case they have not been squashed. The heuristic aligned 102 commits and change sets that had identical commit hash values, corresponding to 108 change identifiers. This low number is due to the large number of alignments already achieved by the first heuristic. Unfortunately, the second heuristic is not robust due to rebasing and cherry-picking, as in these cases the hash value changes and there is no traceability.

Thus, a third heuristic matches the author and commit messages to align commits as these fields proved to be the most reliable to align commits in the presence of rebasing, according to German et al. [193]. This heuristic allowed us to align 22 commits and 19 change sets, corresponding to 24 change identifiers. In total, 442,637 change identifiers are aligned, corresponding to 474,280 git commits and 490,981 Gerrit change sets. These *aligned* commits and change sets are only used in our analyses when directly comparing

[31] `https://docs.opendev.org/opendev/infra-manual/latest/developers.html`

the collaborative or individual activity between the two platforms and are annotated as $\mathcal{C} \cap \mathcal{P}$ in the remainder of the chapter.

4.5.4 Analysing bug-introducing commits

One aspect of code quality related to (**F6**) is the probability of introducing bug-inducing changes (BIC) when doing micro-collaboration vs. individual coding activities. Bug-inducing commits contain changes to files that introduced new bugs but unfortunately slipped through code review and were only noticed and reported later on [194–196]. Existing work on bug-inducing changes has focused especially on predicting whether a given change introduces a bug [144, 197], as well as on heuristics to trace back from a bug-fixing change to the change that introduced the original bug (in particular the SZZ family of algorithms [196, 198]). We follow a 3-step methodology to mine BIC data:

Step 1 — linking bug ID to bug fixing commit ID

Over the studied period, OpenStack has used both the Launchpad and Storyboard issue trackers. We mined the bug metadata (e.g., ID, date, time, and time zone of bug submission) from both tracking systems, focusing only on "confirmed" issues since those are the categories of issues for which bug fixes are being proposed. This resulted in a list of 1,194,026 issues.

We extracted and parsed all the commit messages from the commit dataset $\mathcal{C}$ to identify bug-fix related commits. We used regular expressions to scan the commit messages for the typical trailers used by developers to communicate the bug ID being fixed [144, 196, 199]:

1. $[A-za-z]*[\backslash -][Bb][Uu][Gg][:][\backslash s]?[\#]?[\backslash d]\{6,7\}$

2. $[Ff][Ii][Xx]?[A-Za-z]*?:?[\backslash s]?[\#]?[0-9]\{6,7\}$
3. $[A-za-z]*[Ll][Pp]?[:]?[\backslash s]?[\#][0-9]\{6,7\}$

The regular expressions enabled us to extract bug IDs from the commit messages. We compared the bug IDs from their corresponding issue report on LaunchPad or Storyboard to the extracted bug IDs in the commit messages. If a match was found, we linked the bug ID to the commit ID (SHA-1) of the bug fixing commits (about 1.2M in total). In doing so, we found and removed 231,726 duplicate bug IDs, leaving us with a total of 917,970 unique bug IDs linked to their corresponding bug fixing commits IDs.

Step 2 — identifying BICs

To identify the BICs for the obtained bug fixing commits, we used the SZZ implementation provided by PyDriller [200]. On each bug fix commit, this SZZ variant performs git blame to identify the last commits that touched the lines fixed by the bug fix. Those commits form the initial set of candidate BICs, since they were the last ones to touch the fixed lines. However, since the actual BICs have already been made before the bugs were reported (otherwise, the bug would not exist yet), PyDriller uses the bug report date to remove the commits made after the fixed bugs were reported. If, after filtering, more than one commit remains, the algorithm considers all of them as BICs. We ran the algorithm on 917,970 commits, yielding **315,690** BICs.

Step 3 — analyzing BICs

Based on the identified BICs, we aim to find out if micro-collaborative coding correlates to software quality. To do so, we use a χ^2-test of independence [201, 202] with confidence level $\alpha = 0.001$. This test measures how far the observed counts of a variable are from

the expected counts if the null hypothesis is true. The further, the more evidence the data presents against the null hypothesis. In our case, the null hypothesis states that *there is no difference in the proportion of BICs between individual and micro-collaborative coding*. To report the observed and expected counts, we first count the number of observations (i.e.observed counts) for two events of interest: event A for the response variable (BIC or non-BIC), and event B for the explanatory variable (collaborative or individual coding). Next, for each of the four cases, we compute Expected Count $= \frac{\text{Total for A} \times \text{Total for B}}{\text{Table Total}}$. This allows to calculate the χ^2 statistic.

4.5.5 Validation of qualitative findings through multivariate analysis

In RQ_2, improvement of software quality (**F6**) and productivity (**F7**) were reported as two of the benefits of micro-collaboration. To quantitatively validate these benefits, we use both statistical tests (for **F6** involving BIC) and multivariate statistical analysis on a large number of observed variables that are potentially related to quality and productivity improvement. Inspired by Siegmund et al. [203] we carry out exploratory factor analysis (EFA) and confirmatory factor analysis (CFA) [204] with structural path analysis to analyze multivariate structures in quality and productivity improvement.

These analyses were implemented in Python using Factor Analyzer[32] and semopy [205]. Full details are available in our replication package [165].

Structural Equation Model (SEM) is a set of statistical models rooted on the assumptions of two requirements (1) the concept under consideration must be a complex multivariate relationship, and (2) the data to understand these relationships should be interval-scaled or continuous data [206], even though Johnson et al. [207] argue that categorical data with more than five categories can still be considered for SEM models [207]. Our observed

[32] `https://factor-analyzer.readthedocs.io`

variables contain a mixture of categorical and continuous data.

In order to study the relation of micro-collaboration with the quality and productivity of code contributions, we started off with a catalogue of metrics known from related work to be related to quality and/or productivity [64, 208–215]. We then filtered out all metrics not fit for SEM models, either (1) because they were not categorical with more than five categories [207], not interval-scaled or not continuous [206], (2) because more than 90% of their values were either zero or absent, or (3) because the metrics were correlated. This led us to the 17 observed variables (metrics) that are listed in Table 4.6. The quality metrics are measured per change set, similar to all productivity metrics, except for those marked as "p^o". The latter metrics measure the productivity of the contributors involved in the change sets. Furthermore, the variables marked as "b" are used both for the quality and change-set level productivity analysis. Note that $collabs^o$ is a contributor-level version of *collabs*, hence has a different definition. A separate model will be built for change set- and contributor-level productivity.

Exploratory factor analysis (EFA)

CFA (and SEM) assume the presence of factors, i.e., latent variables that aggregate semantically related observed variables. One can either hypothesize such factors a priori, or use Exploratory Factor Analysis (EFA) to identify factors semi-automatically. EFA basically does this by reducing the dimensionality of the observed variables such that the identified factors are explained by the principal components of the observations. Similar to Siegmund et al. [203], we use EFA to semi-automatically extract the factors, together with the factor loadings, i.e., the degree to which each observed variable is associated with each latent variable. We do a separate EFA on the observed variables related to software quality, those

Table 4.6: Observed variables (metrics) for software quality (q), software productivity (p) and both (b). The productivity metrics marked as "p^a" apply to change sets, while those marked as "p^o" apply to contributors.

Observed variables	Description
coreReviewers (q)	Number of experienced developers (core reviewers and PTL-Approved) with core power (+2) involved in a code review process. At least two +2s are required to trigger continuous integration. [213]
juniorReviewers (q)	Number of junior developers (reviewers) with a +1 power [213]
reviewPriority (q)	Review priority of a patch set, the higher the more immediate attention required from experienced developers. [211] [210]
gating (q)	Number of times that CI/CD runs functional and integration testing jobs (successfully or not) [213]
verified (q)	Number of approval scores (+2) from automated testing, etc. [213]
workFlow (b)	Number of $\pm$[1,2] scores for best practices including coding style, work-in-progress, etc. [215] [211]
comments (b)	Total number of comments on a change set. [209]
commenters (b)	Number of contributors posting comments in addition to the original author of the change set. [209]
churn (b)	Number of added and deleted lines of code. [212]
changedFiles (b)	Number of changed files in a change set. [210]
status (b)	Status of patch sets, which can either be "Merged" or "Abandoned". [212]
collab (b)	Indicates if contribution to a change set was collaborative or not (bool value, 1 if collaborative and 0 otherwise).
duration (p^a)	Time required to complete the code contribution. [209]
codeReviews (p^a)	Number of change sets under review. [208, 209]
collab^o (p^o)	Indicates if a contributor has collaborated at least once with another contributor (bool value, 1 if collaborative and 0 otherwise).
commits (p^o)	Number of commits by a given contributor [212] [214]. Gerrit encourages and enforces rules to split code changes into smaller commits, as this practice makes review quicker and easier to identify potential flaws.[33]
patchSets (p^o)	Number of patch sets submitted by each contributor. [170] [209]
devProjects (p^o)	Number of projects a contributor commits code to. [64]

related to change set productivity, and those related to contributor productivity.

Before proceeding with EFA, one needs to ascertain that the basic assumptions for

factor analysis are met. To do so, we used Bartlett's test of sphericity and the Kaiser-Meyer-Olkin (KMO) tests [216] to examine the strength between the measured variables, and how factors explain each other[34]. Bartlett's test validates the null hypothesis that the correlation matrix of the data is identical to its identity matrix I. If this is the case, then the observed variables are unrelated and the data is therefore unsuitable for EFA (in other words, one should reject this test to continue EFA). KMO tests to what extent each observed variable could be perfectly predicted by other observed variables in the dataset.

The KMO tests revealed an *average* fit of the EFA model for quality ($KMO = 0.75$) as well as for productivity measured at change-set level ($KMO = 0.73$). Bartlett's tests were significant ($p < 0.0001$) with $\chi^2 = 4.2 \times 10^{-6}$ for quality and $\chi^2 = 1.6 \times 10^{-6}$ for productivity at change-set level. However, none of the EFA tests were successful for productivity measured at contributor level (i.e.for the p^o variables in Table 4.6). Therefore, we could not carry out a multivariate analysis at that level, and instead we only conducted a univariate statistical analysis for them.

We then proceeded with EFA on the quality metrics and on the change-set level variables to determine the latent variables and their loadings, resulting in the loadings reported in Tables 4.7 and 4.8. To determine the number of factors, we used the eigenvalue rule [204]. We computed the eigenvalues of the correlation matrix for both quality and (change-set level) productivity, and chose only the factors with eigenvalue ≥ 1. We obtained three latent variables for quality and two for productivity. To determine which observed variables contributed to these latent variables, we considered the absolute value of the loadings to be relevant when ≥ 0.32 [204].

[34] https://www.analysisinn.com/post/kmo-and-bartlett-s-test-of-sphericity/

Table 4.7: The latent variables for quality and their loadings.

Observed variables	automationEffort	socialInteraction	reviewQuality
coreReviewers	**0.409**	0.103	**0.523**
juniorReviewers	0.028	**0.568**	**0.482**
reviewPriority	0.008	0.005	0.030
verified	-0.005	**0.432**	**0.514**
workFlow	**0.684**	0.077	**0.339**
gating	**0.996**	-0.011	0.000
comments	0.028	**0.765**	0.217
commenters	0.013	**0.882**	0.157
churn	0.034	0.222	0.269
changedFiles	-0.005	0.025	0.089
status	**0.989**	-0.010	-0.008
collabs	0.055	0.126	**0.349**

Table 4.8: The latent variables for productivity at change-set level and their loadings.

Observed variables	socialInteraction	reviewProductivity
duration	**0.332**	0.275
collabs	0.164	**0.373**
churn	**0.325**	0.243
comments	**0.583**	0.291
commenters	**0.977**	**0.199**
codeReviews	**0.510**	**0.542**
workflow	0.121	**0.450**

The observed variables highlighted in bold in Tables 4.7 and 4.8 contribute to the corresponding latent variable. The higher the loading, the more the observed value contributes to the latent variable. We then manually assigned a label to each latent variable on the basis of the observed variables it is composed of. For example, factor *automationEffort* in Table 4.7 captures information from variables *gating*, *status*, *workFlow* and *coreReviewers*, most of which relate to the automation effort in the CI process. Note that some of the observed variables contribute to multiple latent variables.

Confirmatory factor analysis (CFA) and structural equation modeling (SEM)

Based on our EFA measurement model, we want to confirm and analyze the relationships among the latent variables. This requires defining a CFA model and structural regression model (SEM) [217–219] in terms of the latent variables, then performing a SEM analysis on these models to determine the relationship(s) between latent variables. A typical SEM model can enable researchers to uncover multiple regression causal relationships in a single analysis among latent variables [220–222]. The left-hand side of both equations are the endogenous (i.e.dependent) variables and the right-hand sides are the exogenous (i.e.independent) variables. Therefore, endogenous variables are explained by exogenous variables.

In particular, we define multivariate regressions for quality (**F6**) and productivity (**F7**) in Equations 4.1 and 4.2 below, and both equations are data-driven. In particular, we did EFA (in two steps; first, to determine the numbers of factors via the scree test, followed by factor analysis), CFA, and SEM (with path analysis):

$$\begin{aligned} socialInteraction &\sim automationEffort \\ reviewQuality &\sim automationEffort + socialInteraction \end{aligned} \tag{4.1}$$

$$reviewProductivity \sim socialInteraction \tag{4.2}$$

Note that the relationships in equations 4.1 and 4.2 both are derived from the data during the exploratory and confirmatory Analysis (EFA and CFA), we have provided our replication package [165], in which we implement the SEM analysis as explained by [205]. First, we did EFA to determine how many factors best explain the data, we used eigenvalue rules and scree-plot test to determine best fit factors. Second, we did factor analysis to discriminate factors below an acceptable threshold of 0.32. In particular, SEM models automatically generate the possible relationships between latent and observed variables. Then, based on the factors selected during the EFA step, we obtained the relationships described in equations 4.1 and 4.2 in the two models of Quality and productivity, which are the structural parts of the models that explain the causal relationships, with the path diagram explaining the magnitude of such relationships. In sum, SEM models contain both a measurement and structural part that we fit to the data to learn about causal relationships between exogenous and endogenous variables.

This allows us to see how *reviewQuality* is explained by *automationEffort* and *socialInteraction* (Equation 4.1), and how *reviewProductivity* is explained by *socialInteraction* (Equation 4.2). These relations will enable us to validate to what extent the act of micro-collaboration impacts both *reviewQuality* and *reviewProductivity* while controlling for the other factors.

We first have to evaluate our models to either accept or reject them based on their fit. We considered the indicators proposed in the literature [203, 206, 218, 223, 223–225]. Cronbach's α [226] explains how closely related tested items are in a *group* [194], which

enables us to establish reliability and validity of the three factors for software quality (*socialInteraction*, *automationEffort* and *reviewQuality*), and the two factors for productivity at change-set level (*reviewProductivity* and *socialInteraction*). The reliability of our measurement ensures its consistency over repeated trials, and the validity indicates to what extent the individual observations rightly measure what they are supposed to measure. To accept the reliability of a model a value of $\alpha \geq 0.7$ is expected. We obtained $\alpha = 0.77$ for the quality model and $\alpha = 0.74$ for the productivity model, implying that both models passed this test. The SEM model fit index [203] tells us how well the model fits the data. For both measurements of quality and productivity our models shows good and acceptable fits (N=608,261).

While our models did not pass the SEM χ^2 index criterion ($\rho <$ threshold of 0.05), this criterion is known to be sensitive and to easily reject models with large sample sizes and minor deviations, and is failed by most of the prior work. Instead, our models did pass more robust criteria such as Root Mean Square Error [218,224] ($0.07 \leq 0.08$), Normed-Fit Index ($0.98 > 0.95$), Tucker-Lewis Index ($0.97 > 0.95$), Comparative-Fit Index ($0.98 > 0.90$), and Adjusted Goodness-of-Fit Index ($0.97 > 0.90$). Our models were accepted based on these goodness of fit indexes and show strong support for both quality and change set-level productivity.

4.6 Quantitative results

This section presents the quantitative empirical analyses that we conducted, using various kinds of statistical testing, to confirm or refute the main findings of the qualitative analysis of the interviews in Section 4.4. We would like to stress that, apart from the two SEM analyses [218, 225], our quantitative analysis methods can only infer correlation, and therefore

do not claim any causal relationship between any of the micro-collaboration related factors studied in this section.

4.6.1 How do OpenStack contributors engage in micro-collaborative coding practices? (RQ_1)

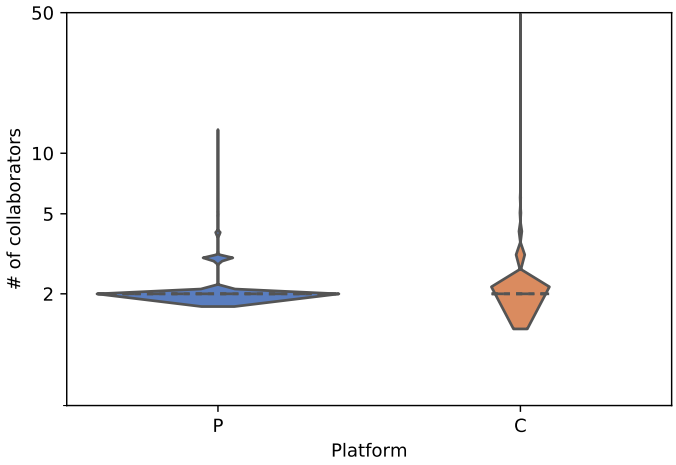
F1 reported qualitative evidence that OpenStack contributors use two main micro-collaboration mechanisms: git commit message trailers and Gerrit change sets. Since the majority of respondents indicated that the practice of git commit trailers was infrequent, this subsection studies the prevalence of these two micro-collaboration mechanisms.

> Micro-collaborative coding happens in OpenStack, but is infrequent. It is considerably more prevalent on Gerrit ($\sim$ 11% change sets and $\sim$ 27% of patch sets) than on git ($<$ 2%), although the former instances are harder to identify manually than the latter. Over three out of four collaboration instances involve only two distinct contributors, while collaboration between larger groups is scarce.

Out of all 917,970 commits in the commit dataset $\mathcal{C}$ we only found 15,801 commits (i.e.1.72%) with git commit trailers. On the other hand, in the 497,442 merged change sets belonging to the $\mathcal{P}$ dataset of 631,201 change set, we found a much higher proportion of 55,794 change sets (11.2%) that can be regarded as review-based collaboration. In terms of individual patch sets (i.e., patch versions), we found that 425,736 out of the studied 1,600,019 patch sets of the considered change sets appear in collaborative change sets, indicating a 26.61% collaboration in terms of contributions to patch sets. In contrast to the trailer-based mechanism, the Gerrit-based micro-collaboration is not explicitly tagged, hence is hard to identify manually or retrace.

Figure 4.2 presents a violin plot of the distribution of the number of collaborators in each individual git commit and change set. Most collaborations involve two distinct contributors, accounting for over 75% of all collaboration instances in either platform. We found an outlier with as much as 49 co-authors in git commits; they were all contributors to the neutron project, performing a divide-and-conquer technique to jointly implement seven sub-features[35], to support some third-party APIs. The outlier with 13 collaborators in Gerrit change sets happened in the DevStack project while working on major changes[36] that aimed to make DevStack more user-friendly to install, test and use.

Figure 4.2 Number of collaborators in collaborative git commits in $\mathcal{C}$; and in Gerrit change sets in $\mathcal{P}$.



> Both commits and change sets need to be considered to gain a comprehensive overview of micro-collaboration within OpenStack, since neither git commit trailers nor Gerrit collaboration are used consistently.

[35] https://opendev.org/openstack/neutron/commit/cd66232c2b
[36] https://review.opendev.org/c/x/group-based-policy/+/359883

Table 4.9 presents the top ten of most collaborative projects at OpenStack. They correspond to cross-project teams[37] that, combined together, account for 35% of the total collaborative activity. Cross-project teams such as Nova, Neutron, Cinder, Keystone and Horizon develop common features that other OpenStack projects consume. They typically manage OpenStack resources such as configuration and deployment services (Openstack-Ansible and Kolla). Despite the 35% magnitude of collaborative coding within these ten cross-project teams, five of them represent projects with the most commits [59]. In general, as observed by [59], collaborative coding doesn't happen frequently during OpenStack development cycles, but is used sparingly during complex features that involve multiple teams and contributors.

As micro-collaboration in OpenStack is observed in both git and Gerrit, we also investigate the overlap in micro-collaboration between both platforms. In other words, how many instances of collaboration are indicated both by git commit trailers and multiple Gerrit patch sets, and how many collaboration instances are unique to either method?

For this analysis, the dataset $C \cap P$ of Section 4.5.3 is used to compare the two platforms in an unbiased way. Out of the 442,637 aligned change identifiers, 56,202 are collaborative either on git or Gerrit and will be used for the subsequent analysis.

Table 4.10 presents the number of collaborative change identifiers on git or Gerrit. Our results indicate that 5,980 of these collaborative change identifiers exist in both $\mathcal{C}$ and $\mathcal{P}$, representing respectively 64.76% and 11.29% of retrieved micro-collaborations in each platform. This overlap indicates cases where the implicit micro-collaboration observed in Gerrit was made explicit through a commit message trailer. The remaining collaborative change identifiers, respectively 3,254 in $\mathcal{C}$ and 46,986 in $\mathcal{P}$, could only be found in one of

[37] https://docs.huihoo.com/openstack/docs.openstack.org/project-team-guide/cross-project.html

Table 4.9: Top 10 of most collaborative OpenStack projects in terms of proportion and number of collaborative change identifiers.

project	proportion (%)	# collaborative change identifiers
nova	9.3	8,778
neutron	6.3	6,007
tripleo-heat-templates	5.0	4,743
openstack-ansible	2.6	2,496
cinder	2.6	2,419
openstack-manuals	2.2	2,075
keystone	2.0	1,872
horizon	1.8	1,743
kolla	1.7	1,576
kolla-ansible	1.5	1,409
TOTAL	35.0	33,118

Table 4.10: Recognition of collaborative activity of aligned change identifiers.

		Collaborative in Gerrit	
		True	False
Collaborative in Git	True	5,980	3,254
	False	46,986	-

both datasets. Especially the latter cases represent a risk in terms of micro-collaboration visibility. Overall, within $\mathcal{C} \cup \mathcal{P}$, commits alone capture 16.43% of all collaboration activity, while change sets capture as much as 94.21%.

> The proportion of micro-collaboration change identifiers in OpenStack fluctuates around 15% from July 2015 onwards and around 19% from July 2017 onwards, while it was less than 12% in the first three years of development.

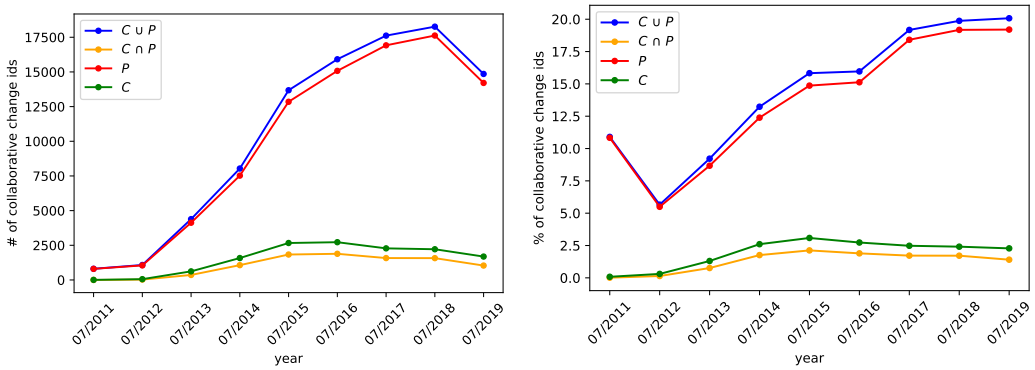
Figure 4.3 shows the evolution during 07/2011– 06/2020 of the frequency of micro-collaborative change identifiers in $\mathcal{C}$, $\mathcal{P}$, $\mathcal{C} \cup \mathcal{P}$ and $\mathcal{C} \cap \mathcal{P}$. Figure 4.3(a) reveals that the amount of micro-collaboration increased significantly for $\mathcal{P}$ and $\mathcal{C} \cup \mathcal{P}$, after which

it decreases again. Meanwhile, $\mathcal{C}$ and $\mathcal{C} \cap \mathcal{P}$ increased slowly until 06/2015, followed by a gradual decrease onwards. The drop of micro-collaboration in $\mathcal{C}$ and $\mathcal{C} \cap \mathcal{P}$ seems to coincide with an overall decrease in the number of OpenStack change identifiers, since Figure 4.3(b) reveals that, proportionally speaking, from 07/2015 onward $\mathcal{P}$ continuously increases but $\mathcal{C}$ and $\mathcal{C} \cap \mathcal{P}$ continuously decrease albeit at a much lower rate. Interestingly, the *proportion* of micro-collaboration change identifiers dropped sharply during 07/2011–06/2012 (negative slope of $\mathcal{P}$ and $\mathcal{C} \cup \mathcal{P}$), which is likely due to the explosive growth of development activity in OpenStack in that period (not shown in Figure 4.3, given the exponential growth in OpenStack development, with a yearly increase in the number of change identifiers ranging from 37% to 135% until 2019, the stable micro-collaboration ratio indicates that collaboration actually happens frequently.): 6,622 change identifiers were made during 07/2011–06/2012 compared to the 17,720 change identifiers during 07/2012–06/2013.

4.6.2 What are the benefits of micro-collaboration? (RQ_2)

This section quantitatively validates the qualitative findings of perceived benefits highlighted in gray in Table 4.2: **F4** *Collaboration improves onboarding*; **F6** *Collaboration improves software quality*; **F7** *Collaboration improves productivity*; and **F8** *Collaboration enables recognition and accountability*.

Figure 4.3 Annual evolution of collaboration frequency in OpenStack. Data points along the x-axis indicate the *start* of each yearly interval (e.g 07/2014).



(a) Absolute collaboration values (b) Relative collaboration proportion (%)

F4 Collaboration is associated with a higher likelihood of improving onboarding.

A recent study by Foundjem et al. [64] reveals that ecosystem-level onboarding at OpenStack correlates with higher retention rate, productivity and quality. Since that study indicated that micro-collaboration enables mentor-mentee interaction, here we focus on studying to what extent micro-collaboration correlates with the retention rate of newcomers.

> Participation in micro-collaboration is correlated with prolonged activity regardless of contributor seniority.

For each studied year of development (ranging from July to June), we use *survival analysis* [227, 228] to analyse the probability that contributors remain active ("survive") for a given period of time. To avoid bias, we split our dataset using a sliding window of four years that we move forward one year at a time. This results in five consecutive periods of four years, i.e., 2011–2015, 2012–2016, ... until 2015–2019. In each window, we perform a separate survival analysis considering only the contributors who had their first contributions in that window. The duration of each contributor's activity in a given window

is calculated as the time elapsed from the first contributed patch set recorded in $\mathcal{P}$ until the last contributed patch set in the full observation period (June 2020). We compare the survival curves [228] for contributor activity in each four-year window along two different dimensions:

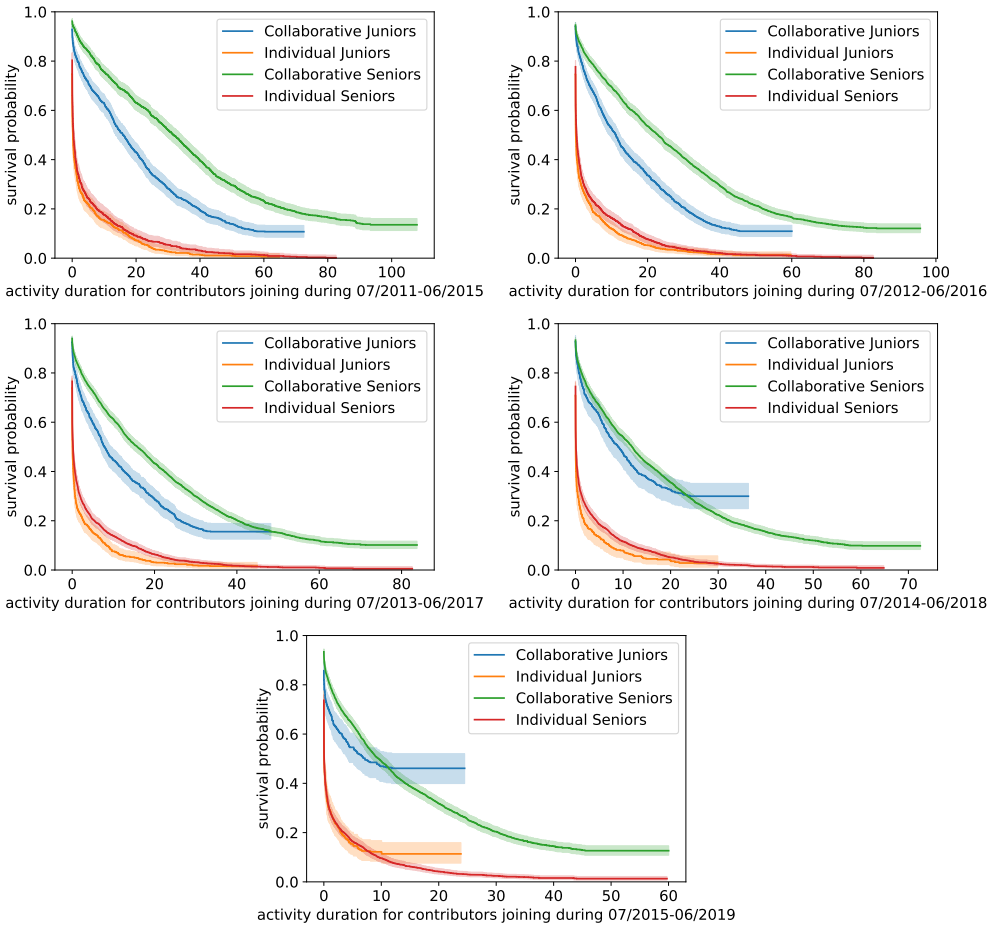
- *junior* versus *senior*: junior contributors have provided their first contribution within the last year of the considered 4-year time window; while seniors have made contributions in earlier years of that time window.
- *individual* versus *collaborative*: individual contributors were never involved in micro-collaboration in OpenStack, while collaborative ones have been active in micro-collaboration.

To clarify this, we consider a concrete example for the time window 07/2013 to 06/2017. The contributors who contributed from July 2016 until June 2017 are considered as juniors, while the ones who contributed from July 2013 until June 2016 are seniors. The sliding window technique allows to compare the longevity of junior contributors in each time window as there is no overlap between the juniors in different time windows (juniors in one time window become seniors in the next time window). The two dimensions of contributor classification yield four different survival curves per time window for all possible pairwise combinations, shown in Figure 4.4 with 99% confidence intervals.

We observe that collaborative contributors (blue/green) are active significantly longer in OpenStack than individual contributors (red/orange). This is visually confirmed by the non-overlapping confidence intervals, and statistically confirmed through log-rank tests with $p < .001$ (after Bonferroni correction to adjust for family-wise error rate) for all periods.

Note that the maximum longevity of junior contributors is decreasing in more recent time windows as the end of the full observation period is June 2020, thus limiting the

Figure 4.4 Kaplan-Meier survival curves with 99% confidence interval (shaded areas) for sliding windows of 4-year periods, for junior/senior OpenStack contributors according to their collaboration status. The activity duration is measured in months.



time they could have been active. To account for multiple comparisons of juniors in one time window participating as seniors in follow-up time windows, we tested all statistical hypotheses after adjusting p-values with Bonferroni correction; the statistical differences were confirmed with adjusted $p < 2.8e-5$.

We also observe that collaborative stakeholders have at least a 15% probability of surviving during the entire studied period, with collaborative seniors usually having a higher survival probability than juniors until June 2017. The survival probability of individual

contributors drops below 20% after approximately 12 months, decreasing until close to 0% towards the end of the studied period. Overall, our results show that micro-collaboration correlates with developer engagement in OpenStack.

F6 Collaboration is associated with a higher likelihood of improving software quality

To quantitatively validate whether micro-collaboration correlates with better software quality we performed (1) χ^2 tests on the bug-introducing likelihood of micro-collaborative vs. individual change sets, and (2) a multivariate SEM analysis on the latent variables for review quality (*reviewQuality*) identified in Table 4.7. The latter analysis validates to what extent the review process is different for micro-collaborative and individual change sets.

Table 4.11: The presence of bug inducing changes (BIC) in collaborative versus individual coding.

	Observed Count		Expected Count	
	BIC	Non BIC	BIC	Non BIC
collaborative	2,320	13,678	5,542.15	10,455.85
	(14.5%)	(85.5%)	(34.6%)	(65.4%)
individual	315,690	586,282	312,467.85	589,504.15
	(35.0%)	(65.0%)	(34.6%)	(65.4%)

$\chi^2(1, N = 917,970) = 2916.21, \quad \rho < 0.001$

> Micro-collaborative changes tend to have a significantly lower likelihood of introducing bugs than individual changes.

Using the approach of Section 4.5.4, we use a χ^2-test across the dimensions of micro-collaboration/individual coding style and BIC/non-BIC change set. Table 4.11 shows the confusion matrix with observed and expected counts. Across the studied OpenStack

projects, 14.5% of micro-collaborative code changes are bug-introducing, compared to 35% of individual code changes. The beanplot of Figure 4.5 compares the distribution of BICs in micro-collaborative versus individual changes across the studied projects. There is a high median of 7,550 BICs for individual commits, compared to a low median of 24 BICs for micro-collaborative contributors.

The χ^2-test indeed rejects the null hypothesis, signalling a significant difference in proportion of BICs between micro-collaborative and individual contributions. Cliff's delta [229, 230] confirmed a *large* effect size for this difference ($\delta = 0.782$).[38]

Table 4.12 compares the top 10 OpenStack projects in terms of their proportion of BICs for collaborative and individual changes. We can indeed observe that the 10 most bug-inducing projects that were collaborative only account for 0.18% of the total BICs, whereas the corresponding top individual projects account for about 8.46% of the total BICs.

> The SEM model indicates a significant, positive causal link between micro-collaborative changes and review quality.

In order to interpret the SEM model for change-set level quality based on Equation 4.1 in Section 4.5.5, we consider the model's path diagram (Figure 4.6). This diagram represents the latent variables (*reviewQuality*, *socialInteraction* and *automationEffort*) as circles, observed variables (e.g.*gating* and *status*) as rectangles, and SEM model loadings as solid, directed edges between latent and observed variables. In the structural model, the loadings are labeled with their path coefficient [231], which is the standardized (β) regression

[38] According to Hess and Kromery [230], we interpret effect size as negligible ($d < 0.147$), small ($0.147 \leq d < 0.33$), medium ($0.33 \leq d < 0.474$) or large ($d \geq 0.474$).

Figure 4.5 Distribution of #BICs for individual (No) and collaborative (Yes) commits.

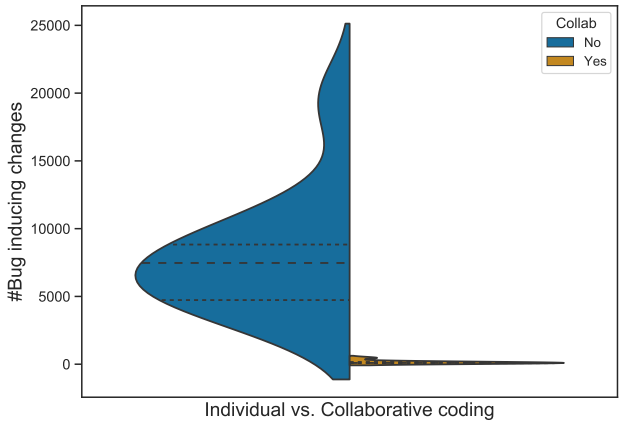


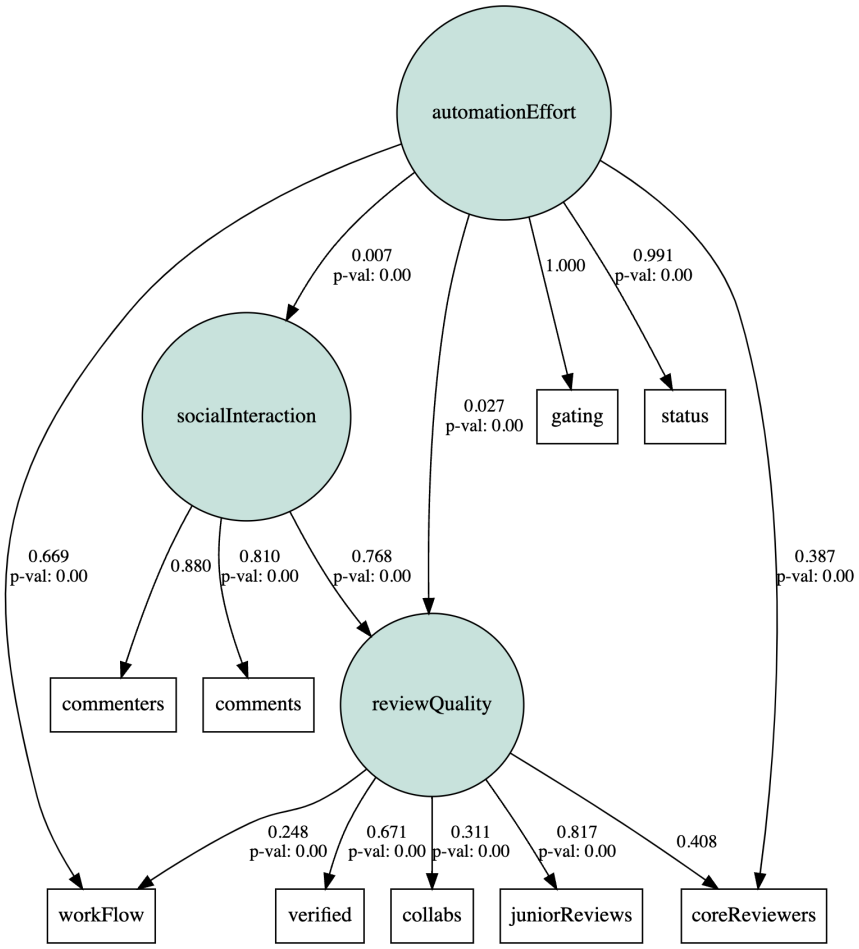
Table 4.12: Top 10 OpenStack projects most affected by bug-inducing changes (in terms of their proportion and number).

Individual coding			**Collaborative coding**		
project	***prop. (%)***	***#BIC***	***project***	***prop. (%)***	***#BIC***
/project-config	2.016	19,400	tripleo-heat-templates	0.048	467
openstack-manuals	1.152	11,089	deb-nova	0.046	443
tripleo-heat-templates	0.940	9,049	swift	0.019	190
releases	0.848	8,162	requirements	0.019	185
x/vmware-nsx	0.836	8,052	rally	0.009	96
tempest	0.715	6,885	tripleo-common	0.009	96
swift	0.495	4,764	trove	0.009	89
x/gantt	0.490	4,720	tempest	0.007	76
senlin	0.486	4,685	rally-openstack	0.007	74
placement	0.478	4,604	tacker	0.007	73
Total	8.456	81,410	Total	0.180	1,789

coefficient[39] that, similar to a traditional correlation coefficient, estimates the strength of a regression path, with a magnitude in the range $-1 \leq \beta \leq 1$. The p-value indicates statistical significance of this strength. Since the SEM model is multivariate, the path diagram shows multiple regression relationships among the latent variables. A first regression relationship

[39] https://www.statisticshowto.com/standardized-beta-coefficient/

Figure 4.6 Path diagram showing the structural analysis for the quality SEM model. Factors are shown as circles, observed variables as rectangles, and loadings as solid arrows.



is a direct path that exists between the independent (exogenous) variable *automationEffort* and the dependent (endogenous) variable *reviewQuality*, with $\beta = 0.027$, while a second relationship exists along an indirect path that flows from *automationEffort* to *socialInteraction* ($\beta = 0.007$), then to *reviewQuality* ($\beta = 0.768$).

The path diagram in Figure 4.6 shows how all paths/loadings are statistically significant at $p < 0.001$. Furthermore, the SEM model reveals causal relationships between all three latent variables, suggesting that *reviewQuality* is explained by *automationEffort* and *socialInteraction*. Notably, the interaction between *automationEffort* and *socialInteraction*

is essential for *reviewQuality*, with a combined β of 0.775 ($0.768 + 0.007$). Since the focus of this chapter is on micro-collaboration, we observe that *collab* (i.e., whether a change set was done in a micro-collaborative way) has a statistically significant relationship with *reviewQuality*, while controlling for the other two latent variables (and their observed variables). Furthermore, this relationship is positive, with more micro-collaboration correlating with higher *reviewQuality*. The SEM model estimate $\beta = 0.311$ means that a one-unit increase of *collabs* corresponds to a 0.311-unit increase of *reviewQuality*. That said, the diagram also shows how *collab* only has the fourth out of five strongest relationships with *reviewQuality*. Hence, although it is not the strongest (positive) explanation of *reviewQuality*, there is a statistically significant relationship.

F7 Collaboration is associated with a higher likelihood of improving productivity

To validate the qualitative finding that micro-collaboration improves productivity, we analyze productivity of micro-collaborative contributors in terms of the amount of technical contributions in $\mathcal{C} \cup \mathcal{P}$ (co-authored commits, created change sets, and contributed patch sets), the time required to complete such contributions, and a multivariate analysis (SEM model) on the latent factors of Table 4.8 for review productivity.

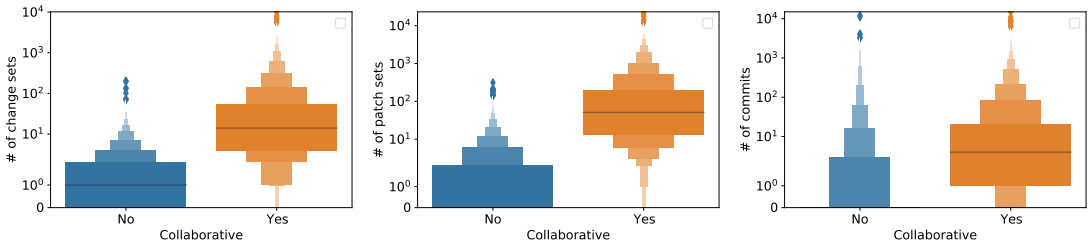
> Collaborative contributors produce more change sets, patch sets and commits. The contribution size of collaborative change sets is larger compared to individual efforts (but with small effect size).

To assess whether *micro-collaborative contributors produce more change sets, patch sets and commits*, we compared the distributions of the number of created change sets, submitted patch sets and commits between collaborative and individual contributors in

Figure 4.7. We considered contributors to be collaborative if they have had at least one collaborative activity. Furthermore, for such collaborative contributors we took into account all their contributions (including the ones they carried out individually).

Figure 4.7 suggests that there is an important difference in the number of contributions between individual and collaborative contributors. We observe that collaborative contributors create more change sets and submit patch sets in $\mathcal{P}$, and produce more commits in $\mathcal{C}$ compared to individual contributors.

Figure 4.7 Overall productivity of collaborative vs non-collaborative contributors.



(a) Created change sets (b) Submitted patch sets (c) Commits

To confirm these visual observations with statistical hypothesis tests, we carried out non-parametric Mann-Whitney U tests, as the assumption of normality is not met in neither $\mathcal{C}$ nor $\mathcal{P}$. The null hypotheses state that there is no difference between collaborative and individual contributors in the distribution of the number of created change sets, and commits. These three hypotheses are rejected ($p < 0.001$) with *large* effect size for created change sets ($\delta = -0.78$), submitted patch sets ($\delta = -0.84$) and commits ($\delta = -0.60$).

As a result, we can conclude that micro-collaboration correlates with *more* change sets, patch sets, and commits.

The experimental setting above takes into account all contributions of collaborative

contributors, including individual ones. However, individual efforts of collaborative contributors might introduce bias in the reported results. To eliminate bias, we repeated the experiment by comparing the distributions of the number of created change sets, submitted patch sets and commits between individual and collaborative contributors, but by only considering the *collaborative* activity of the latter.

The Gerrit findings persist, i.e., the two hypotheses are rejected with large effect size, where differences in Cliff's delta are observed for change sets ($\delta = -0.49$ instead of -0.78) and submitted patch sets ($\delta = -0.72$ instead of -0.91). The results for commits, however, differ, i.e., the null hypothesis is rejected, but individual contributors seem to have more commits than collaborators ($\delta = 0.40$ instead of -0.60 with a medium effect size). This is observed because the collaborative activity attributed with trailers on git is scarce (see Section 4.6.1) and therefore only a small portion of commits are considered for collaborators (1.9% of collaborators' commits are in fact collaborative based on commit trailers). This result indicates that most collaborators' activity is either individual, or collaborative without attribution.

The latter observation is interesting, since the collaborative contributors seem to spend substantial effort on creating reviews and submitting patch sets, but with few commits in git to be marked as collaborative with trailers. Although the lack of collaboration attribution with commit trailers is further investigated in Section 4.6.2, these results showcase the implication of not explicitly marking collaborative efforts, especially since collaboration tracking is one of the top improvements requested by interviewees (see Section 4.4).

Another possible factor affecting developer productivity might be the actual size of the contributions in individual and collaborative contributions. We investigate this factor with a Mann-Whitney test, with H_0 stating that there is no difference between the distribution

of churn of collaborative and individual change sets. We only use the code churn of the last patch set submitted for each change set to avoid overestimating the contribution size. The null hypothesis is rejected ($p < 0.001$), but with small effect size ($\delta = -0.27$). This result indicates that collaborative change sets consist of more churn, but the reliability of this outcome is limited. This observation is in line with OpenStack's divide-and-conquer best practice recommending to split large patches into smaller blocks[40].

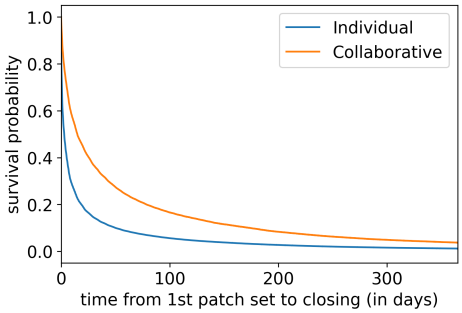
> Quantitative evidence contradicts contributor perception: collaborative change sets tend to require *more time* to finalise, and there is no evidence of a higher success rate.

According to the qualitative interviews, collaborative coding on Gerrit was perceived to allow patch sets to be merged quicker, by allowing other contributors to create additional, improved patch sets on top of existing ones, enabling them to get merged more likely and quicker into the code base. To quantitatively verify that *collaboration allows to complete code contributions faster*, we compare the time (in days) needed for getting patch sets merged, between collaborative and individual patch sets. More specifically, we computed for each change set, the number of days between the first patch set and closing of each change set (either merged or abandoned).

Figure 4.8 presents the survival curves (with 99% confidence intervals) for the collaborative patch set contributions (orange line) and individual patch set contributions (blue line), respectively. The quantitative results contradict the respondents' perception, since we observe that collaborative contributions are finalized slower, either by merging or abandoning, compared to the individual ones. A log-rank test confirms this difference with statistical significance ($p < 0.001$).

[40] https://docs.openstack.org/contributors/code-and-documentation/patch-best-practices.html

Figure 4.8 Kaplan-Meier survival curves (with 99% confidence intervals) for the time between the first and the last patch set in a change set, focusing on the first year after initial patch submission.



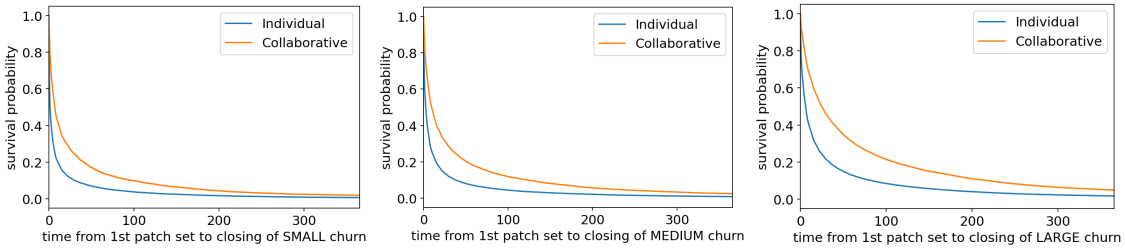
We performed a χ^2 test to verify if micro-collaborative patch sets are more likely to be *accepted* compared to individual patch sets. The null hypothesis, stating that collaboration strategy is independent from the final outcome of the change set (merged or abandoned), could not be rejected ($p = 0.078$). This suggests that the presence of collaboration is not related to the final outcome of the review (80% of individual and 87% of collaborative change sets are merged). Therefore, there is no quantitative evidence that collaboration is related to a higher success rate in merging change sets.

> The longer time taken to finalize collaborative code reviews does not depend on the amount of code churn or number of changed files.

A possible explanation as to why collaborative contributions (change sets) might take longer before they are merged might be due to the scope and size of the contributions. Collaboration could allow to produce and address larger or more complex changes that require more overall effort to produce one (larger) commit.

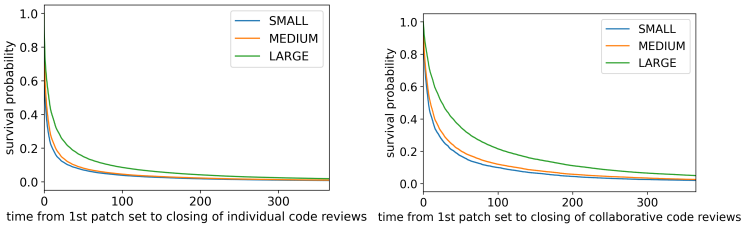
To verify this hypothesis, we categorize change sets based on the code churn (i.e.the sum of added and deleted lines of code) of the last (i.e.accepted) patch set. We grouped

Figure 4.9 Time (in days) to finalize change sets for different code churn categories.



(a) Small churn (b) Medium churn (c) Large churn

Figure 4.10 Time (in days) to finalize change sets for different churn sizes.



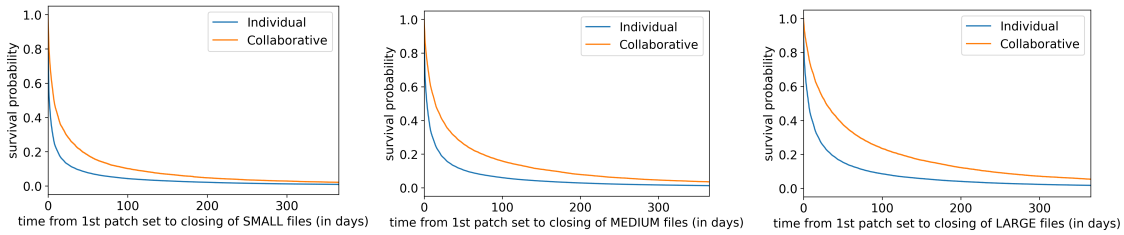
(a) Individual (b) Collaborative

each change set into three distinct churn categories using 33% and 66% of code churn as boundaries. By doing so, a *small* code churn corresponds to < 6 code lines, a *medium* churn between 6 and 37 lines, and a *large* churn ≥ 38 lines.

Figure 4.9 presents the survival curves corresponding to the time it took to finalize each change set, grouped by churn category and collaboration practice. The figure suggests that individual change sets take less time to finalize, regardless of the churn size category. This was statistically confirmed through log-rank tests ($p < 0.001$). Similarly, Figure 4.10 shows that, the larger the churn, the longer it takes to finalize the change set, independent of the individual or collaborative nature of a commit. The hypotheses were tested against the adjusted p-values using the Bonferroni correction by considering all the tests performed for churn categories (Figure 4.9) and collaboration categories (Figure 4.10). Using the adjusted alpha $p < 1.1e-4$, all null hypotheses are rejected.

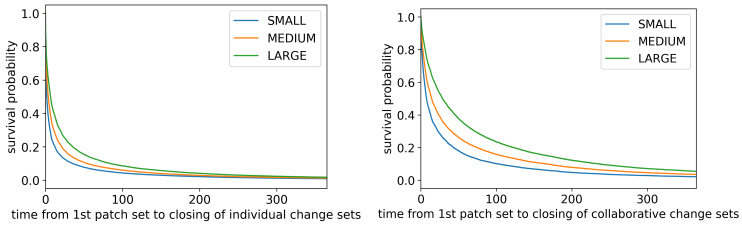
A related analysis based on the number of changed files showed similar results. We

Figure 4.11 Time (in days) to finalize change sets for different file categories.



(a) Small number of files (b) Medium number of files (c) Large number of files

Figure 4.12 Time (in days) to finalize change sets for different file sizes.



(a) Individual (b) Collaborative

created three size categories. The *small* category contained change sets touching only one file. According to the 80% quantile (compatible with the Pareto 80-20 rule), the *medium* category contained change sets touching 2 to 4 files, and the *large* category all change sets touching more than four files. Figure 4.11 presents the survival curves, per file size category, corresponding to the time it took to finalize each change set for the different collaboration practices (individual or collaborative). Figure 4.12 shows, per collaboration practice, the time it takes to finalize change sets of different file size categories.

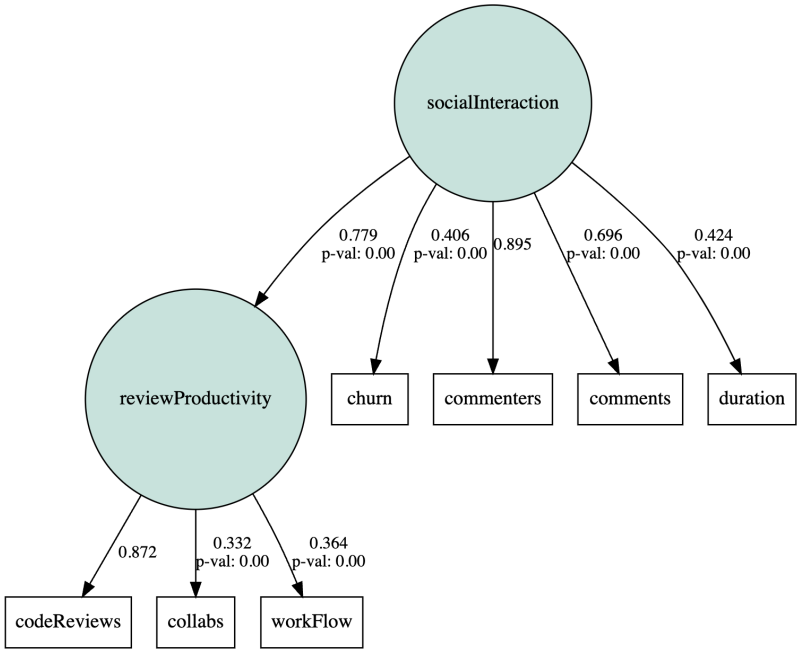
The results show that individual efforts take less time to finalize, while large change sets take longer to finalize than smaller ones. The results are statistically confirmed with log-rank tests and adjusted p-values using a Bonferroni correction (with adjusted $p < 1.1e-4$). These findings are in line with the larger size of collaborative changes. Furthermore, by definition of collaboration, collaborators need to communicate with each other, revise each other's patch sets, integrate their work, etc. While this allows collaborators to tackle more

complex changes using a divide-and-conquer strategy, it comes at the expense of taking more time to produce a lower number of larger commits than individual contributions do.

> There is a moderate correlation between the number of comments and the number of developers posting comments, and the time it takes to finalize a change set. Therefore, communication overhead only partially explains the increased time taken to finalize collaborative contributions.

Another factor that may influence the time it takes to finalize collaborative contributions, is the communication overhead that may affect collaborative activity. A Spearman correlation tests the null hypothesis that there is no association between the *time* it takes to finalize code reviews and *the number of review comments* in a change set. This is a lower-bound measurement of communication overhead as we measure the actual number of comments while finalizing any given collaborative change set, without accounting for possible multi-way communication channels. The underlying assumption is that the more messages are exchanged, the longer it tends to take to finalize a given change set. Although the null hypothesis can be rejected, the correlation is moderate ($p < 0.001$, $r_s = 0.40$). A second Spearman correlation tests the null hypothesis that there is no association between the *time* it takes to finalize code reviews and *the number of developers posting comments* in a change set; this test accounts for the individuals involved when communicating. The second test confirms the findings of the first test, since it reports a statistically significant but moderate correlation ($p < 0.001$, $r_s = 0.40$) between the two studied variables.

Figure 4.13 Path diagram showing the structural analysis for the change-set level productivity SEM model. Factors are shown as circles, observed variables as rectangles, and loadings as solid arrows.



> The SEM model indicates a significant, positive causal link between micro-collaborative changes and review productivity.

The SEM path analysis diagram in Figure 4.13 again shows how all paths are statistically significant ($\rho \leq 0.001$), this time for the productivity model (F7) showing the relationship between *reviewProductivity* and *socialInteraction* (Equation 4.2). We can see how the endogenous variable *reviewProductivity* can be explained by the exogenous variable *socialInteraction* with a strong regression path with $\beta = 0.779$, showing that the former is a complex socio-technical activity, which requires a multivariate analysis to understand how collaborative activities with the observed variables such as *collabs*, *commenters*, and *comments* relate to the latent variables *reviewProductivity* and *socialInteraction*. Therefore

both variables are positively correlated and we claim strong support for F7: Collaboration improves productivity.

When focusing on *collabs*, we again notice a statistically significant, positive relationship, indicating that micro-collaboration has a positive effect on *reviewProductivity*. Similar to before, *collabs* is not the strongest observed variable, with *codeReviews* and *workFlow* having larger loading estimates. Still, a one-unit increase of *collabs* corresponds to an increase of 0.332 units of *reviewProductivity*, hence we again conclude that micro-collaboration plays a significant role w.r.t. *reviewProductivity*.

F8 Collaboration enables recognition and accountability

The qualitative interviews with OpenStack practitioners revealed that OpenStack uses different mechanisms[41] to accredit contributions, as it enables both individual recognition and accountability (**F8**). To enable this, OpenStack's governance team has implemented a number of scripts that automatically mine the various software repositories for people's technical contributions[42].

Since collaborative contributions are only explicitly marked for commits with co-author trailers, implicit collaboration through Gerrit patch set contributions might not be captured by those scripts. We therefore quantitatively assess to what extent due credit is actually recorded.

> The low percentage ($\sim$12.9%) of git commits attributing co-authorship represents only a fraction of the actual code collaboration through trailers and patch sets.

In particular, we looked at Gerrit change sets that became merged as git commits in

[41] https://superuser.openstack.org/articles/auc-community/
[42] https://opendev.org/openstack/governance/src/branch/master/tools

OpenStack's repositories. We counted 497,442 such merged change sets, of which 55,794 were effectively micro-collaborative (see Section 4.6.1). Hence, *11.2% of all merged code that was reviewed through Gerrit was actually collaborative.* Of all these collaborative merged change sets, only 7,184 change sets (corresponding to 5,983 change identifiers) were actually reported as being collaborative by means of a git commit co-author trailer message. Hence, *only 12.9% (7,184 out of 55,794) of the merged collaborative change sets through Gerrit actually recognize that collaboration has taken place.*

> A high percentage (∼80%) of the collaborative coding effort through Gerrit goes unrecognised. In addition, less than half of these collaborative coders are actually credited for their contributions.

In order to better interpret these findings, we quantified the volume of code changes and number of contributors that were not accounted for through commit trailers. More specifically, we computed the churn (number of added and deleted lines) of all git commits corresponding to merged change sets. The total churn of all merged change sets was 12.5M lines, while the churn of all change sets that were explicitly marked as collaborative (in the commit trailer) was "only" 2.9M lines. In order words, *only 19.3% of the collaborative churn was actually recognized as being collaborative.*

In a similar vein, we quantified the number of contributors that did not get any credit for their collaboration. In total, we counted 5,953 distinct contributors having been active as collaborators in patch sets that ultimately lead to merged change sets. As an under-approximation, 3,560 (i.e.59.8%) of those contributors did not get any credit with commit trailers for their collaboration in any of the patch sets for which they were found to be collaborators. As an over-approximation, 5,613 (i.e.94.3%) of the total number of distinct

contributors got only partial recognition, in the sense that they were marked as co-author using git trailers for some, but not all of the merged change sets to which they had collaborated on Gerrit.

4.7 Discussion

4.7.1 The good

This chapter is the first case study of its kind focusing on the notion of micro-collaboration in open-source development, as well as its relation to socio-technical factors such as productivity, quality and developer retention. The recent mechanism of commit message trailers allows to explicitly mark micro-collaborations, but it is used by less than 2% of the git commits in OpenStack. Micro-collaboration through Gerrit code changes is more frequent, accounting for 11% of the code changes and increasing each year. Most of the micro-collaborations involve only two contributors.

Despite the relative infrequence of micro-collaboration, our quantitative analysis was able to validate a number of benefits and expectations expressed by the interviewed OpenStack community members. In particular, we found empirical evidence for the claims that:

- micro-collaboration correlates positively with onboarding and retention, both for junior and senior micro-collaborating participants.
- micro-collaboration correlates with a lower likelihood of bug-introducing changes and with higher review quality
- micro-collaborating contributors produce more and larger code reviews, patch sets and commits

This highlights the value and importance of micro-collaboration practices, even in a remote setting of geographically distributed development. Hence, it would we worthwhile to invest in how to make this practice scale.

4.7.2 The bad

However, we also identified disadvantages of micro-collaboration that contradict the expectations of the interviewees:

- micro-collaboration requires more time, independent of the size of the contribution
- micro-collaboration does not increase the success rate in terms of merged code changes
- the majority of micro-collaboration, i.e., the 80% happening within Gerrit, are not explicitly accounted for

The first two of these observations seem to indicate that activities involving micro-collaboration tackle more challenging problems, which was suggested by qualitative finding **F9**, but not quantitatively validated. These findings seem to reflect earlier findings on the communication overhead involved with collaboration in software engineering [232], and on the scope of code changes (e.g., contributors with larger responsibilities, such as architects, have been found to introduce more bugs due to the sheer complexity of their changes [233]). Future work should quantitatively validate these notions in the context of micro-collaboration.

The third observation is the most surprising since it implies that a non-negligible amount of micro-collaborative contributions is not explicitly accounted for, which might have real consequences, from lost recognition of developers' contributions (which might lead to a loss of "credits") all the way to loss of accountability. If, for example, a vulnerability is

reported for which only one of the possible co-authors is known, precious time could be lost trying to track down this one person as opposed to contacting the group of people responsible for it.

The root cause of this accountability problem is that most of the micro-collaborations can only be identified implicitly by checking the patch set history of commits in the code review environment. Before conducting our study, we were not even aware of such a mechanism (and its prevalence), and neither did OpenStack's community analytics tools, i.e., Stackalytics[43] and Governance[44]. Given the importance of these micro-collaborations, future studies on micro-collaboration should incorporate both kinds of data (explicit recognition through commit trailers and implicit collaboration through patch sets).

While, on the upside, OpenStack could adopt our identification heuristics of Gerrit collaboration to automatically recover these contributions in their statistics, even the best heuristics to identify hidden micro-collaboration fail if the developers involved do not bother to claim their contribution. This appears to be relatively common, as our interviews revealed, since quite some senior developers forego adding their name to commit trailers since they no longer need the OpenStack credits associated with contributions. If such micro-collaboration is not visible in the Gerrit patch set history either, one ends up with a complete loss of traceability.

This raises the question whether commit trailers –or other explicit mechanisms to tag micro-collaboration– should be enforced, or whether it suffices to heavily raise awareness about them in the developer documentation. Tool support could come at the rescue, in the form of dedicated IDE support or hooks in the Gerrit review environment to flag

[43] https://www.stackalytics.com/

[44] https://opendev.org/openstack/governance/src/branch/master/tools

likely micro-collaborations that have not been marked through commit trailers. The popular sourcetree git management tool at one point discussed such support, stating that trailers are "quite cumbersome to add [...] on every commit"[45].

However, tool support for attributing micro-collaboration is not always straightforward. For example, tools might lead to bots being attributed for automated work, or provide incentive for contribution inflation (making many small contributions instead of fewer significant ones). Nonetheless, we believe that accounting for such risks when designing micro-collaboration tools can minimize their effect. More work is needed in this direction.

On the other hand, the limited usage of the existing commit trailer mechanism may reveal underlying misconceptions of micro-collaboration. Change sets that have several contributors might lead to attribution of only a subset of those contributors. This indicates that there might not be a clear answer to what contribution(s) suffice to be attributed with commit trailers. Although every contribution is valuable, regardless of its size, it might be the case that there is a consensus on which contributions need to be attributed. These criteria of attribution, if present, should be explicitly part of the policy and of mentoring when onboarding newcomers to raise their awareness.

Hence, these aspects show the need to better support micro-collaboration with tools, but also to bring awareness of micro-collaboration attribution to the contributors. Overall, we believe that more work is needed to find the right balance of tool support and community awareness so as to enhance micro-collaboration.

4.7.3 The uncertain

Perhaps another tricky issue requiring future work is the ability to copyright the contributions that have been made through micro-collaborations, and to assign the right license to

[45] https://jira.atlassian.com/browse/SRCTREE-5347

them. On the one hand, having multiple people responsible for a given commit, possibly working for different employers, makes it more complicated to decide who to assign copyright and what license is compatible with the authors. On the other hand, in case the actual micro-collaboration is hidden, only one of the co-authors might be assigned copyright and might be consulted regarding licensing changes. Both situations highlight the importance of explicitly marking and identifying micro-collaborations.

Future work should also aim at quantitatively validating the expected relation between micro-collaboration and learning/understanding, as well as the interaction between micro- and macro-collaboration. Macro-collaboration concerns the combination of many individual developers' contributions, as is the common workflow using distributed version control systems like git to achieve a complex and long-lasting task. The benefits of such an implicit type of collaboration might differ from the ones of a more explicit form of micro-collaboration where contributors actually work *together* on shared fine-grained contributions such as code commits.

Another way to view this distinction is by considering macro-collaboration as a kind of asynchronous collaboration (where individual contributors make complementary code changes), as opposed to a more synchronous kind of micro-collaboration where multiple contributors work jointly on the same code change. It remains an open question if there is some threshold in terms of feature complexity, project size or, perhaps, contributor experience above which either of these collaboration types become more efficient.

The kind of communication channel (cf. Table 4.3) being used to give credit to micro- or macro-collaboration may also play a role in how the value and benefits of such collaboration are credited and perceived. If is therefore worthwhile to study the effect of the chosen communication channel(s) on collaboration practices.

While we focused exclusively on OpenStack, one should consider replicating our study on other software ecosystems, especially since the commit trailer micro-collaboration mechanism used could be ecosystem-specific. Furthermore, different projects might put more importance on awarding "credits" to their contributors (even though the need for accountability is universal), might have a different code base size or a larger development community.

Bogart et al.have shown that different ecosystems have different values, cultures, policies, practices and tools [234]. Hence, replicating the study on another ecosystem might lead to different findings. However, there are also some encouraging signals. In particular, the standardization attempts by GitHub and GitLab to use a common Co-Authored-By: trailer in commit messages, together with the elaborate collaboration functionalities offered by such coding platforms (notably pull request reviews and issue discussions) imply that future micro-collaborations in a majority of open-source projects would at least use similar tools and infrastructure. Apart from simplifying the replication of our analyses on such projects, this might also lead to potential convergence of micro-collaboration practices and processes in general.

4.7.4 Validation with OpenStack experts

Since our quantitative analysis validates the qualitative results obtained through interviews with the OpenStack community, we closed the loop by presenting our results to OpenStack experts. For this, the author of this thesis arranged an online meeting with nine Technical Committee (TC) members. Five of them did not take part in the original interview, so as not to introduce bias in the validation of our findings. During this meeting the study was

summarised, followed by an overview of the qualitative findings (Table 4.2) and quantitative results in terms of prevalence, advantages and disadvantages of micro-collaboration. Based on this information, the TC members were asked to validate the representativeness of the findings and to provide feedback based on their knowledge of (micro-)collaboration in the OpenStack ecosystem.

> TC members unanimously agreed that the qualitative and quantitative findings are representative and meaningful.

The interaction between regular code review comments and actual micro-collaboration was touched upon by four experts. Overall, the perceived goal of code reviews at OpenStack is still the critical evaluation of submitted patch sets, not *"taking over other's patch sets"*. Another expert made a compelling case of how code review comments, patch-set-based and commit trailer-based micro-collaboration all are practices across a common spectrum. While code review comments provide suggestions to the initial patch author, patch set-based micro-collaboration deals with fixes and changes that would take too long or be too complicated to explain to the initial author. Finally, the explicit commit trailer-based micro-collaboration would be geared towards more significant collaborations.

Going deeper into understanding what kinds of micro-collaboration are patch set-based instead of commit-trailer-based, a number of suggestions were made. Two experts stated that a *"big part of these micro-collaborations are just related to rebases, so I don't think that we can really consider them as real collaborations that are worth mentioning in the commit message"*. In other words, such patch sets would correspond to *"making minor fixups as an immediate prelude to merging"* in order to resolve merge conflicts that have popped up during code review (typically after acceptance of the patch).

In contrast, another expert assumed *"that one of the more common reasons for change takeovers is reviewers reactivating orphaned patches, or restoring abandoned ones, because the original author is no longer around or has otherwise ceased updating the change themselves"*. This expert suggested us to study how often patch set-based micro-collaborations involved patch sets by author A followed by patch sets by author B *"with no back-and-forth, and perhaps a significant delay between the two"*.

Another factor the experts encouraged us to look at in future work is the fact that *"collaboration deals with human factors like interpersonality and communication skills, and not everyone has developed those skills"*. Furthermore, the finding that more than 75% of micro-collaborations involved two collaborators might *"depend on the level of complexity, the more contributors, the more complex the problem is"*.

Finally, in terms of the overal usefulness of our study, the experts stated that *"the results are promising"*, *"may be useful for community feedback"* and *"can be valued feedback to the OpenStack community"*. Furthermore, the results confirmed suspicions about *"a low percentage of contributors using the git commit trailer. Especially for people who don't care about getting recognition for their job or community anymore"*. As such, our findings strengthened one expert's belief that *"OpenStack should reinforce this policy of using the Co-Authored-By: trailer in each commit that has been co-authored"*.

4.8 Threats to validity

We discuss the main threats that may affect the validity of our findings, following the structure recommended by Wohlin et al. [235].

Threats to internal validity concern choices and factors internal to the study that could influence the observations we made. We took a conservative approximation to quantify collaboration practices. Any contributor that was found to collaborate, regardless of the size of the contribution, was considered to be a micro-collaborator. We opted for this strategy since we believe that all collaborators should be acknowledged, even if their contributions may be very limited in size [208]. Our analysis strategy is supported by the fact that OpenStack projects are known to frequently break drown complex problems into smaller pieces to go under review [236], thus making the size of follow-up collaborative contributions small as well.

Furthermore, since some concepts like software quality and productivity concern complex interactions of many socio-technical factors, we use multivariate techniques such as Exploratory Factor Analysis (EFA), Confirmatory Factor Analysis (CFA), and Structural Equation Model (SEM) to validate qualitative findings related to "(F6) Collaborative coding im- proves software quality" and "(F7) Collaborative coding improves productivity". For other concepts, we used univariate statistical tests and analyses, which do not consider possible confounding factors, either because the preconditions of EFA/CFA/SEM were not satisfied, or because no empirical data was available related to those factors.

Threats to conclusion validity concern the degree to which the conclusions we derived from our data analysis are reasonable. The conclusions derived from our quantitative analysis are supported by statistical tests with high confidence, so they are unlikely to be affected by such threats. The conclusions derived from our qualitative analysis could have been biased by the fact that we interviewed only 16 OpenStack contributors. We are confident that this is not the case, since we continued soliciting more interviews until we reached a point of saturation in the received responses [237]. Finally, except for our SEM model

analyses, our conclusions only revealed correlations between variables that may play a role in micro-collaboration, and not causal relationships.

The findings derived from our multivariate analysis are supported by statistical analysis. First, we did preliminary tests before deciding if we could do EFA. Second, EFA helped us identify latent variables in order to reduce the risks of defining factors solely by our intuitions. Next, we confirmed our factors with CFA, and we used multiple model fitness indexes to validate our models before starting to interpret them.

Threats to external validity concern whether the results can be generalized outside the scope of this study. Given that our findings are based on an exemplar case study of a single ecosystem, namely OpenStack, our observations concern only projects within this ecosystem. These findings might not generalise to projects belonging to other ecosystems. Even if other large open source projects (e.g. Linux kernel, RedHat) adopt similar collaborative practices, the findings for such projects might still differ. Indeed, different software ecosystems have been shown to have different values, policies and practices [234].
In a similar vein, future work should study the extent to which micro-collaboration occurs within companies. While a substantial proportion of open-source projects was open-sourced by or is backed by a company, this does not imply that those companies automatically adopted the same micro-collaboration practices (even though OpenStack itself is backed and developed by a consortium of companies). Perhaps one could study open-source projects to which such companies have been contributing, and interview their contributors, to validate the findings of our study in that context.

Threats to construct validity concern the relation between the theory behind the experiment and the observed findings. A first such threat stems from the identification of

co-authorship attribution in commit message trailers. The trailer message was required to include both substrings “author” and “:”. If either of the two terms did not exist, then co-authorship attribution was not recovered, possibly leading to an underestimation. The strategy could also have lead to an overestimation since it may not have included non-authorship related trailers that still contained both substrings. To mitigate the presence of such under- or overestimations, the second author of the chapter manually investigated all distinct trailer types containing the substrings “author” and “:”. This allowed us to identify and consider additional co-authorship trailers corresponding to either typos (e.g.“Co-authorioed-by”) or alternative trailers used to indicate authorship (e.g.“Co-Authored-With”, “(Co-)Author”, “Author attribution”, “Also-Authored-By”). It also allowed us to ignore false positives (e.g.“Authorization:”).

A second construct validity threat stems from our strategy for identifying collaboration through Gerrit code reviews, which required that the collaborator must have uploaded the contributed patch set, as Gerrit’s author field might still contain information of the initial submitter if the patch set was uploaded by a follow-up contributor. This strategy may have lead us to ignore some Gerrit collaborations. To quantify the extent of this threat, we tried to identify collaboration with two additional strategies: (S1) by considering both the authors of all patch sets *and* uploaders of follow-up patch sets; and (S2) by considering only the authors of all patch sets. Strategy (S1) identifies only 286 *additional* change sets as collaborative, resulting in a risk of less than 1% of missed collaborations due to our original strategy. Strategy (S2) would only identify only 2,793 change sets in total as collaborative, since it ignores collaborators touching and uploading patches, thereby missing 96% of collaborative change sets. This meta-analysis increases our confidence in the adopted strategy to identify and extract collaborations through Gerrit.

Another construct validity threat relates to the time granularity of our quantitative analysis. We opted for an annual analysis (periods July–June as described in Section 4.5.1). We could alternatively have chosen finer-grained time intervals, aligned with OpenStack's release policy and release deadlines (twice per year according to OpenStack's release plan[46]). When repeated the temporal analyses for **F1** and **F4** by considering time intervals based on release dates, the conclusions we could derive for both findings remained the same.

Our quantitative analysis could have been threatened by including archived projects that are no longer maintained. This threat was mitigated by excluding such retired projects from our analysis. Some collaborations could have been overestimated if collaborators used multiple accounts to identify themselves on git or Gerrit. To minimize this threat, we carefully merged such cases into a single identity by relying on a partially internal ground truth data source of developer identities from OpenStack, containing a classified list of contributors and their associated email addresses. We also manually identified the presence of bots and excluded their activity from our analysis in order to avoid such automated processes being counted as attributing to inter-human collaboration. The same identity merging was also used in our analysis of bug-inducing changes to find qualitative support for the claim that collaboration improves software quality. To analyse this claim we relied on bug reports to link bugs to code commits. We encountered the technical challenge that OpenStack started migrating[47] its issue tracker from LaunchPad to Storyboard since March 2015. Since this migration process resulted in a certain amount of information loss, we mined bug reports from both issue trackers and filtered out all duplicate cases that were found on both platforms.

[46] https://releases.openstack.org

[47] `https://openstack.nimeyo.com/108658/openstack-operators-migration-to-storyboard`

4.9 Conclusion

While pair-programming has been a staple of agile development for decades, the corresponding notion of co-authoring individual patches in a remote setting, i.e., micro-collaboration, for a long time has lacked explicit coding practices. The advent of the commit trailer mechanism as a new standard for tagging commits involving micro-collaboration has enabled us to empirically study the prevalence, advantages and disadvantages of micro-collaboration in the OpenStack ecosystem, using a mixed-methods approach.

Interviews with 16 OpenStack community members identified two major mechanisms for micro-collaboration, i.e., commit trailers (the official mechanism) and Gerrit patch sets (an implicit mechanism). Micro-collaboration was said to encourage teamwork, improve onboarding/learning, software quality, productivity and dealing with software complexity. While micro-collaboration was said to enable better recognition and accountability, the interviewees did identify issues involving the tracking and awareness of joint contributions, requiring better automation.

Our subsequent quantitative study on more than 900k git commits and more than 600k Gerrit change sets surprisingly found that the implicit patch set-based micro-collaboration (in Gerrit) was responsible for 80% of the micro-collaborative changes. Overall, patch set- and commit trailer-based micro-collaboration together make up 19% of OpenStack's yearly change set volume (since 2017). We found how micro-collaboration correlated with longer retention of contributors (both junior and senior), a lower likelihood of introducing bugs, a higher review quality and a higher productivity in terms of the number of change sets, patch sets and commits produced, and the size of contributions (small effect size). In contrast, micro-collaboration did take more time, and did not have a significantly higher likelihood of getting a contribution accepted.

The main take-home messages of this chapter are (1) the relatively common occurrence of micro-collaboration in remote development, and (2) the need to consider both the explicitly visible (commit trailers in git) and implicitly hidden (patch sets in Gerrit) micro-collaboration. Future work should delve deeper into the different kinds of patch set micro-collaborations, as well as in a comparison between micro- and macro-collaboration in remote development.

Chapter 5

Release Synchronization in Software Ecosystems

In order to achieve polished SECO-level releases, it is essential to synchronize the releases of the SECO's projects. This chapter empirically studies the release synchronization activities performed by a SECO's release team, and the extent to which they are essential towards a sustainable release process for the overall SECO. This chapter was published in the journal on Empirical Software Engineering (EMSE) 17th December, 2020) [20].

5.1 Introduction

A software ecosystem comprises a set of socio-technically inter-independent software projects on top of a given technological platform [2, 11, 25]. The last decade has seen a proliferation of such software ecosystems [238–240], from programming language-related ecosystems like Maven and npm to operating system distributions like Debian and Fedora, infrastructure-related ecosystems like OpenStack and Eclipse, and mobile app ecosystems like Android and *iOS*. While each project within an ecosystem is managed locally by its project teams (or organization), the strength of an ecosystem lies in both the social interactions between these project/cross-project teams [241, 242] and their reuse of functionality via technical dependencies [72, 80, 81, 243]. Since each ecosystem project is autonomous,

it will have its release cycle and roadmap to release new versions.

While this sounds ideal for programming language ecosystems like npm or Maven, or mobile app ecosystems like Android and *iOS*, independent roadmaps make it more challenging in practice for dependent projects to plan on when to update the new release of their dependency [13,14,18]. Worse, in distribution and infrastructure ecosystems, end-users are not interested in installing one project at a time but prefer to install polished and compatible versions of the most popular projects that form the core of the ecosystem. Afterward, end-users can complement this installation with individually picked releases of other ecosystem projects. The presence of purely asynchronous releases hampers this workflow.

Given this negative impact of asynchronous releases on end-users, many software ecosystems have developed ecosystem-level release synchronization strategies. Indeed, ecosystem release synchronization enables software projects following their roadmaps (release strategies) to engage in a standard and well polished (synchronized) deliverable at the end of the ecosystem release cycle. Usually, the central release team, whose mandate is to synchronize these individual project teams' release strategies, facilitates this engagement.

For example, Figure 5.1 shows the challenges that a typical release team faces each release cycle to *synchronize* the ecosystem releases. Let us suppose that Nova, at the beginning of a given release cycle, starts developing version Dev1.0 of an artifact and releases it in version v1.0, and continuously releasing different versions throughout the release cycle. Similarly, other projects such as Glance, Cinder, or Horizon follow different release roadmaps and have different versions of their releases. These different versions of projects' intermediary releases, for example, might have dependency constraints, which are not compatible with each other during ecosystem-level release. Now, at the end of the release cycle, which version of each given project will be available (or shipped) to the end-users?

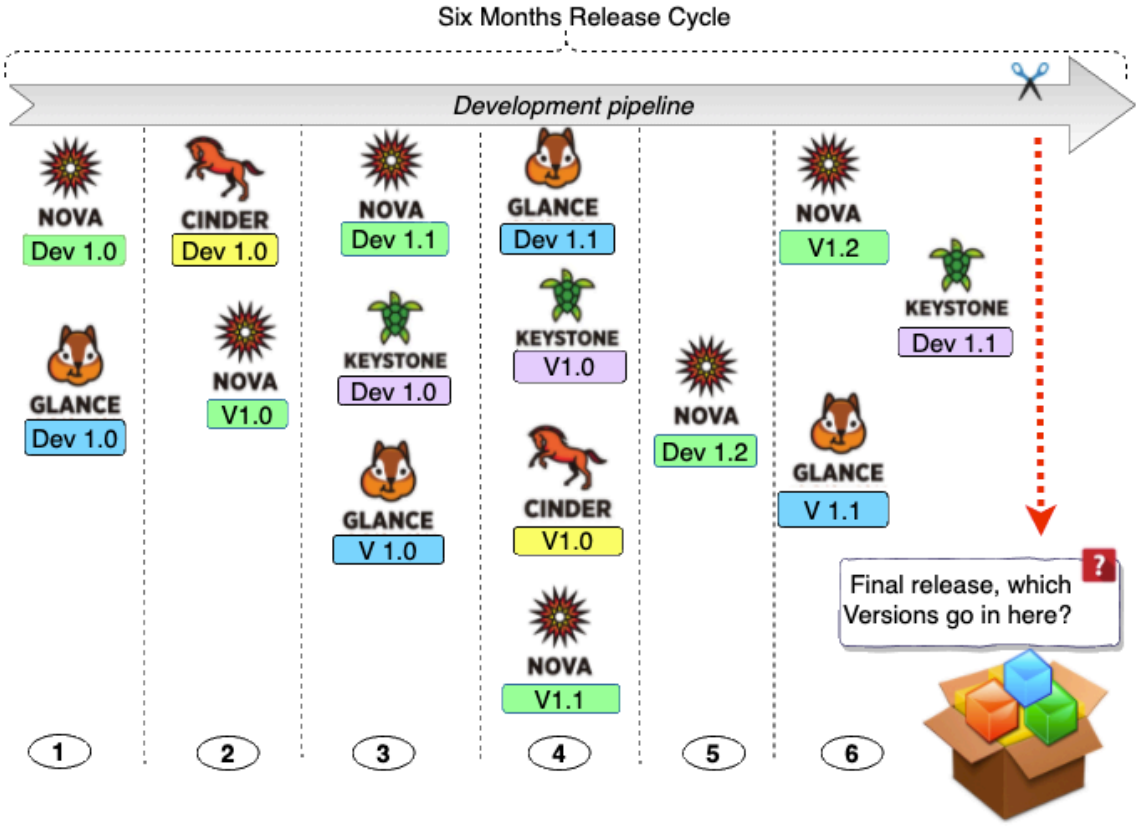
In such a complex ecosystem, the centralized release team comes in to synchronize – package versions of artifacts that are compatible – and ship a well-polished and finished product of the ecosystem to the end-users. For example, in June 2006[1], Eclipse started rolling out "simultaneous" releases of the core Eclipse platform. Each simultaneous release packaged a set of essential and compatible plugins, removing the need for users to install the core platform and then manually install their plugins of choice. Similarly, OpenStack, since its third release in 2011 (Cactus) as shown in Figure 5.3, adopted a "federated" release synchronization strategy which is managed by a **centralized release team** instead of requiring projects to discuss conflicts peer-to-peer (as Eclipse requires).

While research on release engineering has exploded during the last five years [73, 74, 244, 245], most of these works focus on the release process of individual projects, not ecosystems, especially in the context of rapid releases. Other researchers such as Adams et al. [72] studied the integration activities done by maintainers of individual packages in open source distributions between successive releases (e.g., updating to a new release or locally patching an identified bug). Similarly, Nayebi et al. [246] studied release strategies of individual mobile apps. However, none of these studies consider the synchronization of releases between multiple ecosystem projects (packages)

Our work in this chapter is motivated by the work of Teixeira et al. [79]. In their study, the authors synthesized online web resources to document the OpenStack ecosystem's release process and infrastructure and explain the different release strategies in the OpenStack ecosystem. Teixeira et al. reported the "what" about the ecosystem's rapid release process of the OpenStack ecosystem, but not the "how". For example, they found evidence of the different release strategies that co-exist in a complex ecosystem but did not determine why

[1] `shorturl.at/imFM1`

Figure 5.1 Ecosystem release dependencies across project teams following different/multiple release models. Dev X .X means a given project team at some point in the six-month cycle is developing a milestone/intermediate release. V X.X implies that a project team released a specific version of a deliverable/milestone, etc., at some within the cycle.



and how multiple release strategies exist in complex ecosystems such as OpenStack. Moreover, Teixeira et al. did not consider the release synchronization activities and challenges that the release team experienced at OpenStack. The authors encouraged future research to analyze the socio-technical activities to understand better the release process, especially the management of release notes.

Thus, in this chapter, we perform an empirical study on the OpenStack open-source ecosystem to identify the "how" and "why" of the release synchronization process at the ecosystem level, specifically, what does it take for an ecosystem release team to manage

synchronized releases across time successfully? We identify and mine the weekly Internet Relay Channel (**IRC**) communication logs of the OpenStack's release team to understand the activities that the release team performs weekly to coordinate and synchronize the ecosystem releases. IRC is an established, popular protocol for online chat and (textual) meetings between geographically distributed teams and is very commonplace in Open-source communities [247, 248]. We also classify the different release strategies supported by OpenStack, with concrete numbers of projects implementing those strategies for their deliverables and why such multiple release synchronization strategies co-exist in a complex ecosystem. Furthermore, we provide tangible evidence of ten socio-technical activities engaged by the release team during the coordination of project/cross-projects teams.

Furthermore, to validate our findings on the release synchronization activities, which we found, we interviewed eight active OpenStack senior practitioners (members of the release team and project teams) for correctness and completeness. In terms of accuracy, we were 100% correct and, however, for completeness, we missed one activity.

The main contributions of this research are as follows:

- **Qualitative analysis** of 52-weekly IRC online meeting logs, two online release plan documents, and two online release tracking documents spread across two OpenStack releases.

- **Identification** of 10 major activities involved in synchronizing releases between ecosystem projects/cross-project.

- **Validation** of the activities with four release team members and four OpenStack project members.

- **An empirical evidence** of why multiple release strategies co-exist within a complex

ecosystem.

- A **generalization of our findings** on two major open-source communities.
- The **implications** of our findings to academics and practitioners.

5.2 Methodology

5.2.1 Subject system selection

To empirically study release synchronization in software ecosystems, we looked for an ecosystem with the following characteristics:

- sufficient history of release synchronization
- well-documented release process
- open-source, allowing access to any data related to the actual releases as well as the process used
- manage multiple release strategies co-existing during release cycles
- archived communications about the majority of the release synchronization activities and plans

We carefully investigated the online documentation and archived release history of several open-source communities such as GNOME, Apache (ASF), Eclipse, OpenStack, Linux kernel, and the Kubernetes community. Out of the possible ecosystems satisfying the five criteria, we eventually selected the OpenStack ecosystem[2]. First of all, OpenStack is an

[2] https://www.openstack.org

open-source software ecosystem (under the Apache 2.0 license) for cloud computing developed by NASA and Rackspace in 2010. Besides, the author of this thesis is an OpenStack foundation member, making access to prominent individuals for interviews much easier

OpenStack, according to Forbes, is the "de facto standard for open-source based private clouds" [249], and is built on a principle of four opens: **Open** Source, **Open** Design, **Open** Development, and **Open** Community. Since its first ecosystem release, Austin[3], which consisted of only two projects, Nova and Cinder, the OpenStack ecosystem has seen a steady growth and now comprises of 759[4] projects (derived from 63 core projects)[5] [6] and involved 693 different companies. The ecosystem has over 20M lines of code contributed by over 100k community members, including volunteers, across 200 countries.

5.2.2 Communication channel selection

Among the different communication channels that exist in open source ecosystems and developers communities, such as the mailing list, IRC, Slack, Matrix, etc., OpenStack release team uses the IRC channel to discuss and coordinate projects/cross-project teams.

Specific mailing lists (openstack-discuss, release-announce and release-job-failures) and IRC channels (#openstack-release) are used by the release team. **openstack-discuss** is used for discussions related to OpenStack users and developers community, **release-announce** to announce OpenStack releases to the community, and **release-job-failures** to inform the release team about build failures in Zull (OpenStack's CI/CD pipeline).

[3] the release naming convention at OpenStack follows an alphabetical order with each release's first letter. For example, **A**ustin, **B**exar, **C**actus, . . . , and **V**ictoria (current ecosystem release)

[4] `shorturl.at/sADJ7`

[5] `shorturl.at/moNQ6`

[6] For example, Nova is a core project, while *nova – powervm* is a project (module), specifically, a subproject of Nova with a separate development team.

All the discussions in these mailing lists related to release coordination and management are discussed in the weekly IRC meeting (on **#openstack-release**), with links to the message (threads). Similarly, any source of information, such as URLs (links), Wiki page, open review, or issues, is referenced in the IRC channel and is useful to coordinate projects/cross-project teams. Thus, members can follow those links and discuss the content in the IRC weekly release team meeting. For example, during a weekly meeting held on June 5th, 2015[7], during the Liberty development cycle, release team members were discussing a cross-project checklist for the Liberty milestone-1 and how to organize office hours. A release team member suggested that "maybe I should send an email to a list that will serve as a reminder that those things exist". The email was sent to cross-project teams and reference links to a Wiki page mentioned in the IRC channel that points to the announcement in the email.

In another example, discussions on the IRC channel on March 16th, 2018[8], during the Rocky development cycle, shows the conversation (and links to external URLs) among members of the release team on topics such as the release management onboarding process, a conceptual design for the release pipeline proposed by a team member, and many more.

We can see that the release team's IRC channel captures and logs essential information that facilitates release coordination. Therefore, the release team IRC-chat logs with all its content (including external links) is sufficient to understand the release management process at OpenStack.

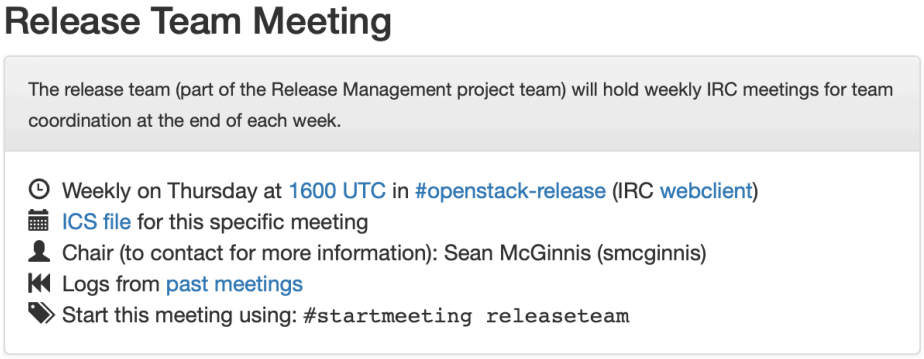
Before the Mitaka OpenStack release, the release team used the #openstack-relmgr-office IRC channel to sync up with Project Team Leads (PTLs) and project team liaisons during the weekly project meeting and during release team office hours. However, this

[7] releaseteam/2015/releaseteam.2015-06-05-13.01.log.html
[8] releaseteam/2018/releaseteam.2018-03-16-15.01.log.html

IRC channel became **#openstack-release**, the renaming ensured that all subscribers to the previous IRC channel had a 'bouncer' configured to continue communication uninterrupted in the new release channel (#openstack-release).

Introduce the 'openstack/release' repository, refining release models, and announcing measures to automate the release process.

Figure 5.2 The release team mainly uses the IRC channel to coordinate weekly meetings.(http://eavesdrop.openstack.org/#Release_Team_Meeting), all past meeting logs and any change of the meeting time/date is available on the website, and publicly accessible.

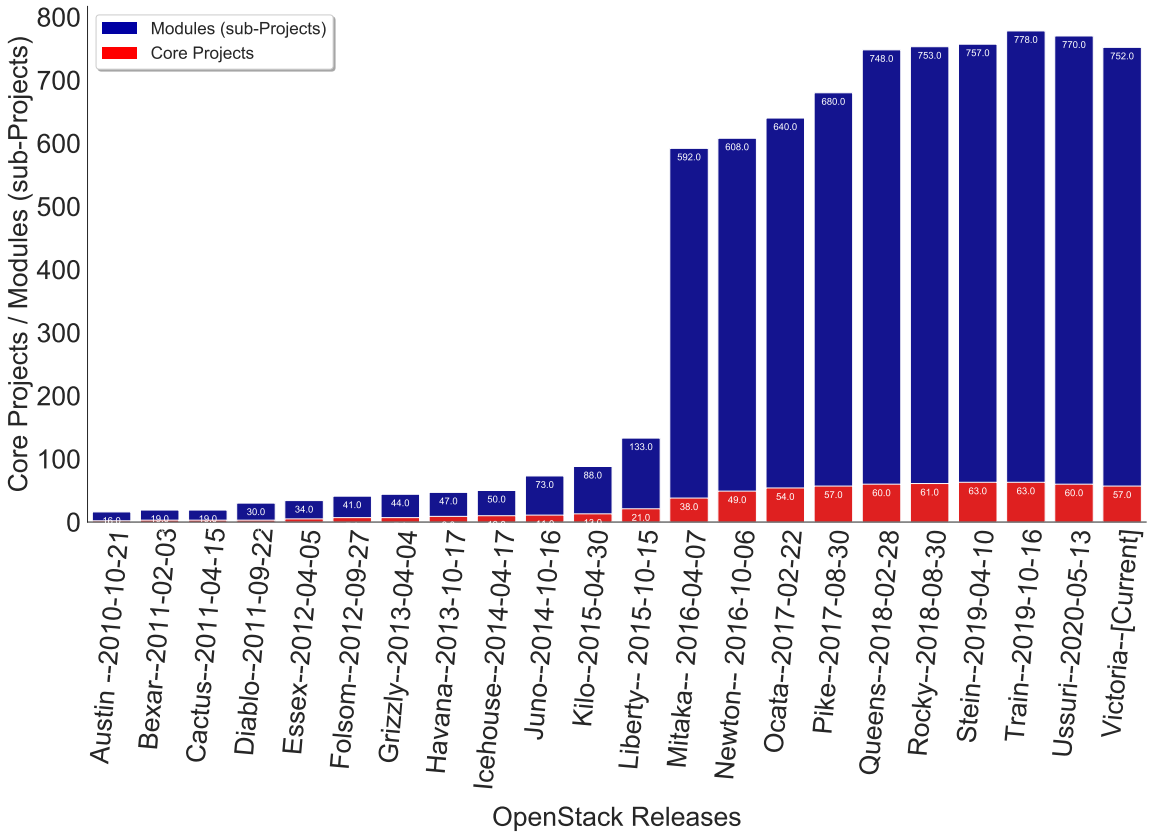


5.2.3 Data extraction from IRC logs and etherpad

Given that the archives of IRC meeting notes of the release team[9] (see example in Figure 5.2) are only available from June 2015, we decide to study the activities of the release team within that period. We select two releases: Mitaka, the first release with complete meeting notes (from 10/2015 — 4/2016), and Queens, the most recent release at the time of this study with complete meeting notes (from 09/2017 — 03/2018). We considered several factors to determine the choice for an older and a more recent release. First of all, it enabled us to compare differences in activity distribution across time. Moreover, from Figure 5.3, we observe that the number of OpenStack core projects and associated projects (modules) exploded from the Mitaka release onward and started stabilizing by the Queens

[9] shorturl.at/jCEGI

Figure 5.3 OpenStack releases over time with different release cycles, 3-months cycle starting from Austin to Cactus and 6-months cycle starting from Diablo to current release.



release.

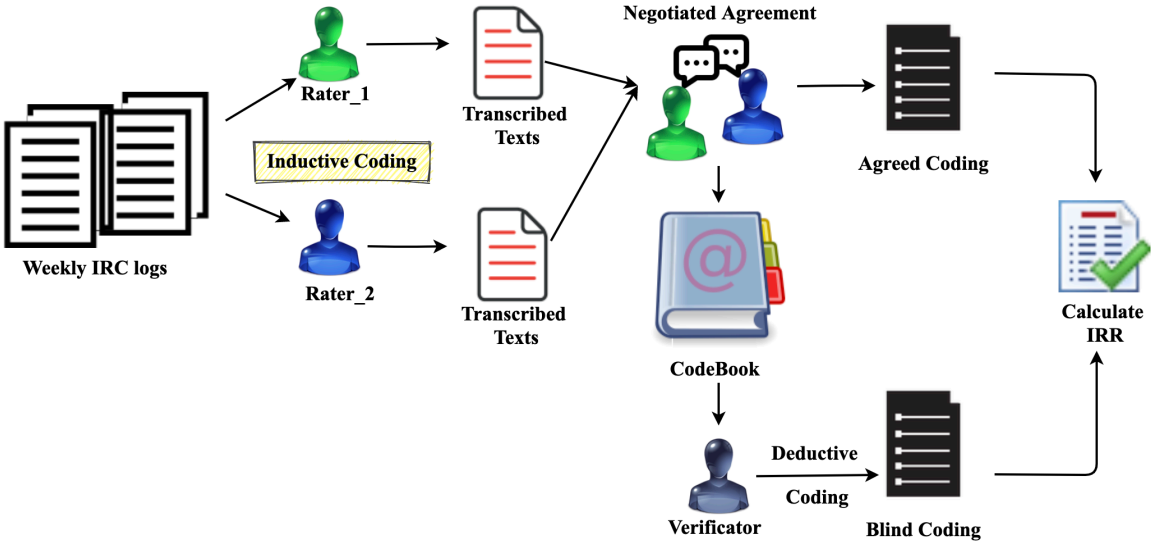
Since each release follows a six-month cycle (26 weeks), we downloaded a total of 52 weekly IRC logs for the two studied releases. The logs for each hour-long weekly meeting is available in textual format (*.txt*, *.html*, and *.log*). Each file is named based on proximity to the next release and discusses the progress towards realizing the release plan. Since the OpenStack release cycle has 26 weeks of release team meetings on average, the meeting names range from R-25 (first meeting of the release-cycle), . . . , R-1, down to R+0, which is the release week.

Besides, we also downloaded the Etherpad documents containing the release plan's status and tracking at the time of each IRC meeting. The release plan includes the ongoing

and open issues for the upcoming release, and it ultimately (during meeting R+0) is used to check whether the release has implemented all essential requirements. The tracking Etherpad is available online[10]. Based on each meeting's date and name, we downloaded the release plan at the time of each meeting.

5.2.4 Qualitative analysis of IRC logs and etherpad

Figure 5.4 We use inductive coding to build the codebook and deductive coding to compute the IRR using Cohen's Kappa for categorical data.



To identify the release synchronization activities performed by the release team, we use two rounds of open coding (inductive and deductive coding), as shown in Figure 5.4. Open coding aims at assigning relevant labels (tags) to chunks of texts (either at the phrase, sentence, or paragraph levels) within the weekly meeting logs and Etherpad documents [250, 251]. In qualitative analysis, these labels or tags are commonly called "codes;" (coding).

For example, in Figure 5.5, we show three different tags (Label #1, Label #2, and Label #3) that we placed on the highlighted text area. Next, we assigned labels to a code category

[10] shorturl.at/kzEI7

(a code category represent a group of similar themes) to the bottom right. Initially, we assign these codes to a low-level (more specific) code category. Then as the coding continues, similar emerging low-level categories are clustered into a common theme, which we abstract as high-level code categories.

Inductive coding. We performed two rounds of inductive coding. In these rounds, we aimed to generate new themes emerging from the data (IRC logs) to build a codebook [171, 185]. Both authors perform this round to calibrate the open coding to obtain high inter-rater reliability [114–116].

In particular, for the first round of inductive coding, both authors independently coded 15% of the weekly meetings [252–254] (the first eight of the Mitaka release) to identify different responsibilities of the release team members. For each discussion and agenda point, we reflected on the activity related to, for example, reviewing fixes for a showstopper bug or discussing the design of a new release tool. We assigned a code for each discovered activity and then annotated the corresponding discussion or agenda point with this code. During this process, we identified and recorded the hierarchical relations between codes. We used the online Dedoose [255] tool to annotate the logs and manage the codes, as shown in Figure 5.5.

After completing the first round of inductive coding, both authors discussed the resulting codes. The negotiated agreement aims to check for each annotated log excerpt whether its associated code makes sense, merging related codes into one or sub-code of a higher-level code. Once both authors consolidated their coding results, they both decided on the high-level code categories.

The author of this thesis continued the process and performed the second round of

inductive coding on the remaining 85% of the documents. Afterward, both the first and second authors of chapter 5 of this thesis deliberated in the codes that emerge during these inductive rounds.

Eventually, a final abstraction of nine high-level codes and their low-level codes emerged, each representing a significant release synchronization activity. The children of each high-level activity correspond to different tasks associated with it. Furthermore, the excerpts related to each code provide illustrations of each activity and responsibility.

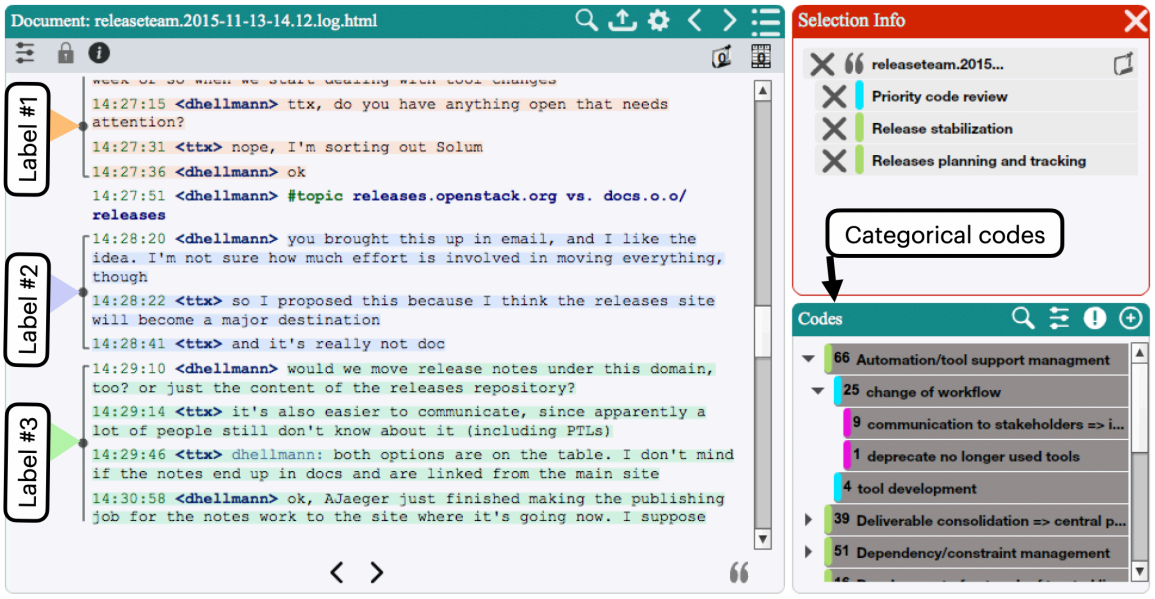
Deductive coding. Furthermore, to triangulate the results obtained with the IRC weekly logs of the two OpenStack releases, we asked an external researcher to independently apply deductive coding on all the weekly-meeting logs of the Ocata release between Mitaka and Queens.

This deductive coding happened in three iterations. The external researcher found some additional 'stand-alone' low-level code in the first iteration. However, after deliberation, both the external researcher and the author of the thesis agreed that the low-level code fits under the existing high-level code. As the coding continues (the second iteration), the external researcher applies more labels from the codebook to the text deductively. After submitted the blind coding file, both the first author of this thesis and the external researcher examined and agreed that there was no high-level code. However, we classified some codes under different high-level categories. Thus the calculated IRR gives a score of κ = 0.86. At the end of the deductive coding, the external researcher improved the labeling and reorganized the code. Both the external researcher and the author of this thesis agreed on the new configuration, which gave a perfect IRR score of $\kappa = 1.0$ [115].

Finally, we rearranged the existing low-level codes in the hierarchy, as shown in the

Affinity diagram in Figure 5.6. Based on the activities' hierarchy and the corresponding IRC chat excerpts, we identified dependencies between activities. These are relations according to which a given activity precedes another activity in time (sequential dependency) or makes another activity more natural to perform (support dependency). Our full coding results are available at our replication package online[11].

Figure 5.5 Excerpt of a weekly meeting IRC log with labels (tags) applied on three sections/paragraphs of the text, and code hierarchy shown on the right.



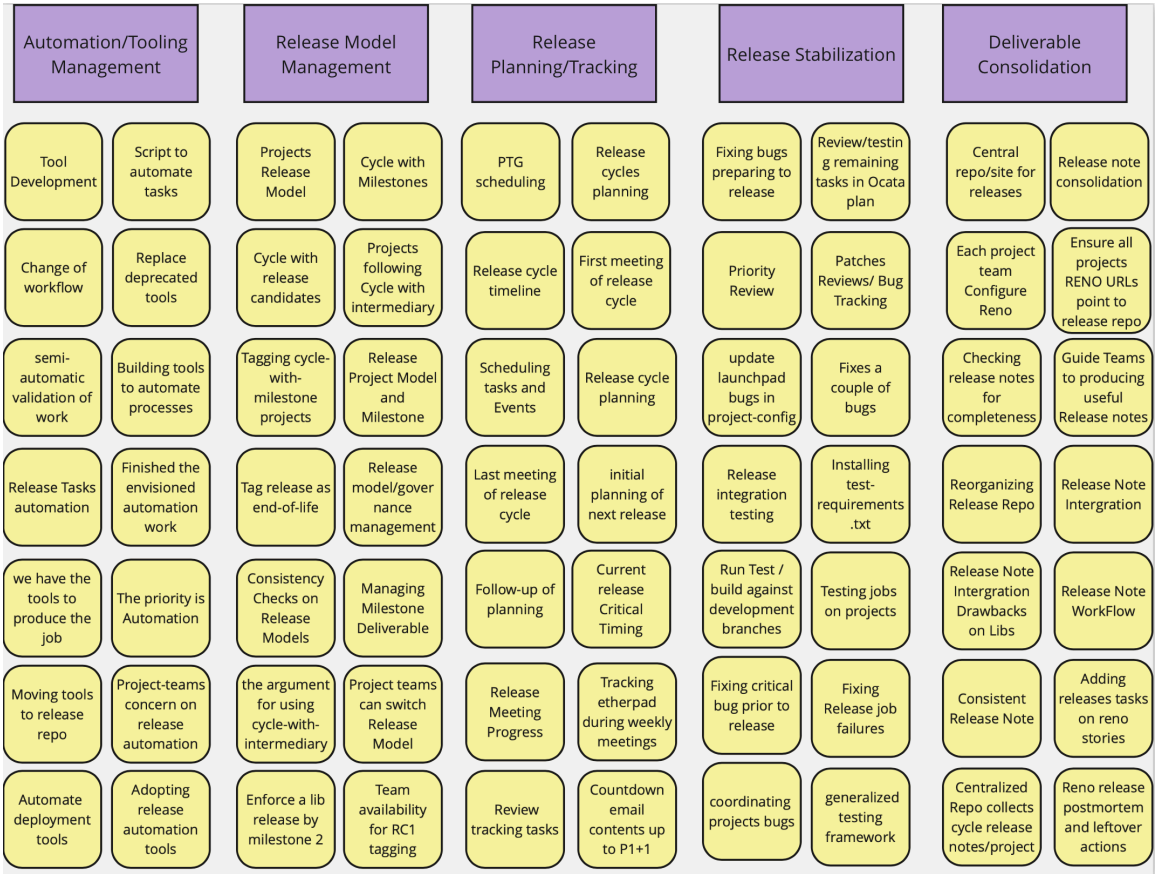
5.2.5 Preliminary analysis of the catalogue of activities

Conjecture 1

The required effort by release team coordination relates to the distribution/flow of release synchronization activities, which interact with each other to produce a synchronized release.

[11] https://doi.org/10.5281/zenodo.4304682

Figure 5.6 Affinity diagram showing five of the nine high-level abstractions of the IRC weekly chat logs. Complete diagram is available online

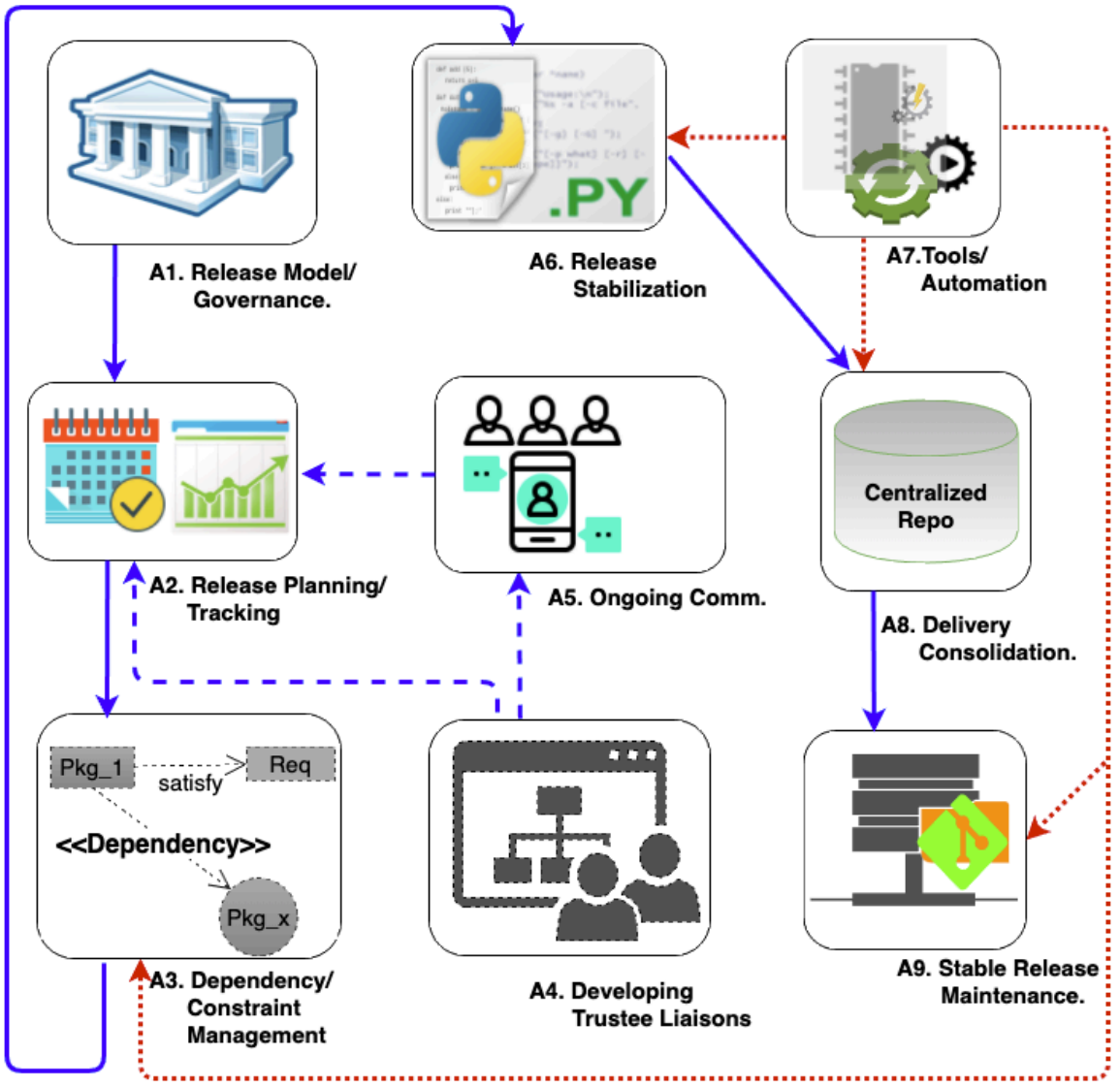


https://doi.org/10.5281/zenodo.4304682

Table 5.1: Overview of the release synchronization activities and their distributions across the Mitaka, Ocata, and Queens releases. Data for Cycle Highlights Management (A10) was not available (N/A) in our coding results since A10 is an activity that we identified after the studied Queens release.

ID	Release synchronization activity	%Mitaka	%Ocata	%Queens
A1	Release Model Management	48	46	44
A2	Release Planning and Tracking	100	100	100
A3	Dependency Management	84	70	72
A4	Establishing a Network of Trusted Liaisons	36	34	40
A5	Establishing Communication Channels	48	40	40
A6	Release Stabilization	52	53	56
A7	Tools/Automation Management	80	83	84
A8	Deliverable Consolidation	76	75	80
A9	Stable Release Maintenance	40	49	48
A10	Cycle Highlights Management	N/A	N/A	N/A

Figure 5.7 High-level dependency among release activities. Regular lines (Blue) indicate sequential activities, while the dashed lines (Blue) show social activities among A2, A4, and A5, meanwhile, the dashed lines (Red) show activities that are supported by Tools/Automation (A7)



Observation: Conjecture 1

Figure 5.7 provides an overview of the identified release synchronization activities, as well as their high-level dependencies. Table 5.1 compares the distribution of the prevalence of the identified activities between the Mitaka, Ocata, and Queens releases of OpenStack. For example, the Release Model Management (A1) activity was discussed in 48% of the Mitaka, 46% of the Ocata, and 44% of the Queens weekly meetings. Release Planning and Tracking (A2), Dependency Management (A3), Automation Management (A7), and Deliverable Consolidation (A8) are the four most discussed activities across the weekly meetings. Despite some small fluctuations in percentages, the distribution of activities is similar in the three releases. Moreover, the volume of activities such as Release Model Management (A1) and Stable Release Maintenance (A9) is higher either initially or towards the release cycle. For example, A1 happens more at the beginning and A9 towards the end. Only A2 maintains a harmonious flow of activities in the Mitaka, Ocata, and Queens releases. Instead, A3, A5 (Establishing Communication Channels), and A9 seem to have significant differences in prevalence: Queens has less A3, A5, and more A9, Ocata also has less A5, A5, and more A9, while Mitaka has more A3, A5, and less A9. ***Therefore, when the release team applies more effort on A3 and A5, A9 will require less effort to synchronize a given release.***

5.2.6 Understanding the OpenStack release strategies

To better understand the different release models (strategies) supported by OpenStack, we cloned the release team's Git repository[12]. We explored the various deliverables within the

[12] shorturl.at/eiGKY

master branch of the release repository "releases/deliverables[13]."

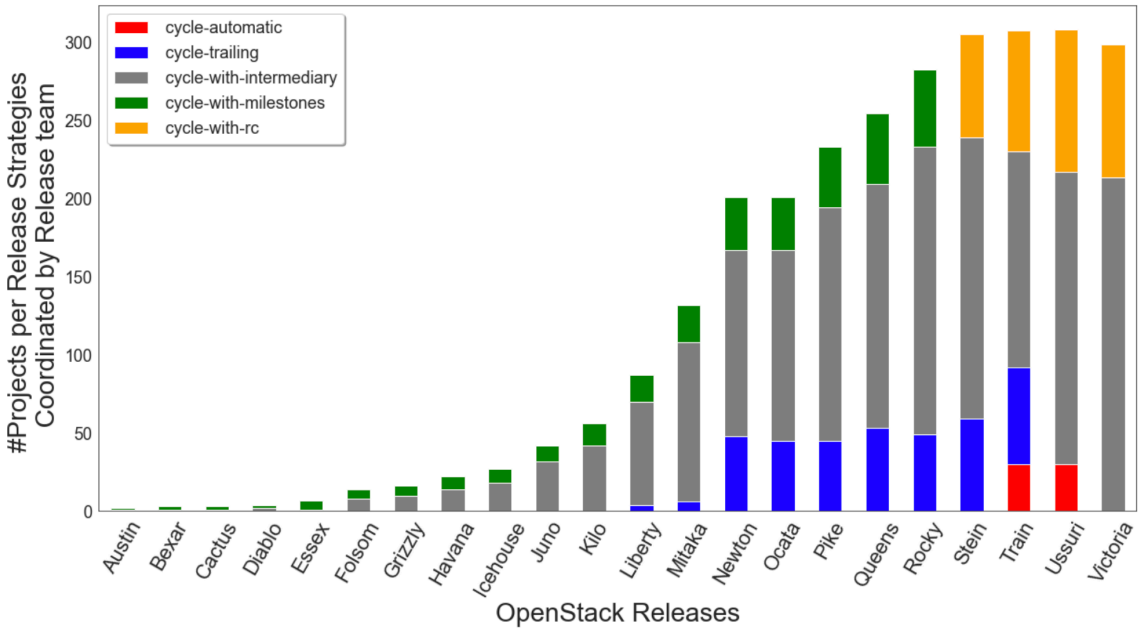
The release repository contains all the releases from Austin to the current development release (Victoria). We parsed the *.yaml* file of all the projects found in each release repository and extracted the 'release-model' type that each project uses for a given release. Based on this information, we show a visualization of all the different release models used within the projects of each OpenStack release in Figure 5.8. *"Cycle-with-intermediary"* and *"Cycle-with-rc"* are the two most common release models, used by 1,746 (58.97%) and 319 (10.77%) OpenStack project/cross-project teams. We found evidence of three legacy release models (*"Cycle-with-milestones"*, *"Cycle-trailing"* and *"Cycle-automatic"*) which have been replaced by either the *"Cycle-with-intermediary"* or *"Cycle-with-rc"* models in recent releases. The reader should note that we omit three strategies that are not managed by the release team from Figure 5.8: Untagged (52 projects), Abandoned (19 projects), and Independent (86 projects).

Release Strategies (#Projects managed by Strategy)

1. Cycle-with-intermediary (**1746**)
2. Cycle-with-rc (**319**)
3. Cycle-with-milestones (**308**)
4. Cycle-trailing (**371**)
5. Cycle-automatic (**60**)
6. Untagged (**52**) ✗
7. Abandoned (**19**) ✗
8. Independent (**86**) ✗

[13] https://opendev.org/openstack/releases/src/branch/master/deliverables

Figure 5.8 Release team manages multiple release strategies per release cycle from Austin, to the current development release cycle; Victoria



5.2.7 Validation using interviews

During the OpenStack summit/PTG in Berlin, Germany (November 13–18, 2018), the author of this thesis conducted an interview study with eight practitioners (four release team members and four project team members) from the OpenStack ecosystem. We selected the participants for the interview based on their longevity in the OpenStack ecosystem. Participants must have actively participated in at least four major OpenStack releases, including this study's three releases (Mitaka, Ocata, and Queens).

We present the demography of the interviewed participants in Table 5.2, and for confidentiality, we use mnemonics to refer the interviewees. Among the four project team members, three were female, whereas all four release team members were male. The release team has only one female member. However, she didn't meet the criteria for inclusion. We observed that the release team members have similar characteristics. Thus, we build

a persona for the release team at OpenStack and explain the skill-sets in more detail in section 5.4. The average experience for a release team member and project team member is eight years and seven years. Each member gained his or her expertise in more than one OpenStack project team, even though some release team members got experience with other ecosystems before joining OpenStack (e.g., in the Linux kernel, etc.).

Table 5.2: Demographics of interviewed OpenStack experts. The experts either belong to the release or project teams and were **active** during the studied period.

ID	region	Gender	Hired	Team	Active	#years
RP1	Americas	♂	Yes	Release	Yes	Six
RP2	Europe	♂	Yes	Release	Yes	Nine
RP3	Asia	♂	Yes	Release	Yes	Nine
RP4	Europe	♂	Yes	Release	Yes	Eight
PP1	Europe	♂	Yes	Swift	Yes	Ten
PP2	Americas	♀	Yes	Ironic	Yes	Seven
PP3	Americas	♀	Yes	Manila	Yes	Six
PP4	Asia	♀	Yes	Nova	Yes	Six

We performed a semi-structured interview with each participant, lasting about 40 minutes. The author of this thesis asked the interviewees for activities they deemed essential in the daily work of the release team and then discussed in more detail those activities of Figure 5.7 that the interviewees did not mention. For each activity, we asked the interviewees for their feedback and experience. In terms of correctness, all interviewees anonymously agreed that the nine identified activities were correct. In terms of completeness, one additional activity, Cycle Highlight Management (A10), which we did not identify during our qualitative analysis, was brought to our attention by the participants. This activity was introduced during the next release after Queens and explained why the IRC logs that we studied did not provide evidence for this activity. In Section 5.3, we discuss Cycle highlight Management activity (A10.) in detail.

5.3 Catalogue of release synchronization activities

The catalog of release synchronization activities presented in this section covers the release activities performed by the OpenStack release team, as identified from their weekly meeting logs. To separate fundamental (OpenStack-independent) principles from OpenStack-specific details, we document the activities using the following structured format:

What: Brief outline of the goal of the activity.

Why: Short description of the rationale behind the activity.

How: The major tasks involved in the activity based on the OpenStack observations.

Examples: Illustration (s) of discussions about this activity in the context of OpenStack; a complete list of all examples identified in the log files, as well as all the codes that we tagged, are available online[14].

Expert: Insights and feedback provided by the interviewed OpenStack experts.

A1.Release model (Strategy) Management

What: Each project that wants its releases integrated into the ecosystem release should select the central release team's desired way to coordinate this integration. Once selected, the project accepts to honor the responsibilities and rights related to the chosen release model, while the central release team pledges to guide the projects according to the selected model.

Why: Given that each project has its release road map and (possibly) dates, some projects might prefer to keep their course, and only at set times (determined by activity *A*2) synchronize with the release team (release-based management model). Other projects might prefer closer follow-up through predetermined deadlines of milestones and final releases

[14] https://doi.org/10.5281/zenodo.4304682

(time-based management model). Hence, an ecosystem release team should offer different release models that the project teams can choose. Once chosen, the model acts as a contract for the release interaction between the project and the release team.

Having a complex ecosystem with multiple release models is a non-trivial problem to understand, even among ecosystem community members. Release team members often have to explain to community members why different release models exist, how they differ from each other, and which one to use in what context[15].

How: The choice of a release model is not immutable. If a project wishes to change its development model in a subsequent release cycle, they must inform the release team, which is in charge of tracking and enforcing the model.

> OpenStack— *All official OpenStack projects' deliverables should go through the Release Management team to produce releases*[16].

Ecosystem projects typically choose a release model based on their near future challenges and goals. For example, time-based release models like OpenStack's "cycle-with-rc" (formerly: "cycle-with-milestones") require a project to deliver a specified number of milestones or release candidates (RC) within an *agreed upon schedule* established by the release team at the start of the cycle. The last milestone coincides with the project's feature freeze [256], after which the release team cannot add any new features in the upcoming ecosystem release before the stabilization activity (fixing critical bugs before the upcoming release). Figure 5.9 provides a graphical overview of this release model.

On the other hand, feature-based models such as OpenStack's "*cycle-with-intermediary* releases" (see Figure 5.10) are more flexible since they allow the projects to make their

[15] shorturl.at/ckmDH
[16] shorturl.at/jrFXZ

formal releases as they see fit, without any imposed milestone/release candidate deadlines nor feature freeze. Instead, the projects should ensure that their current internal release in the last month of the ecosystem's release cycle is sufficiently stable to be included as the project's official release. For example, in Figure 5.10, the blue tags in the current release cycle indicate that 1.2.3 is the final release of the project for the current release. Any newer feature proposed after 1.2.2 will be carried forward to the next release cycle, and, instead, the project is encouraged to focus on making bug fixes for 1.2.2.

Time-based models are typically preferred by projects that aim at doing fewer releases (one-release-per-cycle), especially most core projects that are reaching maturity [257]. Meanwhile, ***feature-based models are popular with projects (modules) that aim at doing several intermediary releases before the final ecosystem release.***

We observed that all OpenStack libraries opt into the feature-based model since it strongly encourages frequent releases, even without rigid rules, and invites a more collaborative interaction with the release team instead of the more policing interaction of the time-based model. OpenStack's current time-based model is already a watered-down version of the previous "cycle-with-milestones" model, which deprecated since September 2018[17], and had two additional milestones (see the greyed-out tags in Figure 5.9). Overall, the release-based model is the most popular in OpenStack.

Examples: `Missing deadlines.` A recurrent topic discussed by the release team is their frustration with projects that fail to meet the deadlines set out by their selected release model. While the first reaction to deadline misses is to send reminders, these still did not prevent projects from missing their milestones. A suggestion was made to unsubscribe such projects from the forcefully "*cycle-with-milestones*" model. In such situations, the

[17] `shorturl.at/pvZ59`

release team has a policy to "force a release" on projects when they miss an important deadline, enabling the team to take the previous stable release of a project and "force" it on the current release for that project.

`Meeting deadlines.` Conversely, on February 9, 2018[18], the release team showed their satisfaction with the progress of projects in resolving stabilization bugs. Some release team members even commented that the Queens release cycle had been one of the most successful in terms of commitment towards deliverable deadlines.

Experts: The interviewed release team members unanimously agreed with our analysis, while the interviewed project members provided additional insights. For example, two people stressed that the current 6-month release cycle is too short, especially for participating companies with typical timelines (delays due to code review and external dependencies are widespread). They have many other objectives apart from OpenStack. An inevitable decline in new feature development was observed, which might also be due to too short a cycle time.

While the missing of deadlines was acknowledged, the intermediate deadlines of the "*cycle-with-milestones*," enabled both the release and project teams tremendously in predicting the outcome of the ecosystem release: *"Yes, for example, that was one of our problems with a patch that I was developing with my team was a feature that was hard to understand. For many other team members, and since we missed this deadline . . . , which meant that we already saw that from a technical perspective, it's going to take a couple of weeks or months to implement the given feature. Still, we already knew that it just wouldn't make into it until the next release"* (**PP2**).

Meanwhile, concerning why projects follow different release models, another expert

[18]releaseteam.2018–02–09–15.00.log.html

explicitly motivates why they follow the "*cycle-with-intermediary*" release model: "*... So, it's some sort of 'flexible' I will say, but not a strict model. We try to get things done the most that we can get done, and then if we fail to deliver what we need to deliver, we do retrospectives and try to address the needs for the next cycle.*" (**PP3**).

Figure 5.9 Six-month release candidate-based model ("cycle-with-rc").

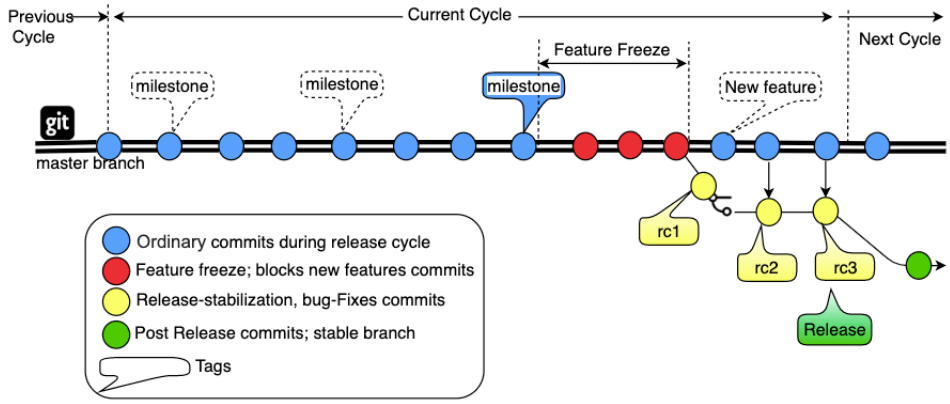
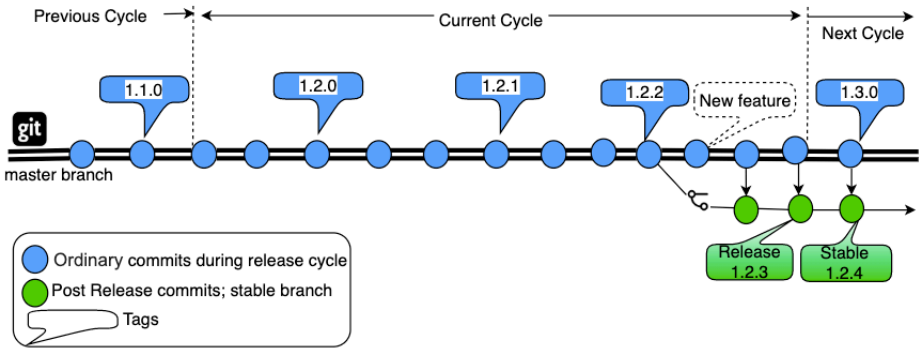


Figure 5.10 Six-month intermediary release model ("cycle-with-intermediary").



A2.Release Planning and Tracking

What: Coordination and tracking of the ecosystem's release roadmap.

Why: Since the release model chosen in `Release Model Management` only stipulates the

high-level obligations of a project in terms of deadlines and hand-off of releases to the maintenance team, the activity of release planning is necessary to consider the specific requirements of the project's release from the perspective of the ecosystem release. This release plan prioritizes the essential features and bugs requested by the ecosystem's users. It serves as a formal agreement or commitment between projects and the release team in terms of *content* of the project's release and the precise *timing* of a given release (e.g., assigning exact dates to the milestones of a time-based release model). To ensure the plan works out, the release team should track the ecosystem projects' progress over time, possibly adjusting strategies as needed.

How: OpenStack (OIF) organizes several events to bring together a wide variety of community members. For example, the main ecosystem event (Open Infrastructure summit) happens bi-annually and is open for everyone in the ecosystem, including newcomers and vendors. Usually, OpenStack announces the most recent ecosystem release to the general public during this summit.
Some other events (Forum and Project Teams Gathering (PTG), formerly known as "the design summit") are more technical and focus on getting feedback from operators and proposing the requirements and design for future OpenStack releases.

The Forum[19] is a co-located event to the main OpenStack summit. It is organized around DevOps (developers+operators) to discuss the "What" of the OpenStack design on specific features or issues and gather extensive feedback from Devs and Ops. Moreover, Forums focus on strategic discussions and marks the next release cycle's planning phase. For example, Devs can ask specific questions to Ops on their user-experience, and Ops provide feedback from the previous OpenStack release. Also, Ops provide suggestions on

[19] shorturl.at/glrwx

cross-community collaboration relating to new features in the upcoming release. Typically, there are three types of discussion sessions: Project-specific sessions, Strategic session on community-wide concerns, and Cross-project sessions.

On the other hand, the PTG[20] happens right after the summit, organized for developers/-contributors from different cross-project and projects teams who are engaged in improving the future releases of OpenStack. PTGs discusses the "HOW" in implementing the forum's suggestions; getting an agreement, building trust among projects/cross-projects teams w.r.t the release teams, assigning work items, and getting work done.

PTG schedules are dynamic and span across three days, permitting project teams to be flexible and productive. During the PTG, the release team starts an online collaborative document (Etherpad[21] in the case of OpenStack), specifying each project's required tasks, the release team manages for the entire next release cycle. These tasks, listed as todos, range from high priority ecosystem features to improvements in release automation or changes to the release process. Liaisons across different projects also integrate their project's proposed project plan. Finally, each task is assigned to a release team member to follow up.

Once the new release cycle starts, the release team goes over each active task in the release plan document at the beginning of each weekly meeting. The assigned team member or liaison reports back and suggests follow-up tasks. As project teams implement tasks, the release team mark those tasks as done. Similarly, the release team updates the plan depending on the newly added or modified task. The release plan is a living document that the release team closely monitors during the release cycle.

[20] shorturl.at/jmpF2
[21] shorturl.at/lnGW5

Finally, on the last day of a release cycle, the release team goes over the plan a final time to ensure that all tasks are completed, with left-over tasks possibly delegated to the next release.

Examples: Last minute issues. On April 1, 2016[22], the last but one meeting (R-1) of the Mitaka release was held. A discussion centered on "step-by-step release week actions" in which the release team lead urged some PTLs and Liaisons to have a pre-release meeting and brainstorm on how to process their releases. The release team PTL also noted that some PTLs/Liaisons who were absent from the previous meeting did not do the tasks that they were assigned to do. Slightly, they were raising concerns that were not part of the agenda for the ongoing meeting and were not also documented in the etherpad (tracking document).

Moreover, one other release team member suggested that since there is already a task action in the release etherpad pointing to the last meeting (R-0), everyone should follow the etherpad documentation. As the discussion progresses, the release team PTL said he would need to send out the usual announcement for the last meeting (R-0) earlier than scheduled, and other release team members agreed to do extra hours of work to get things done on time.

Overwork. On October 20, 2017[23], a project team liaison wanted the release plan to be modified to meet a pending deadline. Several non-trivial changes should go into the plan that required an integration test to validate whether the changes were correct. However, other members refused because if they modify the release-plan at this point in the release cycle, there would be no guarantee that the rest of the release timeline would be respected. After much back-and-forth, some members stepped up who were willing to work extra

[22]releaseteam.2016-04-01-14.01.log.html
[23]releaseteam.2017-10-20-15.00.log.html

hours to fix the problem without delays.

Experts: The interviewees explained that release dates are not determined using strict rules but are established in collaboration with the OpenStack foundation in a pragmatic manner. For example, the release team will "*look for other holidays or major events that are going to be happening and try to pick a good day for the final release [...] we didn't want to release things on what was a Friday afternoon...*" (**RP1**). Similarly, **RP2** (and **RP4**) stressed that a significant goal of the release plan is to "*make sure everyone knows where we are in the cycle,*" rather than serving just a managerial purpose.

A3.Dependency Management

What: Since, by definition, the individual projects within an ecosystem depend on each other and third-party libraries, the ecosystem's releases need to agree upon and specify these dependencies as well as constraints on eligible versions of each dependency.

Why: Ecosystem projects are inter-dependent by nature [78, 80, 243], which can yield a variety of problems commonly referred to as "dependency hell" [81]. For example, two projects could depend on different, incompatible versions of a third project. A project could depend on a project that will not be included in the ecosystem release or on a third-party library with an incompatible license or does not reach the desired quality level. Even if project dependencies would be specified, individual projects could churn out new releases faster than foreseen (given that they are following their road map), forcing other projects to keep on updating to newer versions of their API.

How: Ecosystems require their projects to register all their dependencies in a centralized dependency specification (e.g., in OpenStack this specification is stored in a file called "global-requirements.txt"). As is common in open source projects, project version numbering, and hence the dependency specifications[24], use semantic versioning rules [81]. Furthermore, the dependencies in the centralized specification are enforced during ecosystem release testing. Any changes to this file, specifically, addition, removal, or modification of version constraints, need to be reviewed by the release management team.

The latter team also runs tests against unpinned dependencies — the practice of not making explicit the versions of software components depends on, which is not a good practice to encourage. On the other hand, dependencies pinning 'freezes' dependencies, which in turn makes deployment repeatable. If not, different versions of the same software component will run at each re-staged of servers. Also, unpinning dependencies prevent notification of vulnerability. Consequently, this implies that a test on each patch tests two things, (i) changes in that patch, (ii) releases of dependencies. At the project level, each project has its own local sets of dependencies for that project, and this implies that any arbitrary patch is tested on changes in the patch itself and for dependencies.

The management of dependency specifications and review of any changes to them is not necessarily the sole responsibility of the release team. In OpenStack, these two activities are the shared responsibility of the release team and the requirements team, independent of one another. The dependency specification reviews that they perform consider several important questions, such as: "Does the library or project dependency have a (responsive) maintainer?", "Is it still active or deprecated?", "Does it have a compatible license?", "Do the dependency's APIs overlap with or complement existing dependencies?", "Is the library backward-compatible?", etc. In order not to miss dependency changes during the review,

[24] shorturl.at/ekBF9

tools can automatically flag and track such modifications to the reviewers.

However, what is currently missing in this activity, but seems essential, is any mention of the timing of communication/coordination about dependencies for a given ecosystem. For example, when do projects decide to adopt a new version of another project (which is still under development) and know what version of APIs to use other projects (since even under construction)?

Currently, they do this through project liaisons who updates the cross-project teams weekly on changes to their project that might affect other projects that depend on their deliverables.

Example: `Floating dependency specification` On February 10, 2017[25], the release team tried to resolve the issue of floating dependency specifications. This issue pops up when a maintenance branch is made for a stable ecosystem project release. Often, a project forgets to specify a concrete version number of dependency and instead only refers to the latest version. Since the maintenance branch is dedicated to maintenance activities on the state of the ecosystem at the time of a specific release, while in the meantime the project continues to evolve, floating dependencies would try to download potentially newer and different dependencies than available at the initial time of release, flagging confusing test errors.

As the discussion progressed, some members proposed alternate dependency requirements. One possible idea was to use local requirements, whereby each project handles a local copy of its dependencies, which are consumed by that project. Yet, this idea was rejected by the release team for fear of incompatibilities. Another idea was to develop a

[25]releaseteam.2017–02–10–15.00.log.html

script to validate stable releases (this stable release checked checks for floating dependencies), making sure that a new release of some third-level dependency does not invalidate the constraints change (that goes with every dependency). This impacts new development: "you can't do this change since it breaks an earlier version under maintenance". Eventually, a decision was made to have stable project releases refer to the specific version of each of their dependencies in "*global-requirements.txt*" (discussed in a commit message):

```
Author:     Thierry Carrez <thierry@openstack.org>
AuthorDate: 2019-11-19 10:11:52 +0100

Unpin keyring to unbork twine validate check

As reported by Tim Burke, just-released twine 3.0.0 requires
keyring>=15.1, but we have a 3-year-old keyring==7.3 pin in
releases requirements.

The pin was put to avoid installing a full desktop (through a
DBUS dependency introduced in >7.3) but this seem to have been
fixed between now and then.

Change-Id: I904ea3735daaa8892edc54be4a9611beb66fdfe2
```

Experts: Communication was mentioned to be an essential practice to reduce the impact of changes on dependent projects. The release team members explained how their team could play a vital role in this by warning dependent projects of essential changes ("instructive communication") or organizing online meetings to bring together stakeholders of affected projects ("dialogue").

The project members acknowledged this and also stated *"I feel like we've gotten much better being effective at that kind of communication."*(**PP4**). The release team also explained how their collaboration with the requirements team is crucial to prevent projects from using buggy dependencies.

A4.Establishing a Network of Trusted Liaisons

What: The appointment, assignment, and training of trusted liaisons, specifically, members of individual projects responsible for coordinating their project and the ecosystem release team on all matters related to the inclusion of their project's releases into the ecosystem release.

Why: Given the autonomy of projects inside an ecosystem, the release team needs to interact with representatives of each involved project to coordinate the ecosystem-level releases. This activity covers information exchange, the delegation of technical changes, project members' training about the overall release process and automation, reporting problems, etc.

How: For each project, the ecosystem recommends a single point of contact ("trusted liaison"), which often is the technical lead of a project ("PTL" in OpenStack), but in theory, could be any member suggested by the project or cross-project team. Projects/cross-projects teams announce any changes in the liaison list in time for the upcoming release (e.g., in OpenStack, technical leads are only elected for a given release cycle).

The liaison role is a technically demanding job, with many responsibilities, requiring experienced, skilled people. For example, each liaison must be present in the release team meetings ("#openstack-release" IRC channel in OpenStack) to answer questions about his/her project and be ready to address technical issues. A liaison should also ensure that release-related patches are reviewed on time by their project and should pass on any queries, messages, or training material from/to the release team.

Example: PTL vs. Liaison. During several meetings, the performance of current PTL-liaisons is discussed, with opinions ranging from very positive to needing improvement. From these discussions, it is clear that PTLs' implication in the overall release process is one of the criteria for re-election as PTL.

For example, on November 27, 2015[26], one of the PTL-liaisons was eligible to re-run for the upcoming election due to excellent contributions and commitment over the past cycles; you need to have been a perfect liaison to be re-elected. While the incumbent wanted someone else (without experience) to run for election, he finally agreed to serve if elected in office.

On another note, in January 2016[27], the slow progress of work assigned to another liaison was possibly impeding the person's chances to run in the upcoming PTL election. In this case, the liaison ran for election, however, was not elected as the PTL.

Experts: ***RP2*** mentioned that: "*the release liaison [. . .], it's an important role for the release team to ensure that we have a release at the end of the process. And so we have a lot of safeguards in place, and one of them is to make sure that we know who to contact, who can be involved directly, so that's a very integral part [of the release team's responsibilities]*".

Each week, the project/cross-project team liaisons update the release team regarding envisioned changes in the upcoming release.

[26] releaseteam.2015–11–27–14.34.log.html
[27] releaseteam.2016–01–08–14.00.log.html

A5.Establishing Communication Channels

What: Establishment and maintenance of communication channels between projects (liaisons) and release team to ensure effective synchronization of releases.

Why: While the previous activity identified the liaisons that the release team interacts with, dedicated communication channels are needed to propagate all information related to release activities from/to liaisons, for example, to exchange instructions, bug reports, new releases, manage dependencies (A3), etc.

How: The communication channel depends on the kind of information the release team wants to communicate. For example, to communicate instructions, report major issues, etc., liaisons should attend the (online) meetings organized by the release team. Meanwhile, for more informal contacts, specific chat-rooms, private messages, or mailing lists are used. Project management tools such as Trello (possibly integrated with the issue repository) can enable teams to track work progress.

Apart from conceptual information, ecosystem releases also require the exchange of artifacts with individual projects. Again, ecosystems should encourage the use of different communication channels for different kinds of artifacts. The OpenStack case study showed how loose coupling, supported by automation, is essential. For example, for a project to signal a new release's availability, all it needs to do is use special tags in its Git repository to tag a branch. Release automation then automatically picks up the latest release and automatically starts testing. To exchange project artifacts that need to be consolidated into one ecosystem-level artifact, the release team should follow the process and communication channels of activity A8. For example, to generate the release notes that open source

distributions, press, and marketers use.

Examples: Communication with Stakeholders On March 25, 2016[28], during one of the weekly release team meetings, there was an ongoing discussion concerning what communication medium the release team should use to reach all different stakeholders, including PTLs since the release team wanted to change their meeting days from Mondays to Fridays. Thus, the release team needed confirmation from all the stakeholders before they could make the change.

Some members suggested that the release team lead send an email to the mailing list addresses and ping (to start a conversation with) the relevant people on the IRC channel. As the discussion progresses, we observe that the release team used all the communication media to send the community's message.

Furthermore, one release team member expressed frustration because he could not reach a particular technical committee (TC) member assigned to work with the release team member on an upcoming project team gathering (PTG) meeting. Two weeks later, the release team member could ping the TC member over IRC and invite her to follow the IRC release channel for onward discussion concerning the PTG meeting.

Experts: As mentioned earlier for A3, all interviewees uniformly agreed on the importance of communication and using the right channels. For example, regarding the use of Git tags for communicating (the contents of) a new release, **RP1** added that "*that process helps us do things like making sure that teams are communicating correctly, ... and then chasing up the people that don't ... that's the main: communication, communication, communication*". On the other hand, IRC was preferred for in-person communication with

[28]releaseteam.2016–03–25–14.00.log.html

and amongst the release team.

A6.Release Stabilization

What: Coordination of testing efforts, and fixing of integration and other urgent issues on the upcoming ecosystem release.

Why: Once individual projects have handed off their release to the release team, e.g., after feature freeze in a "cycle-with-milestones" model (see A1), the release team is in charge of testing the upcoming release. Release stabilization is a significant activity since its input is the set of releases of the individual projects planned for inclusion in the forthcoming ecosystem release. At the same time, its output should be a well-polished product, ready for the end-user.

How: The majority of this activity's work involves integration and automatic system testing and bug fixing of identified issues, typically under tight time pressure (due to the impending deadline). For example, these tests would find conflicts between project dependencies or incompatible API changes (cf. A3).

Projects are tested first at the project level; then, the release team does the integration testing and debugging. In general, three types of testing happen here to stabilize the release: unit, functional, and integration testing. In case of problems, the release team collaborates with the projects whose tested release has an issue.

Example: `Priority bugs.` On December 18th, 2015[29], release team members discussed the "priority list" of open bugs blocking the final release. Several tasks were assigned to

[29]releaseteam.2015–12–18–14.21.log.html

members, most of which were related to testing the Reno consolidation machinery (see A8). Some projects had difficulties integrating Reno properly. As a result, these braked some functionalities in those projects, and the release team had to run several test cases, including integration tests on all those projects. Finally, the problems were all resolved, and the final release was successful.

Experts: ***RP4*** stated that "*the stabilization and automation activities are the most time-consuming activities for us*". In particular, the release team spent most of its time reviewing and **testing** submitted features of projects (such features were also discussed earlier on in IRC, after that sent to ZUUL (CI) for automated testing). Due to the amount of effort involved, the release team encouraged projects to attract and train community members to become skilled at helping out with project-level testing.

A7. Automation Management

What: Determining the need for, the coordination of, and the management of ecosystem-wide tools for release automation, such as CI, system, and performance testing and deployment/release.

Why: Release engineering is known to be a highly automated domain [73, 244], since many activities, such as the compilation, testing, deployment, etc. are repetitive and would be error-prone when left to manual interaction. While this holds for the release process of individual projects, it also applies to the ecosystem's release process as a whole, since a central release engineering team would be unable to manually release and coordinate the individual ecosystem projects taking into account the dependency constraints between

them. Without which, this would be a tedious, error-prone job for the central release team. Delegating such automation to the individual projects is not scalable either, given that they lack an ecosystem-wide view.

How: During the weekly release team meetings, automation-related issues are initially raised and discussed at the project-level. Those discussions typically try to flesh out the rough requirements for new automation. If an existing open-source tool is available to satisfy these requirements, the issue can be resolved locally (by the release team). If not, the project-level discussion sometimes brainstorms potential designs for new automation before escalating the issue to the ecosystem-level Infrastructure (Infra) team responsible for building the infrastructure around the ecosystem. However, the release team builds the automation that runs release jobs.

Indeed, the ecosystem-wide infrastructure has integrated some of the tools that the release team conceived and developed locally to facilitate the release automation process. Further, the release team contributed some of these tools to the Python community. In contrast, the release team host other tools on their repository[30], which is publicly available. The release team then brings up such issues in one of their regular meetings to make an executive decision, such as discarding the plans, starting work on a new tool, or evaluating the need for a particular tool.

Overall, it's preferable to use standard tools and platforms such as Ansible, Docker, or Kubernetes over custom scripts and tools. Standard tools enable better compatibility with the full range of technology used by individual projects and reduce development effort. Only when standard tools do not work out are custom ones explored. For example, OpenStack initially used Jenkins as its CI platform, but this did not scale to the massive volume

[30] `shorturl.at/cdgn7`

of commits merged per day and the enormous scale of tests aimed at running on Jenkins. Therefore, the ecosystem made a conscious decision to implement a new CI platform, Zuul. While Zuul was custom-built for OpenStack, the ecosystem recently made it open source to become a standard CI environment.

When the ecosystem decides to pursue a new tool, the release team members consult with the infra team responsible for developing the ecosystem infrastructures. At the same time, the impacted liaisons keep track of progress. In case a tool exists that could do the job, the liaisons instead perform a search online and analyze the identified tool's acceptability. Once the resulting tool (either developed or acquired) has been evaluated and shown to be functional, it's time to discuss its integration into the regular release team meeting's ecosystem release process. Since the developed tools' actual operation is not the release team's responsibility, the infrastructure team, members of the latter, are also involved in those meetings.

Examples: `Standard vs. Custom technology.` On June 5th, 2015[31], the release team members were exploring ways to improve the performance of several tools. Tools developed in-house are mixtures of scripting languages and perform slower [258]. Furthermore, there was an alternative plan to find more scalable, standard libraries to replace those scripting-tools. The release team also discussed a tool for tracking milestones. However, some members suggested that the tool was not idempotent; specifically, repeated executions each time would yield different results. Consequently, the PTL requested a release team member to migrate the Bash tool to Python to reuse existing libraries.

`Temporal measures vs. Full automation` On January 8th, 2016[32], the release

[31] releaseteam.2015–06–05–13.01.log.html
[32] releaseteam.2016–01–08–14.00.log.html

team noticed that they were stuck with the automation process to release milestone-2 for the Mitaka release of a particular project (Barbican) since the project was breaking the Zuul automation pipeline on integration test. While the release team and the Infra teams did the best to resolve the problem, certain activities like release note consolidation and milestone announcements would have to be done manually for the time being.

Experts: **RP3** summed things up nicely by stating that "... *the main [responsibility of the release team] is communication, communication, communication, and then the second provides the tools and the automation behind the process.*"

A8.Deliverable Consolidation

What: Consolidating artifacts produced by individual projects into a centralized, ecosystem-level repository managed by the release team.

Why: While the autonomy of individual projects to set out and pursue their release schedule is one of the strengths of an ecosystem, end-users should not be assumed to keep track of each project's release cycle, versioning numbering, dependencies, etc. Instead, they expect to install and use a vetted combination of compatible project versions. The same principle applies to more mundane artifacts, such as having one source of documentation for the released ecosystem, one set of release notes, or even one central download site.

How: The key to consolidating ecosystem artifacts is to provide a central repository for the individual projects to submit their finished deliverables and guidance and tool support to integrate this submission into their workflow. The latter is the most significant challenge since each project uses its local process and automation, hence loose coupling with the

central repository is preferred. Typically, either an API is provided to the projects or a standard location where projects can put a deliverable for the release team's tool-chain to detect and integrate it automatically.

While this consolidation applies to a wide range of deliverables, these do not necessarily require custom tools or processes to release notes and documentation from actual project releases. For example, the OpenStack release team designed a generic process for consolidating deliverables. During the release team weekly meetings, this process comes up often, which only relied on a central Git repository with a configuration YAML file listing for each sub-project, the current version (commit hash) whose deliverables are part of the ecosystem release.

Example: Release Notes. On Nov. 11th, 2017[33], the Reno infrastructure for consolidation of release notes was discussed. Reno logically groups release notes of individual projects, which are arranged into categories/sub-categories that automatically triage to the right reno category[34], for example, new features, bug fixes, etc.

This grouping occurs immediately after project teams add the notes into their project repository, then Reno pushes and integrates them into a central Git repository. Reno manages release notes for OpenStack deliverables in the same git repository where the project's source code is in, the central repo with all projects' code. Therefore, Reno can track the history of files across all branches and all revisions of the ecosystem. For more implementation details, we refer to [79].

While the Reno tool was ready to go into production at the meeting time, most of the discussion involved informing and training individual projects about Reno. Also, whether

[33]releaseteam.2017–11–17–15.00.log.html
[34]shorturl.at/czNRS

to perform the adoption of Reno in parallel with the training or afterward. The release team finally decided that the training and roll-out of Reno should happen concurrently. Based on the analysis of the log files, this happens to be a common practice at OpenStack. Whenever there is a new tool (in-house or third-party), the release team usually organizes training parallel to project members to use the tool.

Experts: **RP1** mentioned that "*we don't want the release team to manually mark version numbers on a bunch of release notes or anything like that instead, and we want to handle all that stuff automatically*" and that "*Reno has increased the 'number' of release notes.*" To validate the latter, we analyzed the online Reno repository. OpenStack launched Reno in 2015,[35] and in July 2018, over 13,000 release notes were gathered by Reno for over 300 projects, which is far beyond the experts' initial expectation and more than all the release notes written before Reno.

A9.Stable Release Maintenance

What: Coordinating maintenance activities after an ecosystem release, resulting in bug fix (minor) releases.

Why: While work is underway on a new release, the previous release should be kept stable for as long as initially planned. Thus, bugs that slipped through the release stabilization activity or post-release bugs (reported by end-users) have high priority. Special attention goes to reported vulnerabilities, which need urgent resolution.

How: Reported bugs go to the concerned projects where actual fixing happens. Meanwhile,

[35] shorturl.at/AFVY9

the release team is in the perfect position to coordinate the resolution process. Moreover, the release team applies stricter rules about stable releases of projects on their stable branches and should have a review step.

Project teams focused on issues related to their projects because the projects know the internals of their code best and because they typically will resolve such bugs first in the code of their next release (to ensure that they do not have regressions), then back-port the relevant fixes to the stable branch (es) of the previous release(s). The release team delegates reported bugs to the project liaisons, who propagate them within their respective project/cross-project teams.

In OpenStack, stable branches typically are maintained for about 18 months, but this can go beyond (extended maintenance) in case of strong demand. A release enters end-of-life (EOL), specifically, requires no further maintenance activities. Maintenance has three phases:

1. first-six-months: all bug fixes accepted;
2. six to 12 months: only critical bug fixes and security vulnerability patches accepted;
3. 12 to 18 months: extended maintenance, accepting only security patches.

Example: `Whose responsibility.` On May 6, 2016[36], some projects asked the release team to manage stable release management for them. The request was partly due to confusion about the extent to which maintenance of stable releases was covered by the release model that project teams selected and partly due to those projects not being managed by the release team at that time. The majority of the release team members refused the request, preferring to delegate this responsibility to a "stable team" comprising each project's liaisons (as recommended by the OpenStack governance policy).

[36] releaseteam.2016–05–06–14.02.log.html

Experts: **RP1** clarified that "*we enforce stricter rules about stable releases of projects on their stable branches and having a review step [. . .] allows us to make sure that the stable maintenance team has an opportunity to look at the release and approve the release in addition to the release team.*"

As mentioned in A1, the scope of the release model goes beyond the actual release. The exact duration that the release team provides maintenance is undefined: "we will keep two or more existing releases in maintained State and then an undefined number in extended maintenance and then a very few numbers and un-maintained, and then the rest would be end-of-life" (**PP1**).

A10. Cycle highlights Management

Prologue: The release team experts unanimously agreed about the nine identified synchronization activities. They observed that shortly after this chapter's studied period, the release team adopted a new activity, Cycle Highlights Management (A10.), which targets non-technical users dealing with the release process, specifically, public relations, marketing, media, etc. Instead of ignoring this activity (since it came towards the end of the Queens release cycle), we decided to write up still the tenth activity profile, based on the experts' feedback and other background information we could find. In other words, we could not derive this activity from the studied IRC logs and release plans.

What: The release team aims at providing a high-level summary of the next release's features to stakeholders who are neither developers nor operators, such as sales/product managers, media/press, marketers, and users. Thus, this summary should provide a concise

overview of the upcoming new features and other changes in a non-technical manner.

Why: Today's evolution towards more rapid releases and continuous delivery [73, 259] bears the risk of alienating users since they might become overloaded with ecosystem releases whose contributions are unclear and overwhelming. It could lead users to not update to the newer ecosystem releases, which would increase the ecosystem organization's support costs and the risk for vulnerabilities/bugs.

In particular, in the OpenStack ecosystem, a higher volume of emails from non-technical users was received on the community mailing list concerning what to expect in upcoming releases. For some time, each project team responded independently to these community requests, leading to inconsistencies or sometimes no response from project teams. Hence, the ecosystem decided to provide an ecosystem-wide high-level overview of changes, called "cycle highlights."

How: Starting towards the end of the Queens release (see Figure 5.3), specifically, right after the period studied in this chapter, the OpenStack release team introduced a template that the individual projects should use to present cycle highlights Management to non-technical users. The template is stored under *deliverables/$RELEASE/$PROJECT.yaml* in the *openstack/release* repository, and contains the following fields:

1. A summary of fewer than two lines in length.

2. What are the changes/new features, etc.?

3. What benefits do these changes bring to the user?

At the beginning of each release cycle, specifically during the three-day PTG/design

submit, the release team requires all project team leads or liaisons to fill out the template and submit their resulting highlights.

The release team encourages project teams to do this before the rc1 deadline so that it will be able to edit and help projects write consistent highlights that showcase their work.

OpenStack introduced A.10 towards the end of the Queens release cycle. Moreover, Train was the most recent release at the time we were writing this chapter. Besides, we had already studied the IRC logs, and those logs didn't contain any information regarding A.10. Further, information concerning A.10 didn't appear on the PTG sessions before the Queens release. It didn't feature anywhere else in this chapter because it was the most recent information when we were performing this study.

Example: `Whose responsibility?`

The author of this thesis attended a 3-days **Train** in-person PTG event to understand better the design submits. During one of the sessions, there was an ongoing discussion concerning the cycle highlight Management of some project teams that didn't send their cycle highlights to the release team in time.

"*Cycle-highlights collection needs to start earlier . . . otherwise marketing asks for info too early and we can't provide them the info without collecting it manually.*[37]"(Anonymous1)

In response to this suggestion from the foundation members, the release team lead requested the dedicated person-in-charge of A10 to adjust the window frame for A10 and make sure the project teams' cut-off date to submit their highlights comes before the feature freeze period (FF). The foundation would be aware by the **rc1** deadline of any projects that have not yet sent their highlight.

[37] `shorturl.at/hsFP1`

Later, based on the discussion that the release team documented on the etherpad during the Train PTG[38], we learned that about 4% of the project teams did not submit their highlight during the Train release cycle, and we verified this observation by counting the projects that didn't submit their cycle-highlights (it is missing) from the Train release page[39]

Experts: We asked the release team experts to explain why cycle highlight Management is an important release team activity during the interview session: "*... The release team gives cycle highlights to the public relation and marketing staff within the foundation. They use that to talk to analysts. They prepare a presentation at each summit that includes that kind of information sometimes, and it makes it into the keynotes stage at the summit. It's not helping people who are not directly involved in the development understand the significant work that happened for each release.*"(RP1)

5.4 Discussion and implications

5.4.1 Discussion of our findings

Co-existence of multiple OpenStack release models

Indeed, in every release cycle, the release team has to manage multiple release models simultaneously, which is at the heart of the complexity of ecosystem release synchronization, as shown in Figure 5.11. For example, Cinder follows the Cycle-with-rc model; meanwhile, Glance follows the Cycle-with-intermediary model. The central release team has to keep track of both projects throughout the development cycle.

[38] shorturl.at/ctDG2
[39] shorturl.at/xGOQ1

Figure 5.11 Ecosystem release with projects following different release strategies in a release cycle.

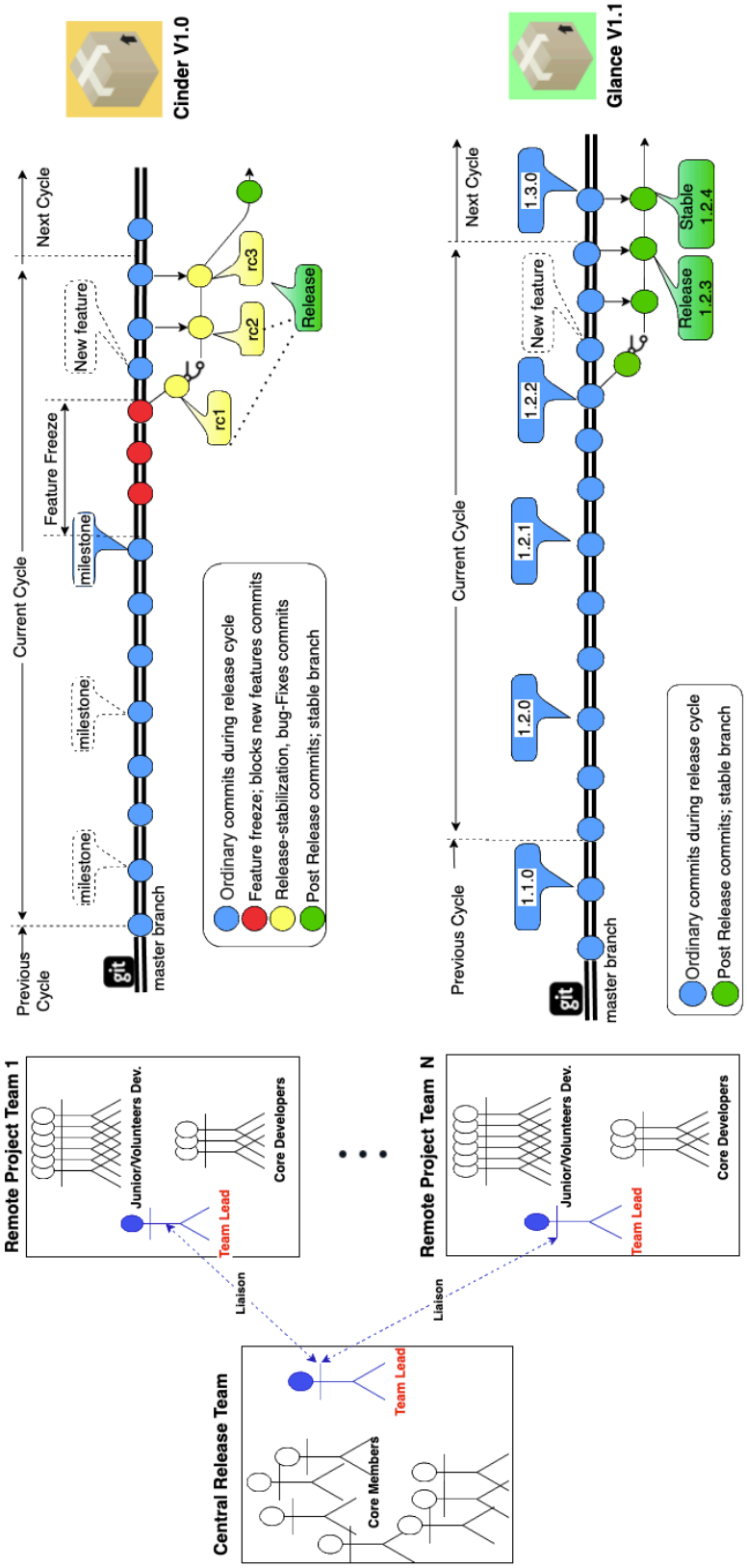


Figure 5.12 Excerpt from a "*.yaml*" configuration file, showing the Swift project on Rocky release following the cycle-with-intermediary release models.

```
---
launchpad: swift
release-model: cycle-with-intermediary
team: swift
type: service
repository-settings:
  openstack/swift: {}
branches:
  - name: feature/deep-review
    location:
      openstack/swift: 331339246228167dfab61690327fc0c3f2fab8af
  - name: stable/rocky
    location: 2.19.0
releases:
  - version: 2.18.0
    projects:
      - repo: openstack/swift
        hash: f270466de363499894317b7c671f65e8a912bd53
  - version: 2.19.0
    projects:
      - repo: openstack/swift
        hash: f0472f1f7975957fc31cb8c123aa82ee47848645
  - version: 2.19.1
    projects:
      - repo: openstack/swift
        hash: 3d2d954107d676e48acb81069639eed15ead5713
  - version: 2.19.2
    projects:
      - repo: openstack/swift
        hash: 5aa4c5d88fec98cf4ca0536c511c61759e865ec9
  - version: rocky-em
    projects:
      - repo: openstack/swift
        hash: 5aa4c5d88fec98cf4ca0536c511c61759e865ec9
cycle-highlights:
  - Added an S3 API compatibility layer, so clients can use S3 clients
    to talk to a Swift cluster.
  - Added container sharding, an operator controlled feature that may
    be used to shard very large container databases into a number of
    smaller shard containers. This mitigates the issues with one large
    DB by distributing the data across multiple smaller databases throughout
    the cluster.
  - TempURLs now support IP range restrictions.
  - The trivial keymaster and the KMIP keymaster now support multiple
    root encryption secrets to enable key rotation.
  - Improved performance of many consistency daemon processes.
  - Added support for the HTTP PROXY protocol to allow for accurate
    client IP address logging when the connection is routed through
    external systems.
release-notes: https://docs.openstack.org/releasenotes/swift/rocky.html
```

Over 2961 projects/modules exist in the OpenStack release team repository across all releases. However, 105 (3.5%) are not coordinated by the release team; they do not follow the 6-months release cycle, even though, from time-to-time, they produce releases independently without the coordination of the release team. The majority of the project teams (96.44%) are coordinated by the release team and follow one of five ecosystem models.

- **Model-1**: Cycle-with-intermediary is the most prominent release model and over time 1,746 (58.97%) of OpenStack project/cross-project teams have been using this model.
- **Model-2**: Cycle-with-rc, which replaces the Cycle-with-milestones model after the Rocky release and ranks the second most popular model within the project/cross-project teams. In OpenStack's history, over 319 (10.77%) project teams have signed in to use this model.

  OpenStack Legacy release models:

  - **Model-3**: Cycle-with-milestones, now replaced with cycle-with-rc, from Austin to Rocky releases, model-3 grows in popularity and then stop existing after Rocky release. Actually, it merges to cycle-with-rc from Stein and onward releases. In total over 308 (10.42%) of OpenStack project/cross-project have used this model.
  - **Model-4**: Cycle-trailing was introduced in the Liberty release and continues till the Train release; later, it was replaced by the cycle-with-intermediary release models. Throughout its existence, 371 (12.53%) of OpenStack projects/cross-project teams used model-4.

– **Model-5**: Cycle-automatic, this release model was also replaced by the cycle-with-intermediary model, and over 60 (2%) of project teams have used this model-5 in only two major releases; Train and Ussuri.

- **Model-6**: Untagged

  (*The release team uses some continuous integration (CI) and automation tools only from source and never tag releases, but need to create stable branches. For example, the Grenade projects*[40] *from the Quality Assurance team uses the 'Untagged' as release-model. Hence, we found 52 instances (1.75%) where different CI tools (projects) across all OpenStack releases are marked 'Untagged'.*)

- **Model-7**: Abandoned

  (*We observe that with time, some OpenStack projects/modules became obsolete and produce no new future releases. These projects can either be 'abandoned' or merged with other project/module. In the history of OpenStack, 19 (0.64%) projects/modules are marked 'Abandon'*)

- **Model-8**: Independent

[40] shorturl.at/gGJY8

> *Usually, projects that are independently released can either be in two possible conditions such as 'Abandon' or switch release model to a release-team managed model. In total 105 (3.5%) of projects have been released independently.*

Moreover, our study of the release team communication (IRC meeting logs) and release plans (Etherpad docs) shows that the ten release synchronization activities apply to all the release models. Besides, most OpenStack ecosystem projects consider the first two models: Model-1 (Cycle-with-intermediary) and Model-2 (Cycle-with-rc). Therefore, no matter the release model that a project uses, we can study their release synchronization activities.

In particular, Model-1 (Feature-based model) is typical with projects that are doing several 'intermediary' releases before the final ecosystem release, and this model favors projects with many new features coming in.

Meanwhile, Model-2 (Time-based model) is typically for projects experiencing a low volume of new features coming in and aim at doing one major release per-development-cycle with one or more release candidates (RC). Notably, most core projects that are reaching maturity prefer this model.

Perception of OpenStack release process

> RP2:"... The release team's primary role is to synchronize the ecosystem releases in collaboration with the project teams that are released managed, ...
>
> The second role of the release team is to provide the **tooling and automation** to support the release process."

Consequently, it comes as no surprise that *A7 (Automation/tooling)* is pivotal to the release process in complex ecosystems. We found evidence suggesting that the release team has automated more than 80% of the release engineering activities. These tools are open source and hosted online in OpenStack's repositories[41], while some of these tools t1[42], t2[43], and t3[44]. have been contributed to the Python community.

On the other hand, some of the project practitioners we interviewed, who also happen to be project team leads (PTL) in their respective projects, affirm that from December 2015 on, some OpenStack projects started experiencing a decrease in new features. Therefore, since fewer new features have been coming in, these projects suggested switching from a six-month cycle time to a nine- or even twelve-month cycle time.

> PP2:"... there's not a ton of new features coming in anymore; it's a lot of like stabilization and optimization. I think that nine months would be fine because there are not as many new things coming in anymore that we need to like release to get out the door for collaborators and stuff."

In contrast, some other projects were still experiencing a high volume of new features coming in (and they requested a shorter, three-month cycle time), the release team settled on the current six-month release cycle. Hence, the six-month cycle time is not a dogma or magical number but rather a balance to satisfy the different ecosystem projects' needs and concerns.

[41] shorturl.at/iAGMN
[42] shorturl.at/elxW9
[43] shorturl.at/bistz
[44] shorturl.at/bxOZ9

Dependencies between activities

We observed two types of dependencies, precisely, sequential and social, among the nine activities listed in Figure 5.7.

First, **sequential activities** exist among Release Models Management (A1) $\Longrightarrow$ Release Planning and Tracking (A2) $\Longrightarrow$ Dependency Management (A3) $\Longrightarrow$ Release Stabilization (A6) $\Longrightarrow$ Deliverable Consolidation (A8) $\Longrightarrow$ Stable Release Maintenance (A9), in this order respectively. The sequential dependencies follow the chronology of the release process, which is supported by Automation/tools Management (A7) except for A1 and A2, as RP2 stated above. A1 is the only activity that occurs only once, at the beginning of each release cycle. Also, no other activity relates to A1, sequentially or socially, unlike the different activities.

Second, the **social activities** Establishing a Network of Trusted Liaisons (A4) and Establishing Communication Channels (A5) ensure that the release team is in constant communication with the project teams at every stage of the release process. Thus, it facilitates ongoing communication with the project teams by ensuring that they have a contact person with the release team, helping the release team keep track of the ongoing progress during a given cycle.

Equally important, Cycle highlights Management (A10.) targets the non-technical audience of what has changed since the last releases or the new changes expected in the current release. OpenStack introduced this practice towards the end of the Queens's release. We found evidence of A10 in all the OpenStack projects located in the release team repository. For example, Figure 5.12 shows A10 at the bottom of the configuration

.ymal file for the Swift project with six new features that the Swift project added in the Rocky release. The release team then collects Cycle highlights Management (A10.) from all cross-projects/projects/modules and submits them to the marketing team. Coordinating A10 activity requires skills in writing documentation in a precise and concise manner.

5.4.2 Implication of our findings

Implications to SECOs

Despite advances in synchronizing ecosystem releases and how release strategies co-exist in complex ecosystems, release processes still fail and become irksome to end-users. Retrospectively, several ecosystems could have mitigated some of the challenges they suffered if our research's findings were available.

A single activity not well performed has a critical impact on the entire release process. As previously discussed, there are sequential and social dependencies between the observed release synchronization activities. Thus, the absence or poor execution of a single activity can derail the release team's efforts and various projects.

For example, in February 2012, Eclipse release manager sent a message[45] to the cross-project teams

He declared with the most profound regret the failure to release the first release candidate (RC1) of that series. While the release manager had been sending 'notes,' they were not regular and timely. As the discussion progresses, other release-team members retrospectively pointed out that they too were running behind schedule. For example, a release

[45] shorturl.at/etH78

engineer responsible for running specific scripts could not deliver the results before scheduled. Therefore, this shows a lack of proper planning and tracking (A1) of release activities. We observe a critical impact of this poor planning in the latter stages of the release process as dependency issues (A3) start to manifest — the release team members were dealing with dependency issues, with over 80 libraries that were missing. They described this as "worst problems" because some features were disabled due to the dependency issues. These need to be resolved urgently to prevent future deliverables from failing in the next RC2.

In another example in the Gnome ecosystem, we observe that effective communication with the network of trusted liaisons (A4 and A5) is very crucial. In 2011, many prominent open source developers (including Linus Torvalds[46]) discontinued[47] from using GNOME[48] because of core features that the release team removed without formally informing the project teams liaisons on any of the communication channels[49], leading to irritating end-users' experience[50]. Indeed, we noticed that GNOME did not make proper use of some of the synchronized activities. The GNOME release team could have created a dedicated channel with the liaisons from project teams. Maybe this could have helped the GNOME release team circulate information about the 'removed feature.' This observation is also captured in a statement[51] by one member of the release team member:

> "However, the "keep things simple and stable" meme got taken too far. Proposals to add functionality got shot down. Modules couldn't be integrated. In trying to keep things simple and stable and polish existing things rather than creating new ones, a

[46] shorturl.at/cekG7
[47] shorturl.at/nvNQY
[48] shorturl.at/rwIJ9
[49] shorturl.at/ntP09
[50] shorturl.at/yOS19
[51] shorturl.at/dpGZ3

> loose and small group of people were inadvertently alienating those who wanted to do new kinds of development in Gnome. There was a lot of discomforts because Gnome seemed stagnant . . . " — Federico Mena Quintero, April 2012.

The findings from this research could have helped Eclipse and GNOME improve their release process, supposed they had our results earlier. *They could learn from OpenStack how to plan and track the release process throughout the developmental cycle, how to manage their dependencies, and how to establish effective communication channels with project liaisons.*

These examples demonstrate what can go wrong in even mature ecosystems. Therefore, mature and young ecosystems need to know about our release synchronization activities to make informed decisions and avoid/reduce such issues. Our findings can help ecosystems practitioners to coordinate cross-project teams to synchronize releases properly.

SECOs need to continuously re-plan and re-adjust their release strategies. Based on the mishaps in the Gnome ecosystem (detailed above), the GNOME community members have since been continually expressing concern for the improvement of GNOME's release process. This is first observed on GNOME's Wiki page[52], which explicitly states that GNOME is looking for ways to improve its release models from other ecosystems. More recently, prominent community members have been proposing[53] ways to improve the GNOME release process.

A similar evolution is seen in the Eclipse ecosystem. In 2012, Eclipse introduced the "Simultaneous Release"[54] (SimRel) policy as an approach to synchronize releases within

[52] shorturl.at/kqrzW
[53] shorturl.at/ceAQ8
[54] shorturl.at/eEM28

the ecosystem. It took six years of continuous discussions and debates before the simultaneous release model was finally implemented in 2018[55]. The main discussion topic in all the threads centered on planing the Eclipse release strategy, which took several rounds of planning and adjustment of the planning.

Despite these advances in improving the release strategy, some Eclipse projects are still not convinced about SimRel. For example, some projects want to follow their schedules; they still need to respect deliverables and deadlines. In contrast, other projects that have reached maturity wonder if they still have to follow SimRel even though they do not have any new release features. Therefore, *our finding will help Eclipse understand that while Planning and Tracking is an essential activity in release synchronization, as some of these activities are sequential*. For example, Eclipse can adopt A1 (Release model management) and implement multiple strategies to take care of different project needs.

Release model management is one of the major challenges that the central release team faces. Even large ecosystems such as Eclipse continuously have to re-plan and re-adjust their release strategies. Therefore, young and even mature ecosystems can use our findings as a starting point or guideline in the evolution of their release strategy."

Implications to practitioners and academics

Our findings have several implications for practitioners, especially on managing the release process of a complex ecosystem.

[55] shorturl.at/wEKL2

Multiple strategies co-existing. Practitioners and academics could learn from our findings on why and how multiple release strategies co-exist in complex software ecosystems. They could understand why different project/cross-project teams follow their release strategies with a common goal to synchronize their releases at the end of each release cycle, as shown in Figure 5.11.

Fast cadence SECOs. Modern ecosystems tend to release with a faster cadence to provide new project features or fixes that meet end-user needs. In a recent talk by Dr. Yvonne Dittrich[56] on continuous evolution methods and tools in software engineering, a curious participant asked an important question to which there was no clear answer at the time: '*What happens to a fast Cadence ecosystem; pieces of technology that are complex and hard to release quickly? — it's much of a challenge.*"

Our empirical findings provide insights for practitioners and academics. For example, academics now have empirical evidence of why ecosystems that release fast and often may opt for the cycle with intermediary releases; meanwhile, different projects follow different release models dependent on their goals. Therefore, to answer the question asked to Dr. Dittrich, we recommend our findings as a guideline to help the ecosystem towards a solution.

> *"*Release engineering *as a force multiplier" –John O'Duinn, director of release engineering at Mozilla during his keynote speech at ICSE 2013.*

[56] A 2016 talk at the Microsoft research facility on "Sustaining Software Engineering Ecosystems"(`shorturl.at/moMNP`)

Release team persona. During the interview session with release team members, we observed that the release team members (release engineers [260]) collectively play an important role in successfully coordinating projects/cross-project teams throughout release cycles. Furthermore, we observed that memberships to the release team are socio-technically demanding within the ecosystem. The release team members are selected based on their experience with project/cross-project teams, the nature of their contributions across multiple projects (core-contributors), their excellent communication skills, and their willingness to serve the community.

In particular, the first OpenStack release team lead, who was also the lone member of the release team during the early days of OpenStack, declares that he worked with the Linux kernel for many years before joining OpenStack during its creation in 2010. The number of release team members subsequently grew to two before the Mitaka release and eight members during the Queens release cycle. For example, in 2015, the second person who served as the release team lead declares that:

> "*... For the Mitaka cycle, our major goal is to increase the work of automating the release process, so we no longer need to run release scripts by hand. These changes to our release process affect all OpenStack projects, Thierry Carrez and I have been working to publicize them in the community. We spoke in Tokyo, and the video and slides are available online.*" — Doug Hellmann, 2015.

Similarly, other release team members reported that they had served different communities as release engineers or core-developers before coming to OpenStack. Also, we found evidence in the IRC weekly logs that release team members are willing to transfer their knowledge to project team members. For example, when OpenStack introduced Reno, the

release team members agreed to train project team members on how Reno works and how to use it in generating release notes.

We found evidence that release team members have successfully built tools to facilitate OpenStack releases and have contributed some of those tools to the python community. Also, release team members have been training cross-project teams to use in-house developed tools to facilitate the release synchronization process. Based on their profound skill sets and experiences, which the release team members possess, we have built a persona [261–263] to describe a typical release team in a complex ecosystem (see Figure 5.13). Our findings show that the release team is pivotal to an ecosystem release synchronization. Consequently, ecosystem managers and practitioners can use the release team persona to hire/select/train the right skills into the release team.

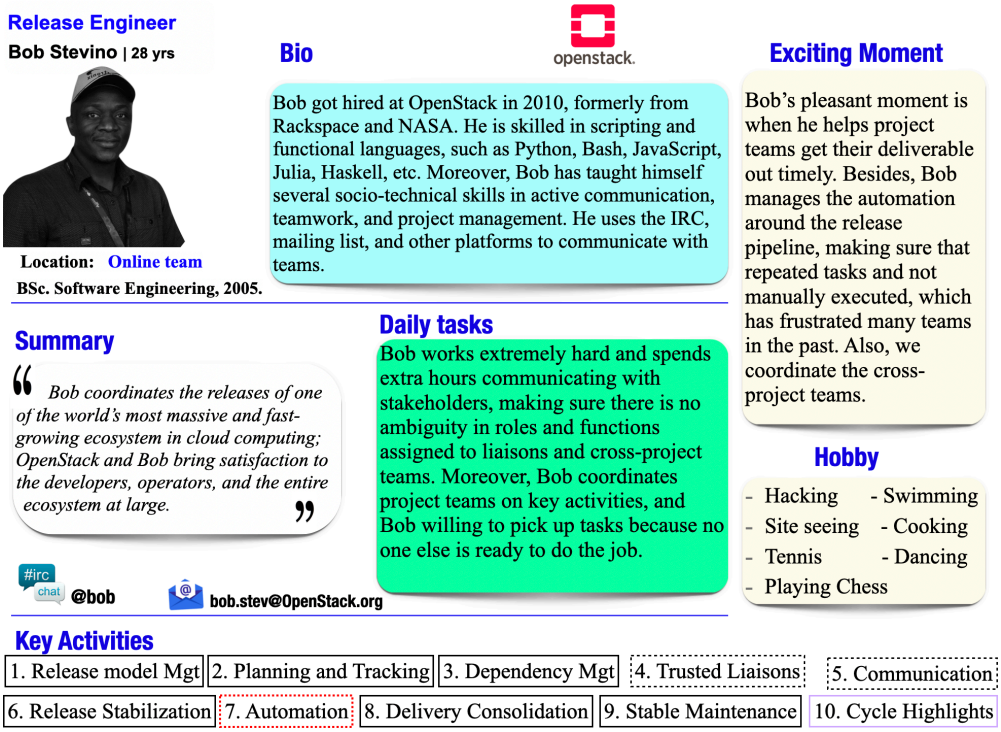
Persona is a humanized view of who a user (in this context, a release team engineer), such that we can derive valuable outcomes. In particular, creating a 'user' (release engineer) personas starts with research to understand the user (release engineer) — by observing, surveying, or interviewing the users. For details about building a persona in Software Engineering, we refer the reader to the following works [261, 262].

5.4.3 Generalization of our findings

Secondary data extraction

To generalize our findings, we extracted data from selected ecosystems that support ecosystem-wide releases such as GNOME and Eclipse. These selected communities have projects/cross-projects teams that follow one or more of the observed release strategies supported by

Figure 5.13 Release team Personas depicting common values; characteristics and skill-set of an ecosystem Release team.



OpenStack and archive of its release activities. Using their online documentations, we identify each ecosystem's structure, release strategy, release cycle, and communication medium used by the release team to coordinate releases.

Some of the online documentation that we consulted include: Eclipse[57] [58] [59] [60] and GNOME[61] [62].

Next, We downloaded the archived mailing lists of these ecosystems using custom scripts[63] and Perceval [264]. Then, we ran regular expressions on the archived dataset

[57] shorturl.at/kBHX9
[58] shorturl.at/jFGQZ
[59] shorturl.at/jrvBL
[60] shorturl.at/giBKN
[61] shorturl.at/oGJX0
[62] shorturl.at/adzIJ
[63] https://doi.org/10.5281/zenodo.4304682

to search for keywords or patterns related to the release synchronization activities that we found earlier and reported in Table 5.1. In each high-level activity, we search for the sub-themes (low-level) words, as shown in the affinity diagram in Figure 5.6. This ensures that we cover a wide range of possibilities of each activity's presence in the extracted text from each ecosystem's mailing list. Moreover, we notice that each ecosystem has its particular technical jargon to express certain concepts; this understanding guided our search of (low-level) technical words related to the release activities. When we are not sure about a particular word (jargon), we search the online documentation, and in rare cases, we email or contact release team members on either the release team mailing list or IRC channel. Afterward, we downloaded the archived mailing list coordinated by the release team for both GNOME and Eclipse ecosystem and perform our search on those archives.

For example, consider a search for the **A7. Automation and Tooling management** activity within the GNOME ecosystem using keywords such as 'script[s][ing],' 'tool[s][ing],' and autom* (for automate, automation, and automating). These keywords return search results that are related to the high-level activity. However, the results are noisy (having false positives). So, we tried different combinations of search queries until we have a meaningful result. Therefore, we randomly select the search result and examine the content. In some cases, repeated patterns of similar search words have nothing to do with discussions on high-level activities. In those cases, we filtered out those patterns and re-ran the search.

Last, we aggregate the number of messages within all threads by date into a *.csv* file. Then we apply data cleaning and pre-processing [265] on the exported *.csv* files to analyze our results (our scripts and data are provided in our replication package[64]). The results of this search could still contain some noise, which could be a threat to our result. We report this in the threat to validity section (Section 4.8).

[64] https://doi.org/10.5281/zenodo.4304682

Table 5.3: Results of manual classifications to validate the regex matching in both Gnome and Eclipse release synchronization activities (A1, A2, ..., A10.), we report the range of precision proportion (PP ± 15 %) per activity on a confidence level of 95% and confidence interval of 15 %.

Activities	**Gnome PP ±** 15 %		**Eclipse PP ±** 15 %	
A1	0.78	1.00	0.76	1.00
A2	0.85	1.00	0.85	1.00
A3	0.85	1.00	0.76	1.00
A4	0.75	1.00	0.83	1.00
A5	0.83	1.00	0.83	1.00
A6	0.83	1.00	0.80	1.00
A7	0.80	1.00	0.78	1.00
A8	0.83	1.00	0.78	1.00
A9	0.78	1.00	0.73	1.00
A10	0.83	1.00	0.80	1.00

To mitigate this noisy result, we could not manually validate all the output results. Therefore, we statistically sampled the search result using a confidence level of 95 % and a confidence interval of 15 % on all the different release synchronization activities in both Gnome and Eclipse. In particular, we manually classified and validated a median of 43 files per release synchronization activity for both GNOME and Eclipse. We used equation 5.1 to compute the precision proportion (PP) of our sampled date. The sample size (S) comes from a population of each activity. Given S, we manually validate the presence of any of the release synchronization activities. Moreover, if a given text doesn't contain any of our studied synchronization activity, we classified it as false positive (FP) or True positive otherwise. Hence, we compute PP using equation 5.1, then normalized the result between 0.00 - 1.00 and report the ranges of PP (PP ±15 %) in table 5.3:

$$PP = \frac{S - FP}{S} \pm 15\,\% \tag{5.1}$$

We might have missed an activity that other communities have, which we did not find at OpenStack. Our focus here is to extract only those activities we know about (A1, ..., A10). We report this threat in the threat to the validity section. Moreover, for future work, we plan to examine the release synchronization activities that exist in other open source communities using different communication mediums such as mailing list, IRC, Matrix, Slack, etc.

Findings

In this section, we discuss the generalization of our findings to other ecosystems, in this case, GNOME and Eclipse, and we show the results of a Time Series analysis (TS) [266] on the extracted activities in Figures 5.14 and 5.15. Time series is a technique that gleans a trend over some time (monthly in our case). Thus, the y-axis in both figures is the number of times the activity occurs, and the x-axis is the time (in months).

The purpose of this generalization is to corroborate if the release synchronization activities (A1, ..., A10) identified from the OpenStack release team communication channels are also present in other ecosystems. Then, we randomly select some mailing list discussions and follow the threads to understand how a given activity was discussed in both GNOME and Eclipse.

We use TS with the "Additive Model" to analyze the data. Using the "Statamodels" library in Python. This enabled us to observe and interpret[65] the trend and other TS components (see Figures 5.14, and 5.15) $Y_{(t)}$:

[65] readers can consult our replication package how the different TS components were model and plotted

Figure 5.14 Time series analysis showing the trends and cyclical w.r.t equation 5.2 of release synchronization activities across GNOME ecosystem releases

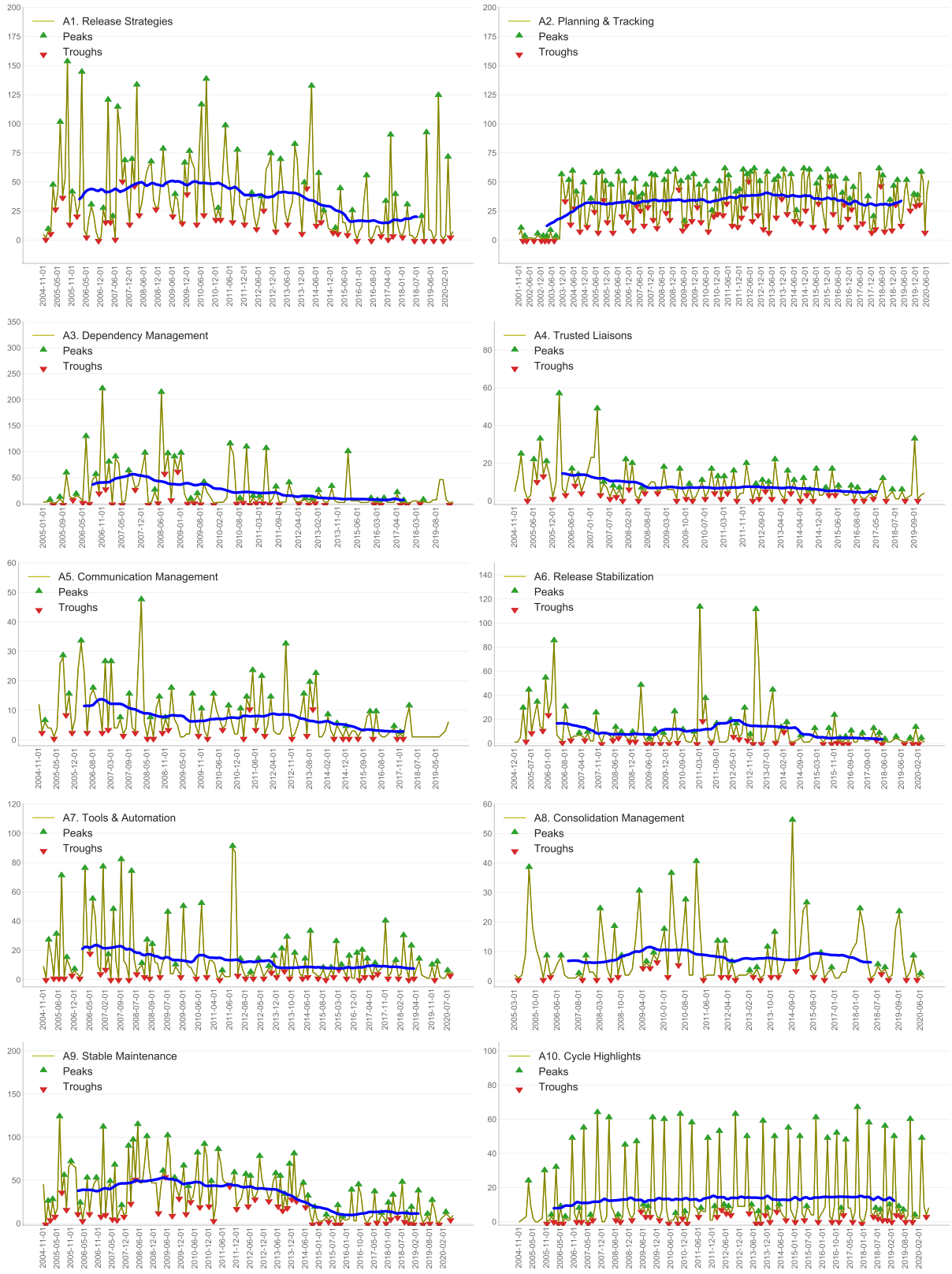
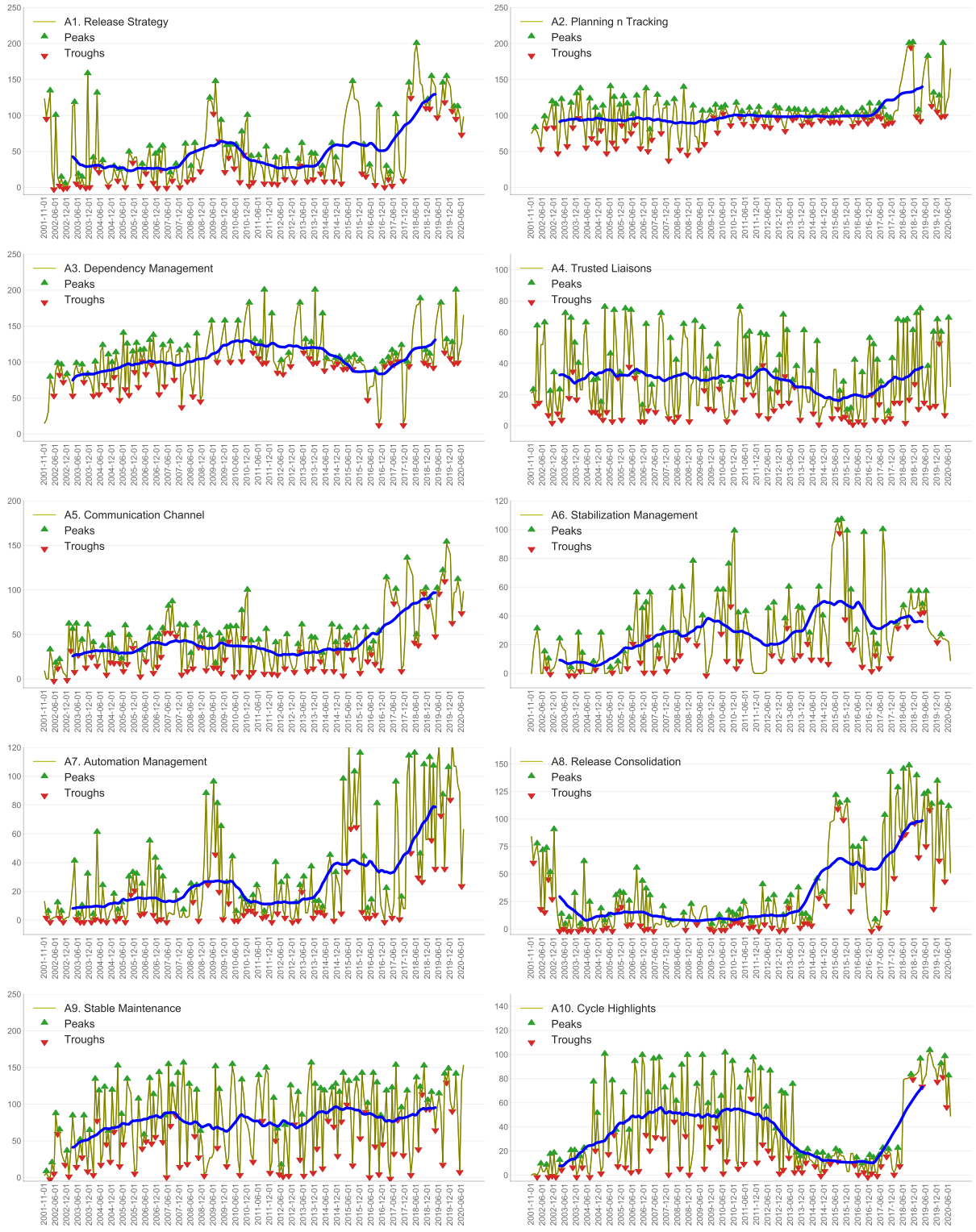


Figure 5.15 Time series analysis showing the trends and cyclical w.r.t equation 5.2 of release synchronization activities across Eclipse ecosystem releases



Trend ($T_{(t)}$): shown in blue line gives the rolling means at different time scales in our TS data.

Cyclical ($C_{(t)}$): the average value in the series that corresponds to a distinct pattern and shows periodical (but not seasonal) peaks and troughs around the trend, revealing a succession of expansion and contraction phases.

Seasonality ($S_{(t)}$): pattern occurring at specific regular or irregular intervals (it has stable variation concerning timing, direction, and magnitude).

Residuals ($R_{(t)}$) or noise, shows the difference between the observations and the corresponding fitted values. This term introduces randomness in TS model:

Equation 5.2 shows the mathematical function of the additive TS model.

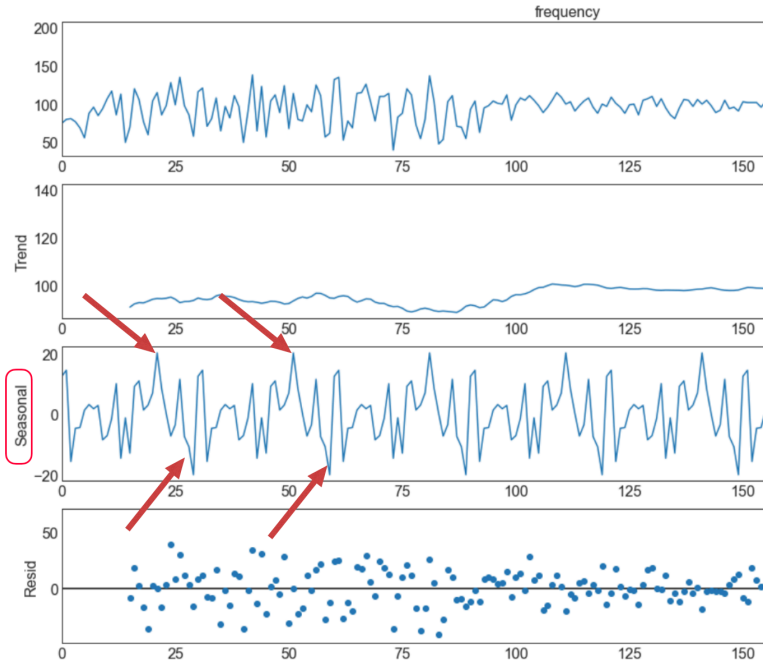
$$Y_{(t)} = T_{(t)} + C_{(t)} + S_{(t)} + R_{(t)} \tag{5.2}$$

The $Y_{(t)}$, is the output data, meanwhile $T_{(t)} + C_{(t)} + S_{(t)}$ is the predictable term and $R_{(t)}$ is the random term. All these components are important in observing TS data (trend, cyclical, seasonality, and Residual). The additive model computes the TS with the residual, as shown in Equation 5.2. The residual gives the difference between the predicted and the actual data. In particular, the residual provides a better indication of our TS model. For example, in our study, the residual was mostly due to random noise, as shown in Figures 5.16 and 5.17 (bottom-most graph). Most of the variations are within 10% variation. Our model performance was good and fitted our data well.

During the additive model training, we noticed that the variability of the peaks and troughs in our data (release synchronized activities A1, . . . , A10) were seasonal; thus, peaks

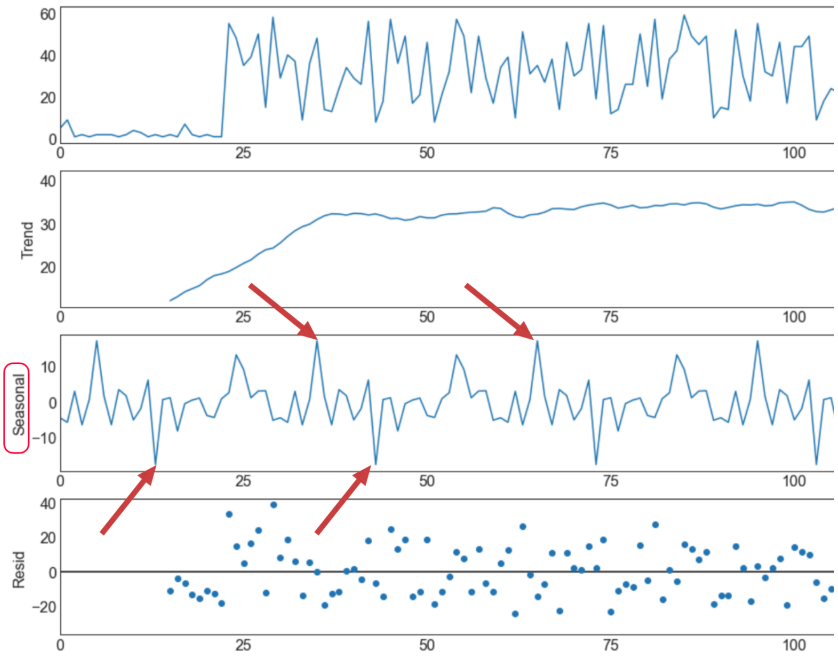
and troughs happened at predictable times. For example, in Figures 5.17 and Figure 5.16, we see that the 'seasonal' has a repeated pattern on peaks and troughs over time. The trends vary differently depending on a given synchronized activity (A1, ..., A10). We refer the reader to [267] for more details on time series analysis.

Figure 5.16 Time series analysis showing seasonality ($S_{(t)}$, repeated patterns; top arrows shows peaks and bottom arrows show troughs), but also ($C_{(t)}$), $T_{(t)}$ and $R_{(t)}$ in the additive training model for A2 in Eclipse ecosystem



Comparing how activities are mentioned/discussed in both SECOs. Since all ten activities have seasonality over time, we can say that the release synchronization activities were mentioned or discussed in both ecosystems. For GNOME (Figure 5.14), we see that although A2 (Release Planning and tracking) was rarely discussed between 2001 and 2003, it gains an almost increasing trend onward. A10 (Cycle highlights Management) was slowly introduced in 2004 and stays virtually constant; both activities were mentioned

Figure 5.17 Time series analysis showing seasonality ($S_{(t)}$, repeated patterns; top arrows shows peaks and bottom arrows show troughs), but also ($C_{(t)}$), $T_{(t)}$ and $R_{(t)}$ in the additive training model for A2 activity in GNOME ecosystem



less initially and maintained a constant flow over time. On the other hand, the trends of the other GNOME's activities decreases toward the end. A3 (Dependency Management) and A4 (Network of Trusted Liaison) had their maximum variation between 2005-2007 and started decreasing steadily. Besides, A1 (Release Model Management) and A9 (Stable Release Management) had a significant decreasing trend from 2013 onward.

Meanwhile, for the Eclipse ecosystem (see Figure 5.15), the activities had more of a fluctuating trend. However, we observe a remarkable turning point in release activities from 2016-2017; most activities slowed down in late 2016 and started rising again in 2017, except for **A2** (Release Planning and Tracking) that maintains a constant flow over time.

We found evidence that both GNOME and Eclipse have ecosystem release and a central release team that coordinate their respective releases. Moreover, we also found evidence of the studied release synchronization activities (A1, ..., A10) at OpenStack been discussed or mentioned in the context of GNOME and Eclipse. Interestingly, in both ecosystems, these activities have seasonality at particular intervals, which suggest that the activities are discussed following the release process of each ecosystem.

In both cases (GNOME and Eclipse), these trends suggest that release synchronization activities are well present and discussed in both ecosystems in different proportions (variation), similar to OpenStack's case. Therefore, we recommend future works to investigate how the release synchronization activities are coordinated in other ecosystems such as Eclipse and GNOME.

5.5 Threats to validity

This section discusses the limitations of our empirical study, following the standard guidelines for empirical studies [268].

Threats to construct validity relate to our findings' meaningfulness due to errors in our measurements. Given that the weekly meeting logs and Etherpad documents are explicitly linked and contain a full history of events, we do not suspect any physical errors in this data. Furthermore, since the OpenStack release team's procedures are strict, the meeting logs and Etherpad documents are accurate and precisely reflect what is discussed or worked on during weekly meetings.

We use regular expressions (regex) to search for themes in the archived emails in both the GNOME and Eclipse ecosystem. Using regex could be noisy and subject to erroneous results (false positives). Though we manually verify a random sample of the search results, there is still noise (false positives) in the remaining large number of search results that we could not manually verify. To mitigate this threat, we make sure that we use specific keywords from those false positive matches to strengthen our search query whenever we encounter a false positive match. We add filtered (PIPE or nested search query) that will further eliminate false positives.

We might have missed other relevant high-level activities present in different ecosystems but not found at OpenStack. To mitigate this threat, we consider future works to explore other release synchronization activities.

Threats to internal validity relate to alternative explanations that might explain our findings. To reduce the risk of subjectivity in open coding, both authors coded 15% of the logs together, followed by in-depth discussion and resolution of the codes. Besides, an external researcher validated the codes on another OpenStack release, which enabled the authors to calculate the IRR, and this is the typical way in which inter-rater reliability is ensured [114, 171, 185].

Threats to external validity concern the degree to which findings can be generalized to other OpenStack releases and different ecosystems. We studied two specific releases of the open-source OpenStack ecosystem and an additional release in the middle to validate our findings. However, further studies are needed to explore release synchronization activities in other OpenStack releases. We have also investigated the release synchronization

activities in different ecosystems, such as GNOME and Eclipse. We used the activities that we found at OpenStack as our baseline to generalize our findings. However, we might have limited our study from discovering new activities in other ecosystems that were not present in OpenStack. A critical element of such studies is access to release communication or the actual ecosystem release team and individual project members. Where possible, our documented activities try to separate OpenStack-specific from more fundamental concepts.

We also build a persona for a release engineer based on the data we got from the release team members during the interview and partly from their activities over the IRC chat logs. Thus, we try to generalize the data as much as possible (one-size fit all concept [269]) to represent the weekly activities of a typical release engineer at OpenStack. We might have omitted a vital value or characteristic of a release engineer at OpenStack or over-generalized their actual values. To mitigate this threat, we build the persona on the shared data that the release team members share and their average age.

Threats to reliability validity. To mitigate such threats, we provide access to the repositories and tools used in this study[66].

5.6 Conclusion

Software ecosystems are more than the sum of their parts, thanks to the socio-technical dependencies between otherwise independent projects. However, such autonomy comes at a price for the end-user since it becomes challenging to determine the core set of projects (and their versions) that should be installed. Thus, many large open-source ecosystems such as Eclipse, GNOME, OpenStack, etc., overcome these challenges by synchronizing

[66] https://doi.org/10.5281/zenodo.4304682

their projects' releases.

This chapter empirically studied OpenStack's federated release strategy by analyzing weekly IRC meeting logs for one year (two release cycles). This analysis enabled us to identify ten major release synchronization activities, which we cataloged and documented. To validate our findings, we interviewed eight experts from the OpenStack ecosystem, both the release and project teams.

In particular, given OpenStack's recent dismissal of the "cycle with development milestones" release model, there seems to be ample need to evaluate and improve existing ecosystem release models. Similarly, loosely coupled yet significant interaction between the centralized release team and the individual projects is an ongoing struggle. One release team member wondered. "Are we providing value and helpful information to people and then if we're providing some value, but they could use other information what that might be?" Perhaps inspiration could be found from the concept of social debt in software architecture, where Tamburri et al. discussed the role of software architects as community builders [270]. The challenges regarding Automation Management (A7) are more related to project and risk management, e.g., determining the value of developing new tools or migrating a tool to a more standard technology (technical debt).

Moreover, our findings provide several implications for advising practitioners and academics on the ten release synchronization activities and empirical evidence of why multiple release strategies co-exist in a complex ecosystem. Further, we build a persona for a release engineer, based on common characteristics and values, which practitioners and ecosystem managers could use to hire the right resources into the central release team. Besides, we extracted the release teams' mailing lists archives of GNOME and Eclipse to find patterns of the studied release synchronization activities. We found that we could generalize our

findings to other ecosystems that produce federated releases.

Finally, we found the unique activity is Deliverable Consolidation (A8) since this activity captures the essence of release synchronization in ecosystems: integrating artifacts of multiple projects into one central artifact. Hence, we hope to explore this activity in more depth in other ecosystems. The scientific community should also consider an in-depth analysis of how these release synchronization activities happen across different open source communities and how they relate to the **quality** of software deliverables. Studies should also be done to investigate other release synchronization activities in different ecosystems.

Chapter 6

A Grounded Theory of Open Source Cross-communities Feedback Mechanisms

This chapter considers the feedback flow of requests and responses between end users and distros towards the upstream SECO projects. It empirically studies the characteristics of feedback in order to arrive at a theory of feedback w.r.t. SECO sustainability, then quantitatively validates important parts of the theory. This chapter will be submitted to the IEEE Transactions on Software Engineering (TSE).

6.1 Introduction

Software ecosystems (SECOs) are groups of autonomous, yet inter-dependent projects working towards a common goal [271, 272] through a complex choreography of socio-technical collaborations and interactions among different stakeholders [24, 93, 94, 273]. Such SECOs can both involve proprietary and open source components, as illustrated by SECOs like Android, Linux kernel, Eclipse, GNOME, SAP, OpenStack, Apache OpenOffice, LibreOffice, Amazon Web Service (AWS), R/CRAN, Node.js/npm, Xero, Zephyr

SECOs, etc. Corporations involved in such SECOs span a wide range, from Microsoft, Apple and Google to open-source entities like the Apache, Linux and Eclipse foundations [46]. SECOs such as the Linux kernel, OpenStack, etc., produce one final product, which distros can customize for end-users. Versus SECOs involved in a large set of products of which a subset has to be chosen and installed by the end-user directly (e.g., Maven or npm).

An important subset of SECOs, featuring the likes of the SECOs of organizations like Microsoft[1], Apple[2] and Google[3], OpenStack, etc., have a so-called `closed-loop` community enabling the collection and propagation of feedback. Such feedback is essential to enhance the scope, quality and overall development process of a SECO, and can span from sending email requests for help or reporting a bug all the way to fixing said bugs or even contributing new features [46–48]. A classical closed-loop community

comprises the following components (see Figure 6.1):

1. A governing board or a foundation overseeing the policies and SECO culture [274].

2. A diverse community of contributors (either part of a company or working as volunteers) who make contributions to the SECO repositories and technical documentation — **upstream development** [273, 275, 276].

3. Companies/vendors who pick up the source code (or final deliverable) from upstream, then customize, package, and integrate the SECO releases according to different use cases to satisfy their end-users —**downstream distributors (distros)** [273, 275, 276].

4. End-users, who either adopt SECO distros to enable their daily work/business or who

[1] `https://tinyurl.com/2wej4jpy`
[2] `https://developer.apple.com/bug-reporting/`
[3] `https://techjourneyman.com/blog/google-ecosystem-explained/`

independently adopt the upstream SECO release [275, 276].

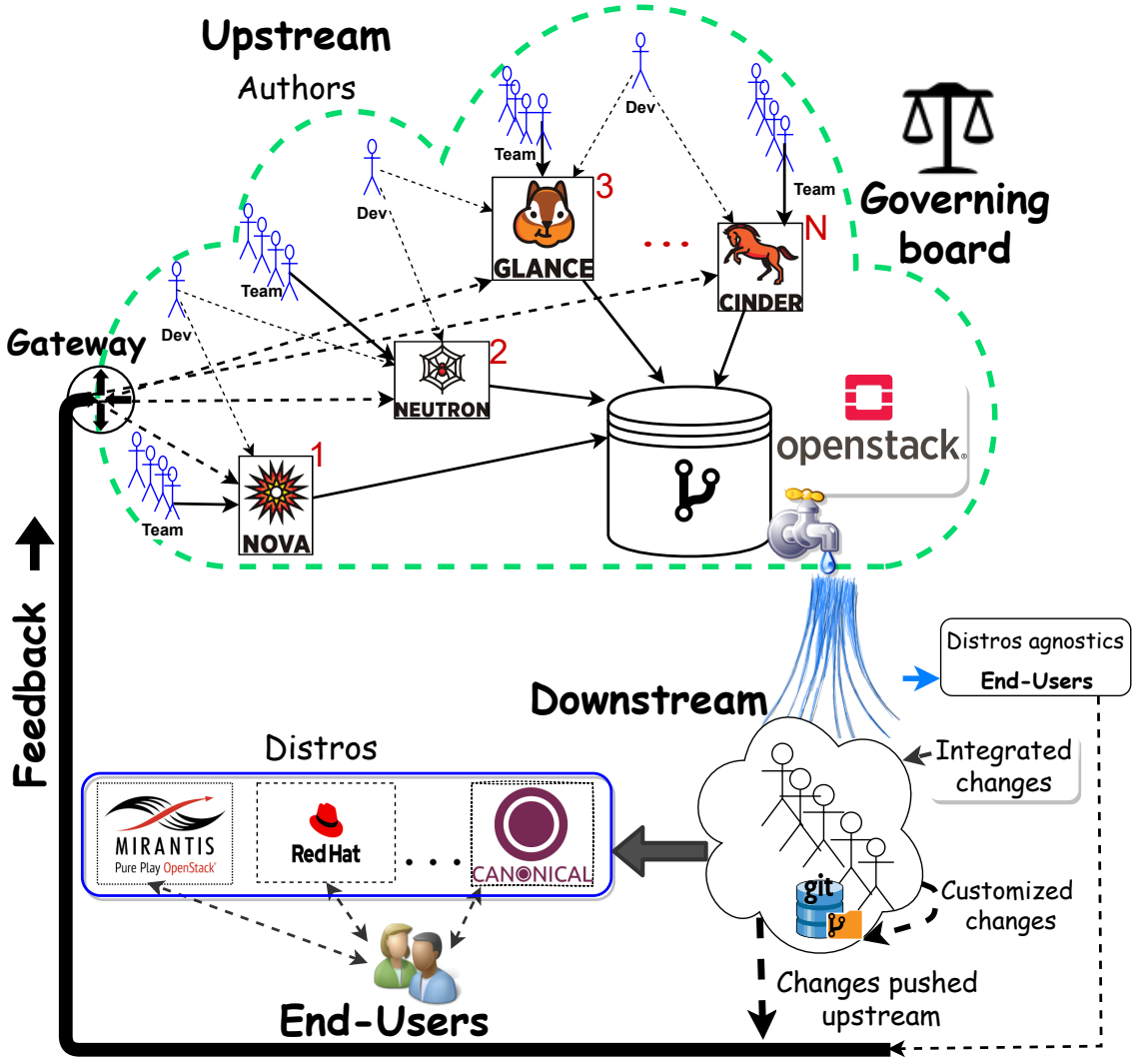
5. A feedback/closed-loop[4] mechanism that enables distros/end-users to communicate **changes** (feedback) to upstream in the form of bug fixes, new features, bug reports, crash reports, emails (with complaints or comments about a feature, request for help), end-users survey, etc [276–278].

The latter feedback mechanism is essential in enabling both open-source and proprietary communities to collaborate in various ways. For example, Jim Whitehurst, the president and CEO of RedHat advocates that *"In innovative organizations, feedback is continual and frank, in other words, it's open"*[5]. Sarah Novotny, the Open Source Lead and CTO of Azure Office at Microsoft states that *"Open source has ... become an industry-accepted model for cross-company collaboration"*[6]. Such collaboration and feedback takes on many different forms, such as organized events (in-person or remote), communication on shared media channels, contributions to issues trackers and code bases, etc., and they create rich links and interactions across SECO communities.

SECOs face major challenges in coordinating/prioritizing feedback across their subcommunities of end-users and distros. Prior studies show that software maintenance, especially within ecosystems and distributions [99, 279–281], causes important issues. For example, Boisselle et al. found that by optimizing the bug fixing process, SECOs can minimize time wasted on duplicate fixes and also reduce integration cost, while Adams et al. found that synchronization with upstream is still a major concern and "requires rigorous book-keeping". Additionally, the rapid growth of SECOs poses a different range of challenges to its distros and end-users in terms of integrating and customizing upstream

[4] https://www.shopify.ca/enterprise/mastering-feedback-loops
[5] https://tinyurl.com/jpmxy7kk
[6] https://tinyurl.com/7t7kpn4x

Figure 6.1 Feedback Loop of OpenStack SECO enables cross-community collaboration between Upstream and Downstream communities



changes. For example, distros need to rapidly integrate upstream changes, while also ensuring that the changes do not break existing end-user applications. At the same time, the number of communication channels within a growing SECO in the worst case grows quadratically [282], unless explicit feedback platforms are provided and enforced to support discussions between the different distros and end-users. Therefore, for a SECO's projects to be sustainable and to grow healthily [25], distros and end-users should be able

to provide timely and meaningful feedback to the SECO community.This chapter performs a mixed-methods empirical investigation involving grounded theory and quantitative study on the large OpenStack ecosystem in order to better understand effective coordination in the closed-loop feedback mechanisms between OpenStack's downstream distros and the upstream development community. In particular, our qualitative study uses the Straussian grounded theory methodology guided by the following research question:

RQ1: ***How do SECOs sustain cross-community collaboration****?*

We address this RQ using unstructured interviews, surveys, and focus groups with five, 21, and 21 participants, respectively, followed by qualitative analysis of 120 mailing lists archived.The outcome of our qualitative analysis elicits a theory of cross-community SECO feedback in terms of the diversity of feedback and the importance of communication and synchronization to drive a sustainable SECO. A theory grounded in data with 15 constructs that explains the diverse feedback mechanisms (2), characteristics (2), challenges (7) and the benefits (4) of cross-community collaboration. Next, we aim at quantitatively validating parts of this theory through the following research questions:

RQ2: ***How diverse is the feedback contributed by distros****?*

RQ3: ***What roles do distros play in synchronizing SECO feedback****?*

By quantitatively mining 140,361 mailing list archives, 21 distros' issue trackers containing 142,914 filed bugs, 65,179 resolved bugs, and 4,349 new features over ten years across 20 SECO releases, we make the following contributions:

- Distros are important producers of feedback within a SECO, contributing 25.6% new features, 30.7% emails on technical topics, 44.3%, and 30.7% bug reports and resolutions, respectively.

- 80% of all distros' feedback are contributed by 20% of distros, which follows a Pareto distribution.
- We detected four feedback communities in the SECO using social network analysis. Using centrality algorithms, we found seven influential distros in the email feedback community, two in the new features community, one in the filed bugs community, and one in the resolved bugs communities .
- Influential distros have higher truck factors ($\geq$3), which are greater than common open-source truck factors.
- Surprisingly, influential distros are not the most dominant in their respective feedback type communities; they play general-purpose roles in synchronizing feedback.

Among others, our empirical study has the following relevant implications for academics, Distros, and SECOs:

1. The identified theory of SECO feedback mechanisms calls on both SECOs and researchers to further explore these notions and to build tools for sustainable SECOs.
2. By leveraging our theory and quantitative findings on diversity and synchronization of feedback, SECOs and distros can benefit from our findings to promote a more sustainable community.
3. Finally, the quantitative findings, and our methodology to gather and analyze these, allow distros to actively measure and promote horizontal collaboration among distros and upstream.

6.2 Background and related works

6.2.1 Subject system selection

This chapter empirically investigates cross-community feedback mechanisms and their significance in the context of a complex open-source SECO. Therefore, it is necessary to have a SECO with the following characteristics:

- growing upstream community
- wide range of downstream/distros community
- open-source, allowing access to any data related to the actual feedback mechanism
- archived communications between upstream and downstream
- well-documented process of the SECOs' internal and external processes

Among several candidate SECOs such as Linux kernel, Apache, OpenStack, Eclipse, etc., OpenStack satisfied the above criteria, becoming our candidate of choice. Below, we provide more background on OpenStack, and we also discuss major related work.

6.2.2 About OpenStack

OpenStack[7] is an open-source SECO for private, public and hybrid cloud computing founded jointly by NASA and Rackspace in 2010 and licensed under Apache 2.0. It follows a rapid 6-month release cycle. As of July 2021, OpenStack is currently developing its 24th release, Xena[8], by a community of over 100K members dispersed across 182 countries, and featuring both volunteers and over 600+ companies [272]. The code base itself has over 20M

[7] https://openinfra.dev
[8] https://releases.openstack.org

lines of code, with 60+ core projects, and over 500 development (sub-project[9]) teams[10] making changes to about 1,283 Git repositories[11] (one per sub-project)

The OpenStack community constitutes both an upstream community and a hybrid (open source and vendor) downstream community of distros. Upstream development is managed by the Open-Infrastructure Foundation (OIF) and follows a rapid six-month release cycle [283]. OpenStack cloud end-users include government agencies (such as the military, civil services), financial institutions, research institutions, telecommunication industry, healthcare, space agencies, manufacturing/industrial sectors, retail, media/entertainment, information technology, etc.

Given the wide range of end-users and their different use-cases, it is impossible for the OpenStack upstream community to create releases that can cater for all types of end-users. Instead, an intermediary layer of distributions (distros) has popped up that systematically track new upstream releases, then, integrate, customize and deliver high-quality services to these end-users, similar to how different mobile device vendors like Samsung or HTC customize new releases of Android for their product teams [284]. Distros may also want to provide/request changes to/from upstream or another distro community.

Usually, the interaction of changes and feedback between upstream and distros (and their users) will have to go through a dedicated channel or mechanism to their destination. However, neither the scientific community, nor industry have a good grasp of which feedback mechanisms are common between upstream and downstream, or of their challenges and benefits to different OpenStack stakeholders.

[9] https://tinyurl.com/4z32mp47
[10] https://tinyurl.com/4jkpv2em
[11] https://tinyurl.com/5xkdjwf5

6.2.3 OpenStack community events

The principal means of communication at OpenStack are mailing lists (for offline communication) and the Internet Relay Chat (IRC —an online text-based chat system that provides synchronous communication among multiple users simultaneously.)[12].

To nourish deeper interactions between the various (sub-)communities, the OIF also coordinates activities and organizes events at various geo-locations throughout the year. Early works [272] have studied OpenInfrastructure Summit, Forum, and the Project Teams Gathering (PTG) events in the context of SECO release synchronization. However, other events foster cross-community collaboration, such as OpenDev events, Meetups, and the OpenStack and OpenInfra days that bring together stakeholders and target different audiences with specific purposes and outcomes. For example, OpenDev brings together the OpenStack SECO's upstream developers and end-users to share best practices, identify opportunities, and advance the state-of-the-art in OpenStack cross-community. Typically, OpenDev is a series of three virtual independent events; each event covers unique topics in a discussion-oriented session focusing on cross-collaboration to explore challenges, sharing common architectures, and collaborations around potential resolutions.

Also, OpenStack and OpenInfra Days bring cloud experts and information technology executives, cloud operators, and distros to discuss open infrastructure use cases. Furthermore, Ops Meetups[13] are a type of event that brings together OpenStack operators to share best practices, ideas and give feedback on diverse topics during working sessions. They are mainly organized as round-table discussion sessions, with only few longer presentations.

[12] https://tinyurl.com/4fme5dx7

[13] https://wiki.openstack.org/wiki/Operations/Meetups

6.2.4 Related work

Grounded theory methodology (GT):

Strauss-Corbinian GT is gaining grounds in software engineering research as a common methodology for grounded theory analysis [82–90], due to its rich intersection of the social and technical dimensions.

For example, GT has been used extensively by Masood et al. [85], Shastri et al. [90,91], and Hoda et al. [92] to study agile software development. Masood et al. collected data from three sources: interviews, participant observation, and scrum guide (*by the book*) and found variations across three core categories between Scrum theory and how it is practiced. Hoda et al. use GT to investigate how Agile teams self-organize, with members repeatedly self-organizing themselves and functioning outside their boundaries, with one person playing multiple (different) roles at different time intervals as needed. The authors use interviews and participant observation to collect data from 58 Agile practitioners across 23 software organizations to generate a theory, which identifies how informal, implicit, transient, and spontaneous roles make Agile teams self-organized.

Palomba et al. [84] did a mixed-method study to understand developers' concerns and how those affect their decisions to eliminate or preserve code smells. The authors use Straussian GT qualitatively with data from a survey to study community smell and quantitatively, and mined nine open source repositories to show that community smells can influence code smells' severity.

In particular, our approach of GT is greatly inspired by Masood et al. We did a mixed-method study with an open source SECO upstream and a mixture of downstream communities to understand cross-community collaboration and feedback mechanisms. GT allowed us to focus on exploring how cross-communities collaborate to drive feedback mechanisms

in a complex SECO and to identify four benefits and seven challenges.

On the other hand, Rodríguez et al. [88] take a different epistemological stand, by using a different variant of GT (Glaserian version —positivist approach) to understand the concept of value-based feature selection mainly when used in the software development process to select and prioritize features based on their values. The authors collected data from three large-scale software organizations practicing value-based decision-making using semi-structured interviews with 21 decision-makers. Their resulting analysis led to a theory "of value for value-based feature selection (software features) in software/software intensive product release planning" of value-based feature selection in software intensive product release planning thst is explained by six constructs.However, Glaserian GT is not suitable for our **socio**-technical activities because of its strong positivist stand.

Collaboration at open source software communities:

Prior works [93–95] have studied collaboration in open source such as OpenStack SECO, WebKit communities, etc., along various dimensions and levels of abstraction.

Zhang et al. [96] empirically studied upstream commercial participation in OpenStack projects, by mining projects' Git repositories and conducted a survey with OpenStack developers. Authors found eight contributions models that drives competing companies motivations to OpenStack, and argue that while competing companies contribute over 90% of commits to upstream as compare to volunteers, these companies can still collaborate while maintaining a balance between the SECO's long-term sustainability goals and maximizing business profit. The authors stated that a majority of these companies are hardware manufacturers such as Dell, Intel, etc., and other e-commerce platforms like Walmart, eBay, etc. Similar to our work, Zhang et al. found that distros ("full-solution providers") are

key players among the eight models that define companies collaborate scheme with upstream projects. On the contrary, they reported that a single company contributes over 50% of commits to upstream projects, which contrast our finding and previous works [93, 97] and the upstream policy. Therefore, we call on the authors to verify this claim. Both Zhang et al., and Teixeira [94] use social network analysis to show collaborative patterns in OpenStack's upstream, and both lines of work agree that competitive companies can still collaborate upstream in the same project.

Our work differs from Zhang et al. in a number of major ways. Firstly, they focused mostly on upstream collaboration during the development cycle instead of considering significant roles that distros play in contributing feedback to upstream. We explore a wide varieties of data sources and venues to draw our conclusions. . We also found and studied different types of collaborations (upstream-upstream [93], upstream-downstream and downstream-downstream), of feedback provided by distros and of feedback mechanisms used to collaborate . That said, Zhang et al.'s work provides an excellent base and inspiration for our work on SECO feedback mechanisms. Both works agree that dominating companies risks the sustainability of a SECO if abandon.

Cross distribution/projects bug:

Cross distribution [95, 98] SECOs are becoming increasingly popular , for example, in the Linux [99], Android [100], and OpenStack [95] distributions, etc., such SECOs and the fact that they keep on growing leads to various cross-distribution issues, some of which have been studied in literature Boisselle and B. Adams [99] quantitatively analyze cross-distribution issue report duplicates across the Ubuntu and Debian bug repositories. Using the within-project approach to detect duplicate in the context of cross-distribution duplicates, and obtain an optimal precision and recall of 43% for an 88% threshold. The

authors estimate the amount of wasted time that contributors/users lost waiting for a fix that already exist: a median of 38-47 days.

Ding et al. [101] empirically analyze the scientific Python ecosystem (upstream/downstream cross-project bugs) to understand the distros' workaround characteristics of upstream fixes. The implications of this study guide practitioners to understand cross-project bugs in software ecosystems. Results show that workarounds have four patterns, three categories of workarounds bugs exist, and upstream fixes are significantly different in terms of LOC and code structure.

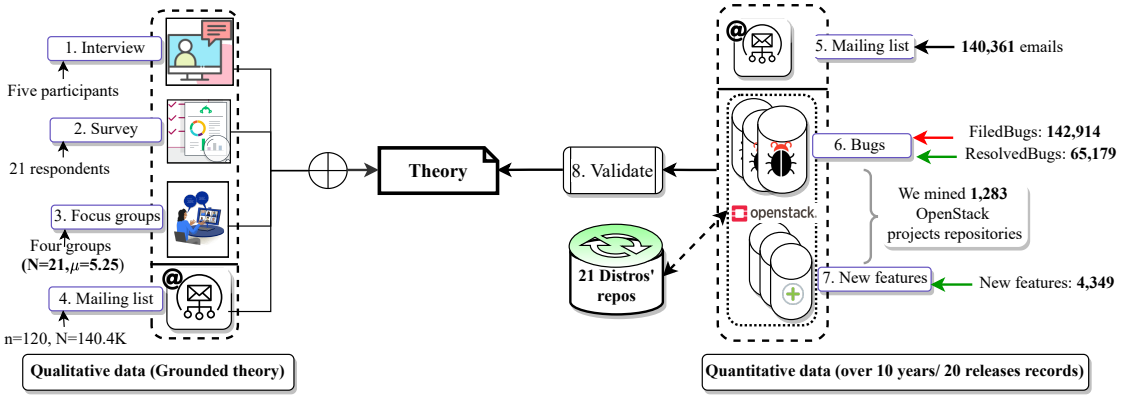
Our study found evidence of distros collaborating among themselves in reduced form and we also show that wasted efforts are pain-points to distros involved in cross-community collaboration, as it takes an extra fix time of 13 days to discover a redundant bug.

6.3 Qualitative analysis

In order to investigate cross-community collaboration and feedback mechanisms in a typical open-source SECO, we use both qualitative and quantitative research methods.Using a grounded theory (GT) approach, see Fig. 3.2, we address RQ1 (**RQ1:** ***How do SECOs sustain cross-community collaboration?***) by exploring the "**How**" of cross-community collaboration and feedback mechanisms, both in the vertical (interaction between distros and upstream communities), and horizontal (amongst distros) dimensions. Grounded theory (GT) is a methodology that aims to generate a 'theory' entirely from data collected and analyzed simultaneously until a theory grounded in data emerges that reveals social phenomena such as relationships, social processes, etc. GT comes in different variances depending on the ontological or epistemological stance [285].

We use the **Straussian** (**Strauss**-Corbin**ian**) variant [286] of Grounded Theory (GT) [287]

Figure 6.2 Methodology for qualitative and quantitative analysis.



adapted to software engineering [82–85, 88, 89, 92]. This variant focuses on the rich intersection of both the social and technical (socio-technical) nature of software engineering, enabling us to understand the significance of a diverse and inclusive cohort of different stakeholders (software developers, teams, community/foundation members, managers, customers/end-users, distros, etc.) and their interactions in a complex cross-community SECO. Furthermore, it allows to study how these stakeholders' socio-technical characteristics (motivations, needs, preferences, strengths, limitations, skills, etc.) enable them to collaborate and produce more ethical and human-centered software artifacts.

Straussian GT follows a rich blend of symbolic interactionism and pragmatism approach [82] suitable for socio-technical exploratory research process that makes no assumption on any prior theory to test on data, but generates new theories entirely from data. The below sections go in further depth regarding this approach.

6.3.1 Data collection

Theoretical sampling and saturation

Step 1: To bootstrap our study, we reached out to an OIF board member (indicated in Table 6.1 as $Up5$ with Role BM) with expertise in OIF customer relations and community

Table 6.1: Surveyed and focus group participants (D1-21), and upstream members, comprising Foundation members (Up1-5) and moderators (Mod1-3).

ID	Community	Role	Yrs($_{Com}$)	Yrs($_{Role}$)
$Dn1$	Downstream	Mgmt	5	4
$Dn2$	Downstream	Engr.	7	5
$Dn3$	Downstream	Engr.	8	6
$Dn4$	Downstream	Mgmt.	11	5
$Dn5$	Downstream	Engr.	7	4
$Dn6$	Downstream	Engr.	9	5
$Dn7$	Downstream	Engr.	9	6
$Dn8$	Downstream	Engr.	4	4
$Dn9$	Downstream	Engr.	6	5
$Dn10$	Downstream	Engr.	8	5
$Dn11$	Downstream	Mgmt	9	5
$Dn12$	Downstream	CustRel	7	5
$Dn13$	Downstream	Engr.	8	5
$Dn14$	Downstream	Engr.	6	5
$Dn15$	Downstream	Engr.	7	5
$Dn16$	Downstream	Engr.	9	5
$Dn17$	Downstream	CustRel	11	7
$Dn18$	Downstream	CustRel	9	5
$Dn19$	Downstream	Engr.	7	5
$Dn20$	Downstream	Engr.	6	5
$Dn21$	Downstream	CustRel	7	5
$Up1$	Upstream	TC	12	6
$Up2$	Upstream	TC	11	5
$Up3$	Upstream	TC	11	7
$Up4$	Upstream	TC	12	8
$Up5$	Upstream	BM	7	5
$Mod1$	Upstream	FM	5	3
$Mod2$	Upstream	Mgr.	10	6
$Mod3$	Upstream	Mgr.	11	7
$ML_{(1..120)}$	Mailing list	Labeling	-	-

Downstream/Upstream participants (Dx/Upx) with #years of experience with their community Yrs(com) vs. in their various roles Yrs(Role).

services to get an initial understanding of how distros and the upstream community collaborate to form a complex SECO. As suggested by this board member, the author of this

thesis then contacted the OIF vice-president for an exploratory online unstructured interview (1.Interview, Fig. 3.2), which lasted 18 minutes. Towards the end of this interview, the author of this thesis asked the vice-president to recommend potential members in the SECO who could be resourceful to our study (snowball sampling).

Four technical committee members (indicated in Table 6.1 with role TC) were suggested, which we contacted over the IRC channel for an unstructured online interview. While we did not have permission to record the discussion, the TC members allowed us to write down notes and use the IRC logs as textual communication data. These exploratory online discussions lasted between 10–18 mins each, and enabled us to have a high-level overview of various vendors/distros that operate within the OpenStack ecosystem. We aimed at learning how the communities around OpenStack function, for example, if there are policies that govern distros' interactions with the upstream, who are the contact persons for each distro, etc. We also asked the following open-ended questions[14]:

1. How is the downstream community organized?

2. How do upstream and downstream collaborate?

Step 2: Next, the TC members gave us indications and resources based on which we contacted 35 distros for a survey study, 21 of which responded, yielding a response rate of 60%. For this survey study (2.Survey, Fig. 3.2), we contacted each distro to understand their demographics, size, interactions and collaboration concerning the upstream in terms of the kind of feedback provided and the mechanisms used. The survey instrument contained both open- and closed-ended questions. The survey instrument is available in our replication package [288]. Examples of questions asked:

1. How do you proceed to integrate upstream changes?

[14] shorturl.at/crCD0

2. how do you manage these type(s) of changes locally?

Step 3: While analyzing the survey data, we realized that we needed more insights in cross-community collaboration, for example, who is collaborating with who in the community? What type of collaboration takes place? Do distros associate with other distros to solve complex problems, such as bug fixes, new features, bug/crash reports? Is there a mechanism to keep track of distros' contributions? These sets of questions guided our choice for using a focus group study [289, 290] with the surveyed distros.

The focus groups were held during the virtual **PTG** event on October 18-22, 2021 during the Covid-19 pandemic. To accommodate focus groups with all 21 surveyed distros, we simultaneously collected and analyzed data in four different stages [85] in four distinct groups (3.Focus groups, Fig. 3.2) and at different times. Each such focus group study involves participants with a high level of involvement in the community and experts in cross-community collaboration and OpenStack-related operations.

The questions for our focus groups in the scope of cross-community collaboration and other complex topics (e.g., end-user participation) had been pre-approved by OpenStack, and include:

1. Can you describe your recent experience when you collaborate (upstream/downstream) on a feedback?

2. How do you cope with the upstream release cycle?

We had an allocated time slot of 70 mins per day starting from Monday, Oct. 18 through Thursday, 21, 2021, a total of 280 mins. We randomly assigned participants to a particular group with a median size of five participants. To moderate the focus groups, OpenStack assigned two members with managerial roles in the community to assist the author of this thesis.

While conducting the focus groups, we continuously refined our sets of questions. We also continuously observed the focus group participants, and did not notice any participant dominating the discussion or lurking; they were all engaged and proactive in the discussions. At the end of each focus group discussion, OpenStack's virtual communication environment automatically generates a transcription of the discussion. Since this was not perfect, we had to make minor corrections due to different speaker accents.

After finishing the fourth round of focus groups, we realized that we were not learning any new concept/theme/category from the previous groups or data we analyzed. As such, we recognized that we had reached a point of saturation and stopped focus group activities.

Step 4: During the focus groups, about 90% of the participants mentioned that they use the mailing lists the most to discuss bug fixes, crash reports, new features, events, etc., with the upstream community. Hence, we considered the mailing lists as an essential source for our study, and decided to also mine the "openstack-dev" mailing list (4.Mailing list, Fig. 3.2) archives from April 2012 (the first six-month release of OpenStack, Essex, and around when OpenStack established its foundation) to October 2021 (Xena release, when we conducted the Survey and Focus group study). Since there are 20 releases between Essex and Xena, we assigned the mailing list threads in each release's cycle to a separate bin, then randomly selected six email threads in each bin, then coded, analyzed and compared those to each other. We iteratively continued this process (random selecting/coding/analysis) until we reached saturation [286], which in our case was after 120 manually analyzed mailing list archives[15] from a population of 140,361 emails with threads.

The author of this thesis conducted the unstructured interview, and was one of the moderators of the focus groups in collaborations with OpenStack foundation members.

[15]Emails were extracted with their threads in a single text ("*.txt" file, so if a particular mail has 10 threads, all the threads will show in a single (.txt) page.

6.3.2 Data analysis

Straussian GT emphasises three methods for data analysis, explained in detail in the following subsections, i.e., **open**, **axial**, and **selective coding** [286,291]. For each of these, all three authors of this chapter did ten rounds of qualitative analysis before reaching perfect agreement. Each round started off with the individual coders independently (re-)coding their sample of the qualitative data obtained from the unstructured interviews, survey, focused groups, mailing list data and memos [82,287] (*to discover patterns and relationships that exist among concepts, sub-categories, and categories*) based on our code book available online [288]. This codebook was generated from the **Nvivo**[16] application that we use to facilitate the process of labeling/coding of the qualitative data.

The results were then compared to each other, and discussed in online meetings, during which the emerging codes, concepts, sub-categories, categories and theories are further refined. This process continuously increased the level of abstraction until a theory emerged from the data [86] based on the **characteristics** and **types** of cross-communities collaboration, and the associated **advantages/challenges** of cross-community collaboration and feedback.

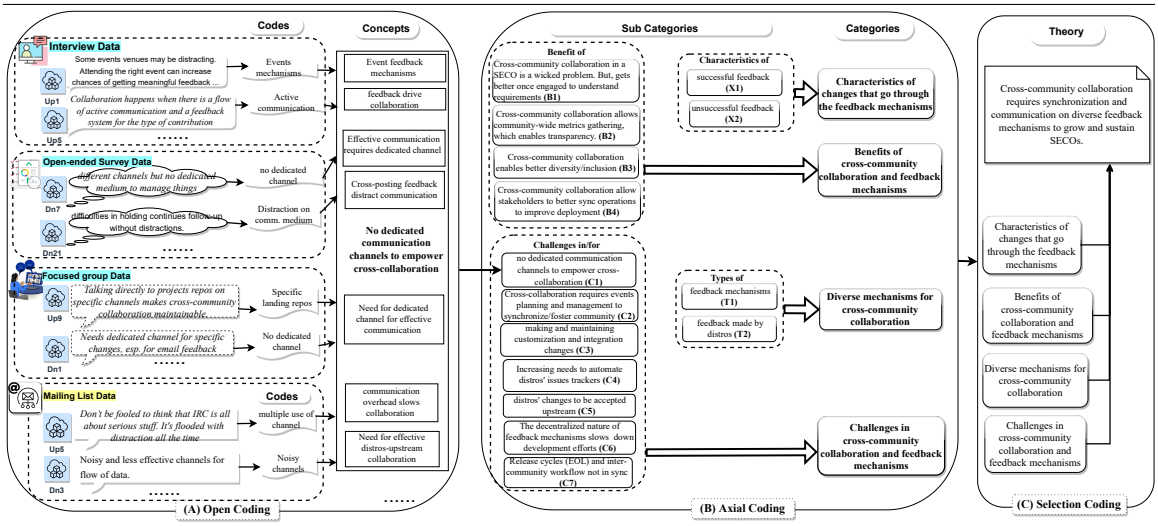
Fig. 6.3 (C) visualizes the resulting theory. To facilitate the authors in connecting and working on the same platform remotely and asynchronously, we used Miro[17].

Open coding

Our open coding [286] analyzes the transcripts of the unstructured interviews, survey, focus group and sampled email data to label occurrences of events, actions, interactions, etc. relevant to cross-community collaboration. We used four guidelines in this open coding

[16] https://tinyurl.com/4pbwrhmt
[17] https://miro.com/app/dashboard/

Figure 6.3 Summary of GT data analysis, showing how (A) **Open coding**, (B) **Axial coding**, and (C) **Selective coding** generate a theory of how cross-community collaboration drives feedback mechanisms for SECO growth and sustainability. *(See enlarged figures in appendix .2).*



phase: (i) we asked specific and consistent questions to the data during coding (for example: What is happening here? Under what conditions is this happening? What is the study of this data about?); (ii) we maintained a well-defined manner of labeling the text based on an inductively derived code book, (iii) we actively reflected on the coding process, and we wrote down notes (memos) to keep track of our reflections; (iv) we minimized our assumptions by being as open-minded as possible [286, 291].

Fig 6.3 (A) shows a concrete example of open coding of the interviews, survey, focus group and archived email data that we coded. First, we extracted keywords at the lowest level of granularity and converted them into codes, e.g., *'Event mechanism'* and *'Active communication'* from the interviews with participants Up1 and Up5, and *'No dedicated channel'* from the survey and focus group with participants Dn1 and Dn7, respectively. These codes are then mapped to their associated concepts, e.g., *'Event mechanism'* maps to *'Event feedback mechanisms'*. The same kind of coding iteratively is performed on the

four data sources, leading to a final set of codes and concepts.

Axial coding

This coding builds on the open codes and concepts by moving and re-arranging codes and concepts into sub-categories, then to establish links (relationships) between sub-categories, grouping them into higher-level categories, as shown in Fig. 6.3 (B). To facilitate the linking process, we use four guidelines: (i) we look at the **condition**s under which concepts occur , (ii) we were aware that a label might fit under multiple themes/concept/categories, although under different **context**s, hence we used this context to guide how we do the linking of concepts and/or sub-categories; (iii) we also consider the **consequences** if we link sub-categories and categories; (iv) finally, we consider what action and interaction **strategies** a link reveals.

Indeed, the actual coding builds conceptual families using a bottom-up approach to form a coding paradigm [292] by continuously comparing new data to existing concepts, sub-categories and categories to obtain a better fit to the identified relationships and groupings. Furthermore, whenever a new category emerges, we review the literature to compare this emerged category to existing categories. Also, we use sensitizing concepts [292] making sure that every concept is provisional as we tested different configurations by continuously moving concepts/categories while building a theory from the data.

Eventually, 15 sub-categories were derived from the 188 concepts, and were linked to four categories established during axial coding, as shown in Fig. 6.3 (B). These categories explain the types of feedback mechanisms (**T1 and T2**), the characteristics of feedback requests (**X1 and X2**), and the potential benefits (**B1, .., B4**) and associated challenges (**C1,**

..., C7) of doing cross-community collaboration. For example, "Cross-community collaboration enables transparency in measuring community-wide metrics" was mapped to the category "Benefits of cross-community collaboration and feedback mechanisms,". Likewise, "*The decentralized nature of feedback mechanisms slows down development efforts*" was mapped to the "Challenges in cross-community collaboration and feedback mechanisms' category.

Selective coding

This last coding phase formalizes the relationships between categories into a theory, which becomes our core category [286, 291]. Then, we consolidate all other categories under this core category through a use case involving all categories. As shown in Fig. 6.3 (C), our theory states: "*Cross-community collaboration requires synchronization and communication via diverse feedback mechanisms to grow and sustain SECOs.*"

Our memo notes [92], which we use to advise our data analysis , were instrumental in supporting the relationships among sub-categories, categories, and core category with empirical evidence, grounding our theory in the data.

6.3.3 Qualitative results analysis

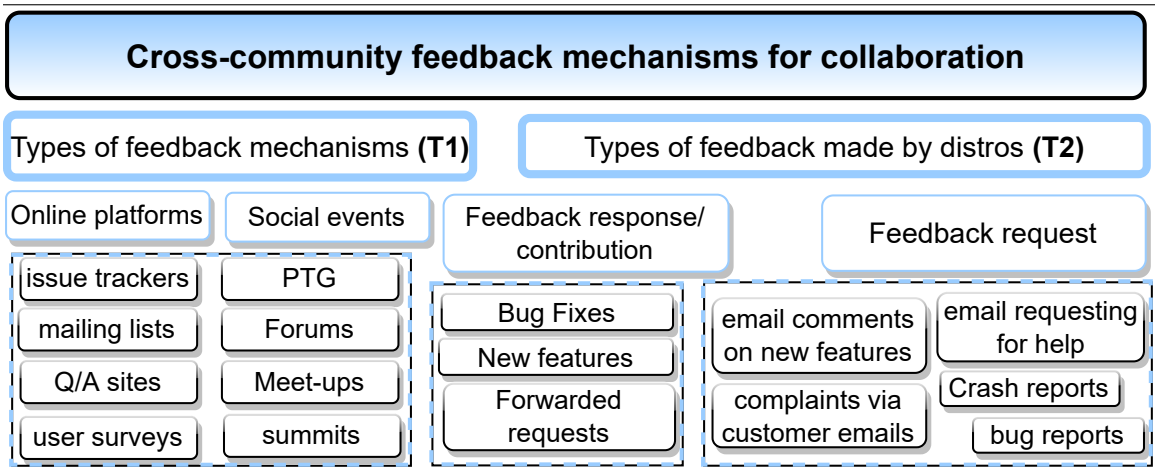
RQ1: ***How do SECOs sustain cross-community collaboration?*** Our GT answers this RQ via the following derived theory:

> *Cross-community collaboration requires synchronization and communication via diverse feedback mechanisms to grow and sustain SECOs.*

This section discusses each of the 15 sub-categories of Fig. 6.3 (B) grouped by the 4 high-level categories they belong to.

Diverse mechanisms for collaboration

Figure 6.4 Types of feedback mechanisms.



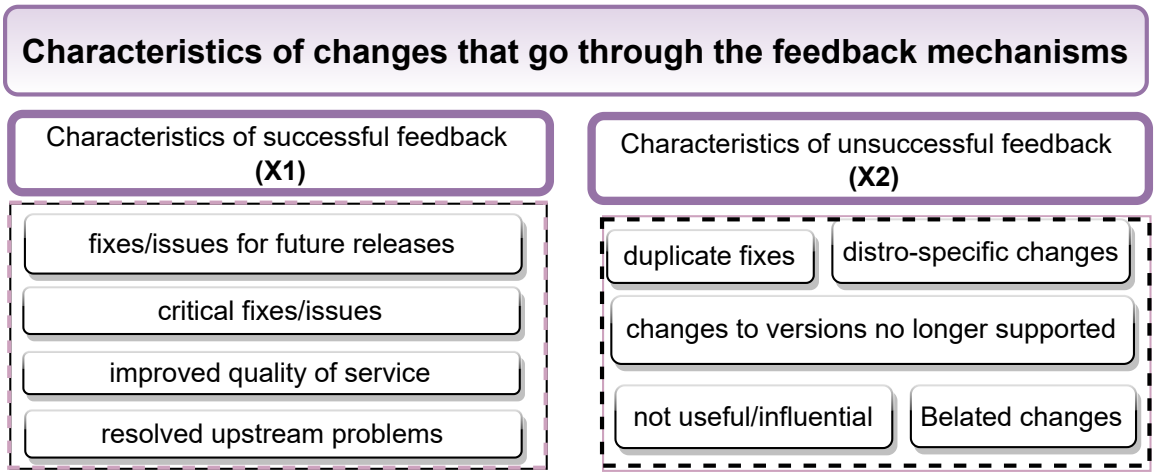
Sub-category T1 comprises the mechanisms used to provide feedback between the upstream and downstream communities of a SECO. As shown in Fig. 6.4, these mechanisms can either be online platforms or social events. The former correspond to issue trackers for code-base related discussions (launchpad, Storyboard, Gerrit, etc.), developer mailing lists for discussions on events and the codebase, and online collaboration tools such as user surveys and Q&A sites (e.g., *Ask.openstack.org*, *Serverfault*, etc.) to discuss technical questions and problems on a variety of topics. Social events refer to the OpenStack Summit, PTG, Forums or Meet-ups), which can either be in-person or virtual (such as in 2020 and 2021, due to Covid-19). Distros have different preferences for these feedback mechanisms. For example, Dn1's distro values PTGs, mailing list, and Forum, in that order, while Dn3's distro would only prefer PTGs and meet-ups, and for others (Dn17 and Dn21) social events are of equal importance.

Sub-category T2 consists of the diverse range of feedback requests and responses provided by distros and their stakeholders. As shown in Fig. 6.4, feedback requests can refer

to problems, bugs, crashes, complaints or suggestions for new features. Ideally, these requests should be resolved through a feedback response, which could consist of bug fixes, the implementation of a new feature or just forwarding a request to the right party for further processing. Surveyed distros reported that Bug fixes, New features, Emails with complaints/requesting help, and Bug reports are the most common feedback responses and requests sent to upstream OpenStack projects.

Characteristics of changes

Figure 6.5 Characteristics of feedback.



Not all feedback is resolved successfully; hence, sub-categories X1 and X2, respectively, focus on the characteristics of successful and unsuccessful feedback (see Fig. 6.5). Perhaps unsurprisingly, successful feedback (X1) involves timely, important, high-quality and relevant issue reports and fixes. Although, based on the surveyed distros, unsuccessful feedback is vetted before being sent out, such feedback typically meet one or more of the conditions of X2 that reduce chances of a successful feedback.

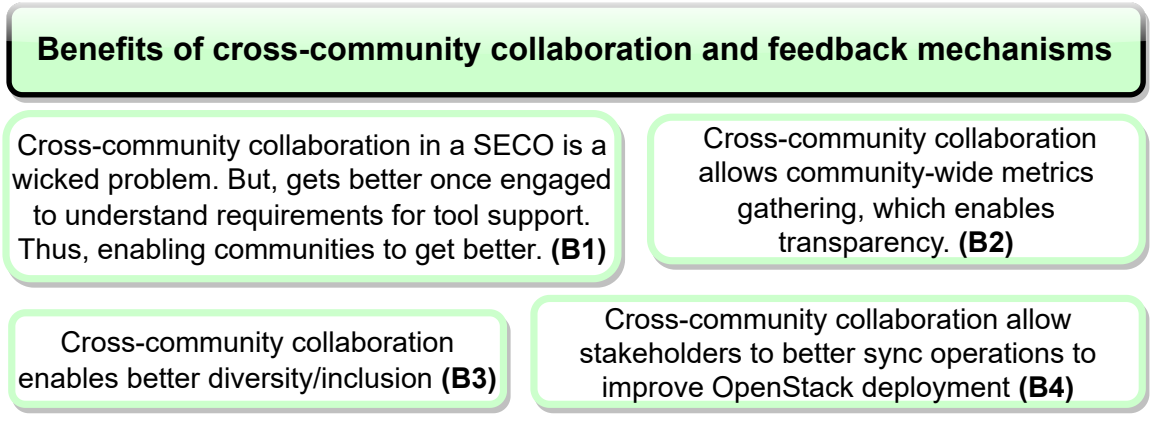
"Belated changes" and "duplicate fixes" are the most mentioned characteristics of rejected feedback in X2. *Up3* mentioned that *"if we find Faults today and report them tomorrow I'm sure someone will fix them but if you find faults today and report them in years time we might not be able to get anyone to listen."* (Up3).

Dealing with *duplicate fixes* also is taxing, since it corresponds to wasted effort. This occurs when a distro invests substantial resources to address an issue, only to realize that another community had resolved the issue earlier. "*We have been working hard for that fix for the past two months, and yesterday I was shocked after posting the fix to Nova when Mirantis reported that they resolved the bug over a month; this is frustrating.*" (Dn3). "*No one has yet offered a solution to avoid duplicate fixes. It's still an open problem that I wish perhaps the scientific [community] may one day suggest a breakthrough to this challenging problem.*" (Up4).

The other reasons for rejection are related to the general usefulness of feedback outside the distro providing it, as well as to the timeliness of suggested feedback.

Benefits of cross-community collaboration

Figure 6.6 Benefits associated to cross-community collaboration



Four out of 15 constructs in the emerged theory in Fig. 6.3(B) address the benefits

(B1, B2, B3, and B4 shown in Fig. 6.6) of cross-community collaboration and feedback mechanisms. Some benefits such as B3 (*better diversity/inclusion*) are observed to manifest sooner than later. One of the focus group participants noticed: "*The community is so large, but everyone in it seems to value and respect one another. This creates a safe place for anyone, client or vendor, to grow and thrive*"(Dn16), enabling a diverse and inclusive atmosphere to quickly appear (and potentially remain for a long period of time).

On the other hand, it takes a while for B1 and B2 to have visible effects and observable benefits. Even though cross-community collaboration (B1) becomes visible only after multiple attempts, the effects are long-lasting, and pay off dividends. This is emphasized and highlighted by the participants in this study and the mailing list discussions : "*Several successful calls for operations (Ops) and development (Dev) to collaborate more closely to speed up upgrade time to newer releases and deployment. Distros/end-users usually spend several hours on technical know-how and staffing, making them vulnerable to solo efforts.*"(Up4).

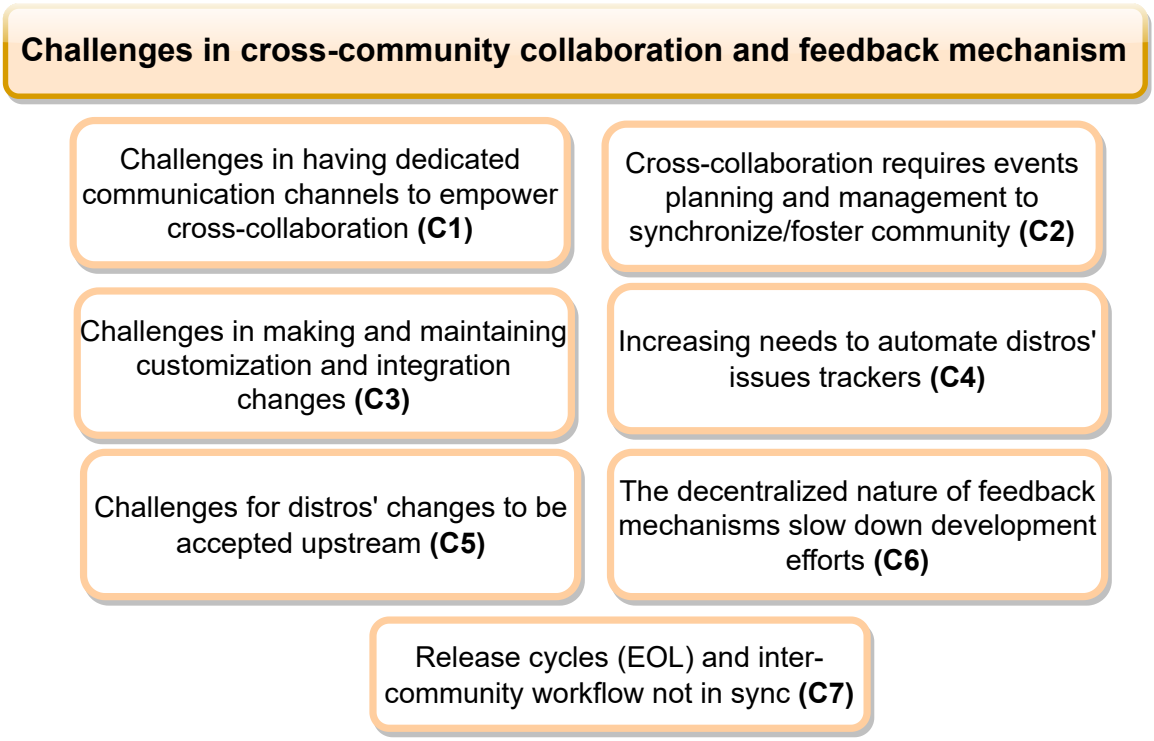
Similarly, B2 ensures that metrics at all levels of contribution and collaboration are measured, enabling a transparent SECO for all stakeholders. "*When collecting statistics, we think of transparency and deliver these metrics in an API format so that projects can use them on-demand. OpenStack is massive and complex, and it's tough to understand what's happening everywhere. Measure and reporting on key areas of community contribution are vital for the community.*"(Dn13).

Meanwhile, the ability to better synchronize operations (B4) has mid-term effects benefiting end-users and operators, and easing deployment operations. One analyzed distro suggested that cross-community collaboration has increased their chance of getting a lasting solution as they can learn and benefit from others' rich experiences (B4):"*Deploying*

OpenStack with large clusters of more than 2000 servers in our case was a significant issue until we discussed this problem and folks from other communities and some distros brought in different solutions sets, and we now have a stable solution to the problem."(Dn15).

Challenges in cross-community collaboration

Figure 6.7 Challenges associated to cross-community collaboration



Cross-community collaboration also brings 7 challenges to the communities involved. These challenges, shown in Fig. 6.7 are friction points that can slow down development time if not properly managed. Five of the seven challenges relate to purely technical activities, i.e., C3, C4, C5, C6, C7, relating to T2 (Feedback response, Fig. 6.4) and both X1 and X2 in Fig. 6.5. On the other hand, C1 and C2 refer to social challenges or requirements, and as such relate to T1 in Fig. 6.4 (i.e., *Online platform* and *Social events*, respectively).

Concerning C1, surveyed participants affirm that communication through dedicated channels is challenging, making it difficult to keep track of cross-community discussions: "*Upstream needs a dedicated system for feedback. With uncontrolled traffic, channels quickly get noisy to focus on a follow-up discussion.*"(Dn7) Also, another participant shared concerns about the decentralized nature of the OpenStack codebase (C6), which comes with additional communication overhead: "*Project segmentation with multiple channels to support each sub-project or team...*"(Dn15). Since OpenStack community members are distributed across the globe with different time zones, coordinating events to accommodate everyone is challenging (C2): "*For this brainstorming session, we had IBM team in Beijing and other guys on east coast of US. In order to cover two time zones 13-hr apart, we chose 9pm EST time for the convenience of us.*"(Dn13). C2 became even more prominent during the Covid-19 pandemic, as lockdown rules pushed all meetings onto online platforms running in different time zones.

The other challenges involve potential disconnects between distros and the upstream OpenStack project. For example, concerning C3 and C5, distros' changes usually involve customizations made for their own needs: "*we usually add our specifics to changes that are not needed upstream or at least for everyone to use.*"(Dn6). While this is a good thing for distros (customization gives distros a competitive advantage), it is challenging for upstream to accept such changes. Similarly, C7 relates to a disconnect between the release cycles of upstream and distros. Upstream follows a rapid six-month[18] release cycle [283], with releases going end-of-life (EOL) after 24 months. However, distros have longer release cycles and provide support to end-users for longer periods (between 48 to 60 months). Therefore, if distros make changes on a release that has gone EOL and push that change upstream, it will most likely be rejected (X2, changes to a version no longer supported):

[18] `https://releases.openstack.org`

"*we found a bug and reported it to OpenStack Mitaka [release] and they say 'it's in Ocata we are now in Queens come and fix that in Queens' but that function is not there anymore we are already pretty late so we kind of lose the essence of CI and we lose the value we can add to this ecosystem.* " (Dn9).

While participants were unanimous about the many mechanisms and forms of feedback between distros and upstream, as discussed earlier, the degree of horizontal collaboration amongst distros was said to be limited. 85.7% outright report that their distros do not collaborate with other distros. Part of the reason for this are the lack of automation between distros' issue trackers (C4), as well as the decentralized way in which inter-distro communication and feedback is handled (C6).

While the decentralized nature of feedback may seem good for "divide and conquere" purpose, this doesn't seem to be a perceived advantage for distros since they have have to wait for all collaborators to be in synchronization, which usually is not the case. *"OpenStack has several projects that we push feedback patches directly to, for example, Neutron and all its sub-projects repos, separately. We usually get delays when collaborators from other communities are not in synch ."*(Dn21). *The status quo of communication among distros and upstream on patches is icky, slowing down the iterative code review process, especially in reported unresponsiveness to code reviewers' comments.*(Up4) It is challenging to keep track of all communications going on around code reviews for patches across different sub-projects.

6.4 Quantitative study approach

This section aims at validating the grounded theory derived in the previous section, in terms of the following two research questions:

- **RQ2:** ***How diverse is the feedback contributed by distros****?*
- **RQ3:** ***What roles do distros play in synchronizing SECO feedback****?*

6.4.1 Approach for RQ2

For RQ2, we first analyze the surveyed distros' feedback activities to understand the prevalence of New features, Filed bugs, Resolved bugs and Emails from April 2012 to October 2021 (spanning 20 OpenStack releases). For this, we mined the following repositories:

1. **Upstream codebase repositories**: *The open-source Stackalytics infrastructure*[19] hosted by OpenStack[20], collects and processes commits, lines of code changed, code reviews, new features, filed and resolved bugs from the OpenStack codebase and project repositories. It also provides the official mapping from each contributor to the project(s) they contribute to, and provides their affiliation (distro), and all the changes they make. Thus, for filed/resolved bugs and newly accepted features, we used Stackalytics to mine the OpenStack issue trackers (Launchpad and Storyboard) and 1,283 upstream codebase repositories, comprising both the 63 core OpenStack projects and all major sub-projects.

2. **Distros issues tracker/repositories**: Next, we obtained permission from the 21 studied distros to access their private issue trackers and repositories. For each distro, we wanted to map the Stackalytics results in terms of feedback received upstream (such as filed bugs, resolved bugs, and new features) to the distros responsible for it. For this, we used issue IDs and email addresses, and usersID for contributors,as shown in Fig. 3.2. Therefore, we ended up with a mapping of each distro and the list of all

[19] https://wiki.openstack.org/wiki/Stackalytics
[20] https://opendev.org/x/stackalytics

feedback types (filed bugs, resolved bugs, new features and Emails)

3. **The developers' mailing list**[21]: We extracted 140,361 archived emails that contain the various kinds of feedback sent between distros and upstream projects (see Fig. 6.4), such as emails requesting for help, bug/crash reports, comments/discussions on new features, etc. Emails on such developer mailing lists are moderated, and only authorized authors can send messages. Emails have [tags] between brackets in their subject with the involved (sub-)project/module name, and special keywords to indicate targeted projects/distros and topics of discussion. We follow a similar approach as in our earlier work [272] to filter out emails by feedback type. For example, we use regular expressions (regex) such as:
   `egrep -iRno -C 10` $'patch|patchs|patchset|issue|issues|bug|bugs|[N|n]ewfeatures|feature|[B|b]lueprint|[B|b]lue prin*'$ *.txt,
   etc., to extract all exchanges on bug patches and new features. After the filtering process, 140,361 emails were classified under the 21 distros from an initial extracted list of 166,101 emails . The 25,740 emails not related to distros' feedback or generated by bots in the CI/CD pipeline were ignored.

Analysis on extracted — RQ2:

We then aggregated our analysis at project level. For example, Nova (a core project) [283] has about 20 sub-projects[22] (e.g., $nova-spec$ and $nova-powervm$), feedback can be submitted to. Therefore, we aggregated all changes that belong to these sub-projects under their core project. Thus, we mined activities for New features (44 projects), Filed bugs (44 projects), Resolved Bugs (44 projects), and Emails (49 projects) that each distro authored from the Essex (2012) up to the Xena (2021) release, a period of about 10 years and 20

[21] https://tinyurl.com/2p9ezk7u
[22] https://tinyurl.com/2p9ejr9j

OpenStack release activities. As of October 30th, 2021, the OpenStack upstream codebase comprises #LOC: 60,301,408 and #Commits: 423,973. Concerning Bug and Crash reports, we did not have data for these two categories to analyze, hence we consider their analysis as future work.

Next, we measure how feedback is dispersed for all four feedback types to understand the pattern of feedback distribution in the studied period. We also calculate if a given feedback type is dominating in a distro or not. For a given distro, we first determine the relative proportion of feedback $|F_n|$ that it provides for each feedback type, such that $F_1 > F_2 > F_3 > F_4$ (and $\sum_{i=1}^{4} F_i = 1$). We then calculate the relative gap between the top two feedback types provided by the distro, i.e., F_1 and F_2, as follows: $100 * (F_1 - F_2)/F_2$. If this relative gap is greater than or equal to 40%, we argue that the feedback type of F_1 is dominating amongst the feedback provided by the distro. If not, we argue that no feedback type dominates.

Finally, we also calculate for each feedback type the normalized entropy across all distros [293, 294]. An entropy value of 1 means that all distros contribute equally to a specific feedback type, while an entropy value of 0 means that one distro dominates all produced feedback of a given feedback type. We also calculate the cumulative distribution function to visualize how feedback is spread across/dominated by distros. For Bugs Filed and resolved, we also measure the median time it takes for distros to resolve a bug.

6.4.2 Approach for RQ3

Secondly, we aim at understanding how (well) feedback is synchronized across communities. For this, we use social network analysis [295] to measure the coverage of feedback broadcast by distros to projects' repositories. For example, if a particular distro or project

has a fix or a solution to a problem, ideally it would like to propagate this across the entire network such that other distros/projects become aware, trying to reduce wasted effort and resources for distros. We perform this analysis for the network structures of the four types of feedback studied in this chapter.

For each network structure, the nodes represent projects or distros. We use an edge list to construct social network communities [296] that represent the relationships between distros and projects in a weighted and undirected graph, using the data mined in the previous subsection. As edge weights, we use the number of feedback contributions of a particular category (e.g., fixes) made by a particular distro and sent to a given project. Full details of the network structure and analysis are available online in our replication package [288].

To measure how feedback is communicated and synchronized across the four social network communities, we use several algorithms (such as independent cascading, influence maximization, and nine centrality algorithms explained below). First, we adopt an independent cascading [297] model, where a node (distro or project) becomes aware of feedback based on the connections to each of its neighbors independently, and on some probability proportional to the weight of the corresponding edge. So, if a node is aware of, say, a fix, that node is assigned a probability to spread that information about the fix to its neighbors in the community.

Moreover, given the many possible paths between potential network influencers (the most central nodes/distros with large impact on the network structure and its ability to spread feedback across various communities) and influential nodes (nodes with more connections among connected distros and projects) [295, 298, 299], we also use a seeding approach [300], which is an influence maximization strategy that searches the best nodes to "intentionally activate" in other to spread information (feedback) across the network

community. Since finding an optimal seeding is an NP-hard problem [298, 301], we experimented with nine centrality algorithms commonly used in the literature [302–306], implemented in the NetworkX library [296] in Python.

The centrality algorithms enabled us to obtain 4,268 different seeding options to identify top influencers (distros/projects) in the cross-community social networks. The latter enable to spread feedback (fixes, Bug filed, new features, and emails) further into the network community, to inform other distros that initially would not have been aware of the feedback. Among the nine centrality algorithms, the following three outperformed the others in several communities in terms of centrality measures: the Load centrality in the Filed bugs community, Betweenness centrality in the Fixes community, and Katz centrality in both the Email and New feature communities.

However, since our network communities involve complex network structures, we also decided to use a strategy that could identify hidden communities within our social networks, to simplify them before running the centrality algorithms. We did community detection [307] using the common Girvan-Newman [308] and Louvain [309] algorithms. In our study, the Louvain algorithm performed better than Girvan-Newman. It detected one community in the Filed bugs and Fixes networks, two communities in the New feature network, and seven in the Email networks. Next, we then apply the three successful centrality algorithms discussed earlier in each of the detected communities.

Finally, to understand the vulnerability of a detected community in case a key distro should abandon the SECO, we adopt the truck factor metric. In our context,*it measures the minimum number of distros to quit before the SECO is "hit by a truck"* [310, 311]. For all four types of feedback, we sorted the dataset in descending order of feedback contribution, then measured how many distros contributed more than 50% of a given feedback type.

6.5 Quantitative results

6.5.1 RQ2: *How diverse is the feedback contributed by distros?*

Distros are essential and productive in cross-community feedback, contributing 25.6% of all new features in the OpenStack ecosystem, authoring 30.7% emails relating to technical topics, reporting and resolving 44.3% and 30.7% bugs respectively.

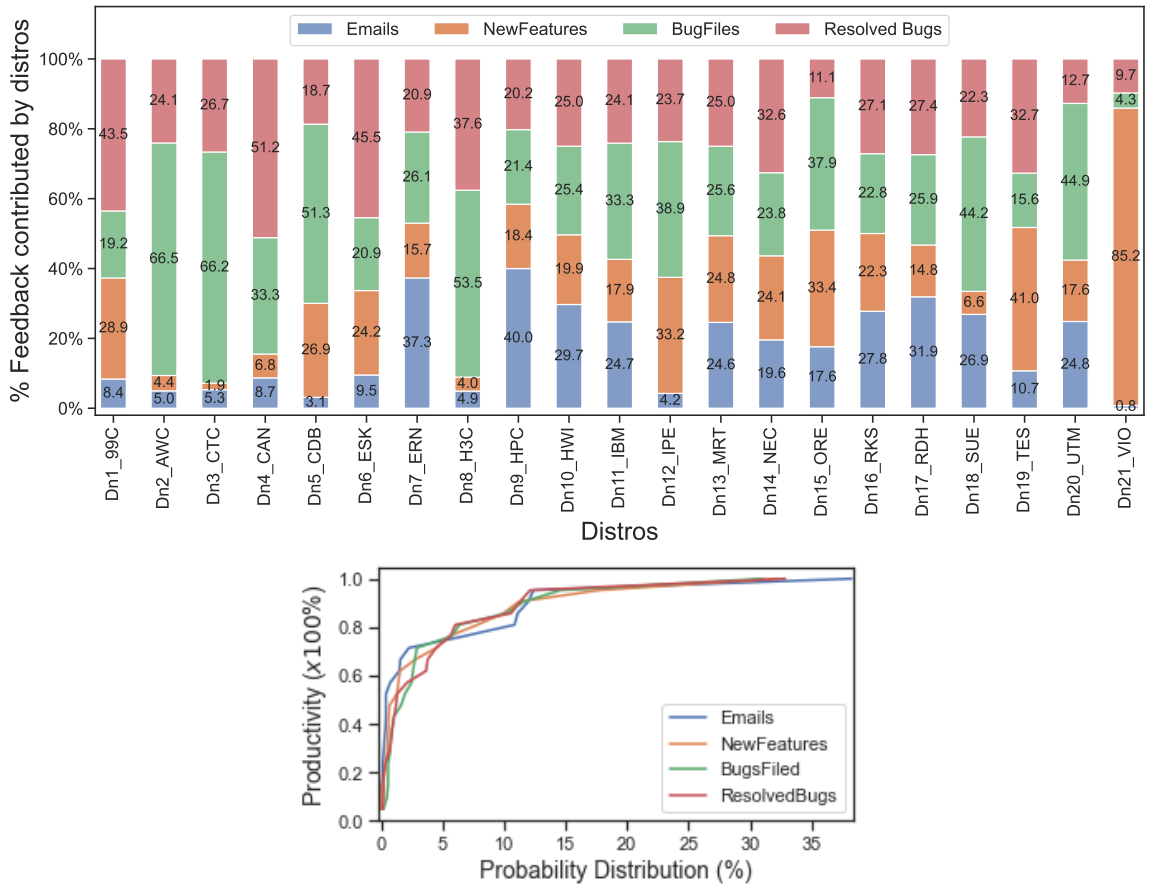
According to the observed feedback types (see **T2**), distros turn out to be productive in terms of all feedback types. Yet, one issue identified in our qualitative studies involves a lack of traceability between different aspects of a given feedback contribution. For example, the Mailing-list feedback channel is noisy with lots of distraction that can prevent other distros from becoming aware of what is happening. "*Because conversations can be spread around multiple places, it can be helpful to link all of these conversations with little breadcrumbs. A mailing list thread might reference a [Gerrit] review, which might reference a log of an IRC conversation, which might reference a blog post, which might reference a bug, which might reference a previous commit message which referenced a previous mailing list thread.*"(Dn17).

The feedback contributions from distros follow a Pareto distribution (power-law) [51] with 80% of all downstream changes coming from 20% of the distros.

This distribution is similar for all categories of feedback that distros make, as shown in Figure 6.8 [R]. Upstream has a policy to regulate upstream companies' contributions in each release/development cycle: "*One of the things that we really believe in and encourage upstream is the diversity of contributions. So, most contributions to projects hosted by the OpenStack foundation shouldn't just come from one company.*"(Up5)

Our Pareto result does not contradict this policy because no single distro dominates. However, about five distros contribute 80% of feedback on average. Furthermore, the feedback

Figure 6.8 Feedback productivity contributed by distros, [L] %distribution of feedback, [R] Pareto CDF per feedback type.

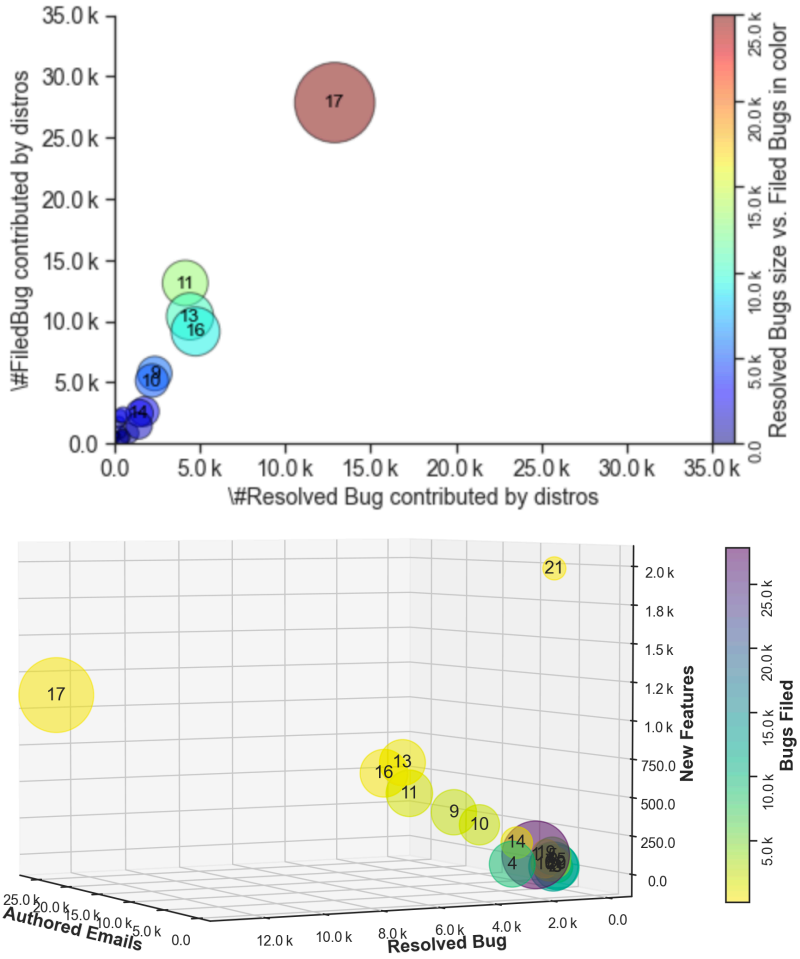


entropy for all four categories is close to 0.22.

In particular, the entropy for Filed Bugs is 0.227, while the other three entropy values (Emails, New Features, Bug Resolved) = 0.222. This further confirms that contributions to all feedback types are not diverse (some distros are dominating) in terms of participating distros.

While nine distros provide the four types of feedback in (more or less) equal proportions, 12 distros are dominated by one type of feedback. Fig. 6.8[L] shows the proportion of feedback types contributed by distros. If a distro has at least 40% for a given feedback type, we say its feedback is "dominated" by that feedback type. In particular,

Figure 6.9 At the top #Bug filed vs Resolved [L] at the bottom, shows a 4-D plot [R] of feedback contributed by distros. Circle size represents relative feedback quantity, and numbers represent distros (without pre-/suffix for visualization purposes).



three distros (Dn1_99C, Dn4_CAN, and Dn6_ESK) dominate in Resolved Bug feedback, six (Dn2_AWC, Dn3_CTC, Dn5_CDB, Dn8_H3C, Dn18_SUE, Dn20_UTM) in Filed Bugs, one (Dn21_VIO) in New features, and two (Dn7_ERN, Dn9_HPC) in Emails. Nine distros had values below the 40% threshold, therefore, these nine distros (see Fig. 6.8[L], from Dn10_HWI to Dn17_RDH, and Dn19_TES) have two or more of their feedback types almost evenly distributed.

Fig. 6.9[Bottom] summarizes the feedback productivity of the distros in terms of all 4

types of feedback, i.e., Emails on the $x-axis$, Resolved bugs on the $y-axis$, New features on the $z-axis$, and Filed bugs represented by color intensity. Fig. 6.9[Top] projects this plot onto the dimensions of Filed bugs and Resolved bugs. These plots not only show that for a given feedback type only a small subset of distros dominates the traffic in the feedback category, but also that these dominant distros differ across feedback types. For example, fewer than five of the 21 distros in the Filed Bugs community are not colored yellow in the 4th dimension in Fig. 6.8[R], while in the New Feature category 85.2% of the traffic comes from distro Dn21_VIO. Similar observations hold for the other feedback types.

A median of 25% bug reports are duplicate reports.

Fig. 6.9[L] shows that distros file more bugs than they resolve, since all distros are above the lead diagonal line. They have filed a total of 89,713 bug reports, with a median of 1,755 reports, 75% of which are **unique** (reports that are not a duplicate of an earlier reported bug). This implies that a median of 25% of reports were redundant (duplicate reports), affecting a median of three distros. Among the total number of uniquely reported bugs, 43.8% (a median of 518) were resolved as **useful feedback** by distros, while 56.2% (median of 862) are **missed opportunities** (reported without any response).

Wasted effort is a pain point for distros, since they spend an extra fix time of 12 days to resolve wasted effort.

We observed how 54.2% (39,286) of the bug fixes were propagated upstream, while 29% remained locally since they corresponded to local customizations, and 16.8% were **wasted effort** (resolved redundant bugs), i.e., bugs that were already fixed by others and fixed again by the second distro after a period of time, without noticing the earlier fix. Distros spend a median of 13 days to fix unique bugs (useful feedback) compared to an extra median of 12 days (i.e., median of 25 days in total) to fix redundant bugs (wasted efforts) or to discover

whether an existing fix exists.

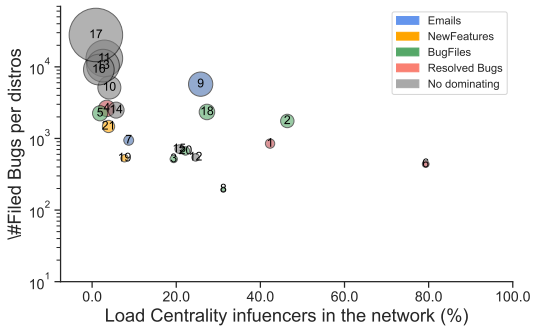
6.5.2 RQ3: *What roles do distros play in synchronizing SECO feedback?*

Surprisingly, influential distros are not the most dominant in their respective feedback type communities; they play general-purpose roles in synchronizing feedback.

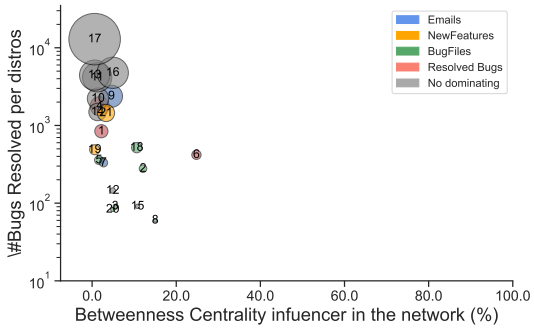
Each of the four plots in Figure 6.10 (a), (b), (c), and (d) shows the role of distros in the network community for one feedback type. These plots were obtained by the influence maximization algorithms that identified the most influential distros as shown in Table 6.2. Then, based on our community detection algorithm, two communities were detected in the New features community, one community detected in both the Bugs filed and resolved communities, and finally, seven communities were detected in the Email community. An $x-axis$ position more to the right in Figure 6.10 indicates lower influence, while the nodes are coloured according to their dominating feedback type (or grey if no feedback type dominates). The size of each distro in a particular feedback category in Figure 6.10 (a), (b), (c), and (d) indicates the volume of that particular distro's feedback. For example, in Figure 6.10 (d), (#21) has the largest volume of new feature feedback.

In Fig. 6.10(a), we notice that dominating distros are more spread out horizontally, while the distros without dominating feedback type (grey) are clustered right below the most productive (and at the same time most influential) distro (#17). This observation indicates that the filed bug community is the most demanding community to synchronize. Distros that are closer to the most influential distro quickly become influenced; hence centrality algorithms use less computational power to reach them, unlike distros/nodes that are distant from influential distros (see Table 6.2 for influential distros in each community).

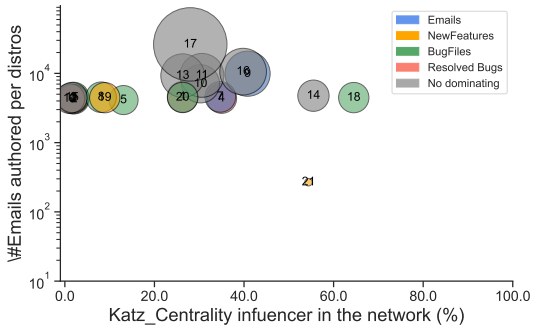
Figure 6.10 Centrality algorithms' detected communities for each feedback type.



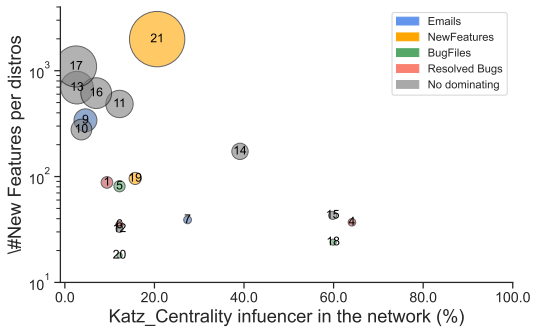
(a) Filed Bugs vs. Load Centrality



(b) Resolved bugs vs. Betweenness centrality



(c) Email vs. Katz Centrality



(d) New Features vs. Katz Centrality

In contrast, Fig. 6.10(b) (the resolved bug community) requires fewer effort to synchronize feedback . It shows clusters of dominated distros rallying under the influential distros (#17), with little spread on dominating distros.

The email community in Fig. 6.10(c) spreads more along the horizontal axis but stays almost constant on the vertical. This is the only community where we have two influential distros (close to the $y-axis$), i.e., the dominating distros (#4) and (#9).

Finally, Fig. 6.10(d) has the widest spread in terms of feedback and it is the only community where a dominating distro produces the largest amount of feedback (#21). Based on Table 6.2 we notice that the new feature community has two detected communities, which, according to Katz centrality, simplifies feedback synchronization within a social network.

Community detection algorithms and influence maximization are pivotal to identify feedback synchronization, since such algorithms detect simplified versions of feedback communities and influential distros. As such, we observe patterns of relationship between influential distros and dominant feedback types.

Influential distros are associated with a higher TF ($\geq$3) in terms of produced feedback than is common for regular open-source projects.

Besides the Email community with a TF of six, all other communities have a TF of three as shown in Table 6.3. When compared to the TF of regular open-source projects [310].

Furthermore, we observed in Table 6.2 and Table 6.3 that

influential distros turn out to have a more consistent and regular flow (see Fig. 6.8) in contributing feedback to upstream projects and are more central to reach all other projects and distros based on the centrality algorithms.

SECOs can run a significant risk of sustainability degradation if influential distros become dominating in their feedback types and their influence become even strong, i.e., TF $<$ 2,

(see Table 6.3) this condition will imply a variety of implications. It maybe an indication that the SECO is homogeneous, therefore, not diverse, etc., or other distros are abandoning at faster rate. However, based on our qualitative study, we did not find any such case, toxicity, etc. Moreover, OpenStack has upstream policy that regulate monopoly from a single company.

Table 6.2: Influential distros in detected communities (DC).

Community	Most influential distros/nodes	#DC
New features (NF)	Dn17_RDH, Dn13_MRT	2
Filed bugs (FB)	Dn17_RDH	1
Resolved bugs (RB)	Dn17_RDH	1
Emails (EM)†	Dn1, Dn4, Dn9, Dn11, Dn13, Dn16, Dn17	7

Suffix of influential Emails† distros removed for space purposes.

Table 6.3: Truck factor for each detected community (DC): NF, FB, RB, EM.

DC	TF	Value $\geq$	50-PCT	TF Distros†
NF	3	3.7K	3.1K	{Dn13, Dn17, Dn21}
FB	3	51.4K	44.8K	{Dn11, Dn13, Dn17}
RB	3	22.1K	44.8K	{Dn11, Dn13, Dn17}
EM	6	72.4K	67.4K	{Dn9, Dn10, Dn11, Dn13, Dn16, Dn17}

Suffix of all influential distros removed for space purposes.

6.6 Threats to validity

Our study might have been affected by various threats that we discuss below.

Construct validity: Relying on the OpenStack events and allotted time to conduct the focus groups study could jeopardize our design constructs. However, OpenStack events turn out to be the only venue where we could reach out simultaneously to all the distros and interact with them on shared objectives.

Using the Stackalytics community tool to obtain our quantitative data about the upstream community could introduce bias. However, Stackalytics is not locked to a particular distro. It uses common git commands/functionalities similar to custom scripting, but with the unique exception that while querying confidential repos, it automatically reads confidential data containing permission tokens mapping each change to its authors and their affiliations/distro. This is essential for our study.

To extract themes in the mailing list archived, we use regular expressions, which could be noisy and subject to false positives. To mitigate this threat, we strictly used the queries of [272] on the same dataset.

Internal validity: These threats relate to alternative explanations to our findings. Our study may suffer from subjectivity in qualitative analysis. To mitigate this risk, we did multiple (10) rounds of iterations in our GT following rigorous methodologies for open/axial/selective coding. All authors were deeply involved in several rounds of negotiated agreement and deliberations to reach consensus. We also searched the literature in each given step to guide our decisions.

Moreover, the author of this thesis is a foundation member of OpenStack, which could bring potential bias to this study. However, to mitigate this threat, his involvement with OpenStack was limited to facilitating access to resources that would not have been possible

otherwise, i.e., distros are vendor locks. The other authors of this chapter actively participated in this study, and each step of the GT analysis was intensively deliberated. Therefore, any bias from the author of this thesis is canceled out.

External validity: OpenStack is a complex open-source SECO, however our results may not generalize to other SECOs with distros. To mitigate this threat, our survey, focused groups, GT and quantitative studies were designed to be as much ecosystem-agnostic as possible, and hence should be applicable to other SECOs. Hence, SECOs, practitioners and researchers could benefit from our study design to analyze cross-community collaboration. Also, the Covid-19 pandemic has introduced a culture of online collaboration that could disrupt cross-communities collaboration, or potentially change existing practices. More research is needed to understand the impact of this threat.

Reliability/Conclusive validity: We make our datasets and tools available online [288] for replication and transparency, with exception to confidential participant information.

6.7 Conclusion

The sustainability of SECOs remains a substantial challenge despite their proliferation. To mitigate this challenge, this chapter studies the mechanisms used by the OpenStack SECO to collect feedback from 21 distros and end-users, then propagate such feedback to improve upstream releases, tools and policies.

First, we use the Straussian grounded theory methodology with an open-ended unstructured interview, survey, focus groups, and analysis of 120 mailing list threads. The resulting theory features 15 constructs divided across four categories (2 feedback mechanisms, 2 characteristics of feedback, 7 challenges and 4 benefits of cross-community collaboration) to explain how SECOs manage to sustain cross-community collaboration. Then, we

validate our theory by empirically analyzing changes that distros contribute upstream as feedback by quantitatively mining 140,261 mailing list emails, 142,914 filed bugs, 65,179 fixes, and 4,349 new features. Finally, we use influence maximization from social network analysis to understand how feedback is synchronized/broadcast throughout the SECO.

Amongst others, our results suggest that distros are important producers of feedback within a SECO. They contribute 25.6% new features, 30.7% emails on technical topics, 44.3% and 30.7% bug reports and resolutions, respectively. Distro feedback follows a Pareto distribution with 80% of all distros' feedback contributed by 20% of the distros. We detected four feedback communities in the SECO using social network analysis. Using centrality algorithms, we found seven influential distros in the email feedback community, two in the new features community, one in the filed bugs community, and one in the resolved bugs communities. Influential distros have higher truck factors ($\geq$3), which are higher than common open-source truck factors. Surprisingly, influential distros are not the most dominant in their respective feedback type communities; they play general-purpose roles in synchronizing feedback.

Chapter 7

Discussion and conclusion

7.1 Discussion

SECOs are playing an essential role in modern software engineering, from infrastructure ecosystems like the Linux kernel and OpenStack, to programming language-related ecosystems such as Maven and npm. The main reasons why such SECOs build a collective infrastructure or product based on collaborations between a wide range of socio-technical stakeholders, are not only to share the burden of the work (letting everyone focus on their own expertise), but to keep innovating, polishing and evolving the SECO in light of the rapidly changing requirements and environments of stakeholders. However, such sustainability should not be taken for granted, since quite some obstacles could lead a nascent SECO to its demise: lack of contributors, lack of productive collaboration, missed internal deadlines, lack of feedback (or inability for the feedback to reach its destination).

This thesis focused on those four obstacles along the closed-loop feedback lifecycle of a typical SECO, aiming to empirically validate the statement:
"The sustainability of a SECO relies on a number of essential socio-technical activities along its lifecycle."

In particular, we empirically analyzed the socio-technical activities in place in a large, successful SECO (OpenStack), as well as the extent to which such activities relate to increased sustainability, using a mixed-methods methodology. The construct of sustainability is measured through a variety of metrics, depending on the study and context.

The four steps socio-technical activities proposed in this thesis found to correlate with sustainability could be implemented differently across SECOs, hence requiring changes in methodology, but consistent findings can be made. For example, we conducted an onboarding study at Microsoft[1] using a different methodology; such as interviews and surveys. However, the results were consistent with the analysis in Chapter 3.

This thesis has several implications for different stakeholders. First, the research community now has a theory that explains the importance of socio-technical activities of feedback synchronization, communication, and diversity in growing a sustainable SECO. This theory is essential, not only for existing SECOs, but also for the many domains where new SECOs are being planned or built up, such as the emerging SECOs of AI-enabled systems[2], IoT systems and wearables. All of these are characterized by a rapid flow of new stakeholder requirements and expectations, as well as fast turn-over of relevant technology. Since developing a SECO in such circumstances is challenging and risky, our findings on the provenance and flow of feedback in a SECO can help manage expectations of SECO owners.

Second, our empirical findings allow SECOs to better understand their communities, the importance of onboarding (as well as challenges), influential contributors, etc. SECOs can also learn the different strategies to synchronize release schedules and use our persona to recruit potential candidates to their release team.

[1] Results of this onboarding study not included in this thesis for privacy concerns.

[2] https://www.mckinsey.com/business-functions/quantumblack/our-insights/global-survey-the-state-of-ai-in-2020

Third, our empirical findings regarding micro-collaboration have exposed important gaps in the traceability of joint developer contributions, where a substantial proportion of micro-collaboration cases is hidden within the review environment. Our methodology for identifying such cases could help practitioners re-document such micro-collaboration, for example using today's best practices (e.g., commit trailers). Furthermore, we also found indications of how micro-collaboration has a strong relation with onboarding.

Finally, the methodologies used by us in the four empirical studies, including the choice of SECO sustainability metrics, represent perhaps the largest contribution to practitioners. They can use it to measure their own SECO's sustainability, across a wide range of interpretations. Furthermore, organizations, businesses and investors could rely on our findings to invest in SECOs that are sustainable to protect their return of investment.

7.2 Conclusion

This thesis provides empirical findings on SECOs' sustainability by focusing on four major SECO lifecycle phases, i.e., onboarding, micro-collaboration, release synchronization, and cross-community feedback mechanisms. It makes the following contributions:

— We conducted a participant observation study at the SECO level to understand the phenomena of onboarding and mining OpenStack software repositories to find evidence of new contributors' technical activities such as bugs, code changes, etc. We found a significant correlation of onboarding with increased gender diversity and patch acceptance rate, while the time until the first commit and bug-proneness of contributions decreases. In other words, good socio-technical onboarding practices seem to relate with SECO sustainability.

— We performed a mixed-methods analysis of collaboration practices in the OpenStack

ecosystem through semi-structured interviews and a mining repositories analysis. Our empirical findings reveal that micro-collaboration code practices relate to an increase in

productivity, diversity, and quality, amongst others. Also, micro-collaboration encourages teamwork, enables accountability and recognition, and facilitates complex problem-solving skills. These findings again suggest that good micro-collaboration practices relate with higher SECO sustainability.

— We qualitatively analyzed OpenStack's release team's IRC meeting logs, which allowed us to identify, catalogue, and document ten major release synchronization activities, which we further validated through interviews. We built a persona to advise SECOs on skills required to be a successful release team member. Then, we also mined 22 OpenStack release repositories with over 2,961 projects to find evidence of multiple release strategies co-existing each release cycle. Hence, judicious use of release synchronization activities relates to higher SECO sustainability.

— Finally, we conducted a mixed-method study to understand the effectiveness of feedback mechanisms through a grounded theory study of unstructured interviews, a survey, focus groups, and mailing list archives, leading to a theory involving 15 constructs. We validate our theory by mining the upstream/downstream code base. Our empirical investigations suggest that distros contribute substantially to a SECO's feedback and that influential distros in a SECO do not focus on one specific feedback type, but cover a wide range of feedback types. This again suggests that a good choice of feedback practices can relate to higher SECO sustainability.

As such, our empirical studies were not able to reject the thesis statement, which (until further notice) still stands:

"The sustainability of a SECO relies on a number of essential socio-technical activities

along its lifecycle."

7.3 Future Work

This thesis opens many opportunities for future research on sustainable SECOs, some of which are already mentioned in the thesis. Below, we discuss higher-level directions of future work.

The research community should investigate onboarding at the organization level when multiple SECOs co-exist and interact. In particular, large organizations such as Microsoft, Meta, etc., usually acquire external companies to form a SECO; this acquisition typically creates a sudden change in the environment, requiring a massive onboarding effort for the employees of the acquired company. As newly acquired companies are moving in with their own tool sets, practices and culture, it is not clear how to set up an effective onboarding process in a short time for the nascent SECO to continue evolving.

Similarly, future work should also investigate other organization-level onboarding, including when contributors switch from one SECO to another within the same organization.

Future work should also aim at finding ways to automate and/or improve micro-collaboration, given that micro-collaboration has been shown to have several benefits, automating this activity could bring more transparency to the SECO and ensure proper attribution of rewards. Review environments like Gerrit or GitHub pull requests should be enhanced with better tool support to make micro-collaboration a first-class citizen, since GitHub's current best practices involving commit trailers in commit messages is still quite fragile. Explicit traceability would improve several software engineering tasks, including vulnerability management and debugging.

Future work should also investigate different types of SECOs, distributions and SECO end products, in order to validate the generalizability of our catalogue of ten release synchronization activities. SECOs should continue building more powerful tools to track these

activities, and perhaps consider the use of predictive models to notify the release team and affected projects of delays and other issues ahead of time.

Less straightforward, but at least equally important would be for researchers and SECOs to investigate ways of improving the social activities "A4 Establishing a Network of Trusted Liaisons" and "A5 Establishing Communication Channels", respectively.

Finally, there are more feedback types than those we have studied. One possible feedback type that future work could explore is how end user requirements propagated via the SECO's feedback mechanisms drive upstream development. Furthermore, future work should aim at quantitatively validating the remainder of our qualitative findings.

Bibliography

[1] Konstantinos Manikas. Revisiting software ecosystems research: A longitudinal literature study. *Journal of Systems and Software*, 117:84–103, 2016.

[2] Slinger Jansen and Michael A Cusumano. Defining software ecosystems: a survey of software platforms and business network governance. *Software ecosystems: analyzing and managing business networks in the software industry*, 13, 2013.

[3] Anil Gore and Sharayu Paranjpe. *Introduction*, pages 1–7. Springer Netherlands, Dordrecht, 2001.

[4] Christopher Bogart, Christian Kästner, and James Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 86–89. IEEE, 2015.

[5] Geir K. Hanssen. A longitudinal case study of an emerging software ecosystem: Implications for practice and theory. *Journal of Systems and Software*, 85(7):1455–1466, 2012. Software Ecosystems.

[6] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416, 2019.

[7] T. Mens, M. Claes, and P. Grosjean. Ecos: Ecological studies of open source software ecosystems. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 403–406, Feb 2014.

[8] Eleni Constantinou and Tom Mens. An empirical comparison of developer retention in the rubygems and npm software ecosystems. *Innovations in Systems and Software Engineering*, 13(2-3):101–115, 2017.

[9] Alexander Serebrenik and Tom Mens. Challenges in software ecosystems research. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, page 40. ACM, 2015.

[10] M. Hilton and A. Begel. A study of the organizational dynamics of software teams. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 191–200, May 2017.

[11] Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems – a systematic literature review. *Journal of Systems and Software*, 86(5):1294–1306, 2013.

[12] Keith Mallinson. Smartphone revolution: Technology patenting and licensing fosters innovation, market entry, and exceptional growth. *IEEE Consumer Electronics Magazine*, 4(2):60–66, 2015.

[13] M. Gómez, B. Adams, W. Maalej, M. Monperrus, and R. Rouvoy. App store 2.0: From crowdsourced information to actionable feedback in mobile ecosystems. *IEEE Software*, 34(2):81–89, Mar 2017.

[14] Caio Steglich, Sabrina Marczak, Cleidson R. B. de Souza, Luiz Pedro Guerra, Luiz Henrique Mosmann, Fernando Figueira Filho, and Marcelo Perin. Social aspects and how they influence mseco developers. In *Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '19, pages 99–106, Piscataway, NJ, USA, 2019. IEEE Press.

[15] Keith Mallinson. Smartphone revolution technology patenting and licensing fosters innovation, market entry, and exceptional growth. *Consumer Electronics Magazine, IEEE*, 4:60–66, 04 2015.

[16] S. Braun, R. Carbon, and M. Naab. Piloting a mobile-app ecosystem for smart farming. *IEEE Software*, 33(4):9–14, July 2016.

[17] S. L. Lim and P. J. Bentley. Investigating app store ranking algorithms using a simulation of mobile app ecosystems. In *2013 IEEE Congress on Evolutionary Computation*, pages 2672–2679, June 2013.

[18] Reuben Binns, Ulrik Lyngs, Max Van Kleek, Jun Zhao, Timothy Libert, and Nigel Shadbolt. Third party tracking in the mobile ecosystem. In *Proceedings of the 10th ACM Conference on Web Science*, pages 23–31. ACM, 2018.

[19] Aaron Beach, Mike Gartrell, Sirisha Akkala, Jack Elston, John Kelley, Keisuke Nishimoto, Baishakhi Ray, Sergei Razgulin, Karthik Sundaresan, Bonnie Surendar,

et al. Whozthat? evolving an ecosystem for context-aware mobile social networks. *IEEE network*, 22(4):50–55, 2008.

[20] Armstrong Foundjem. Release synchronization in software ecosystems. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 135–137. IEEE, 2019.

[21] WHO. who.int. https://www.who.int/about/who-we-are/constitution, 2019. Accessed: 2019-07-07.

[22] D. Posnett, V. Filkov, and P. Devanbu. Ecological inference in empirical software engineering. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 362–371, Nov 2011.

[23] E. Constantinou and T. Mens. Socio-technical evolution of the ruby ecosystem in github. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 34–44, Feb 2017.

[24] Tom Mens. An ecosystemic and socio-technical view on software maintenance and evolution. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–8, 2016.

[25] Tom Mens, Bram Adams, and Josianne Marsan. Towards an interdisciplinary, socio-technical analysis of software ecosystem health. *arXiv preprint arXiv:1711.04532*, 2017.

[26] B. Baudry, M. Monperrus, C. Mony, F. Chauvel, F. Fleurey, and S. Clarke. Diversify: Ecology-inspired software evolution for diversity emergence. In *2014 Software*

Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), pages 395–398, Feb 2014.

[27] H. C. An. Research on software problems based on ecological angle. In *2009 International Conference on Environmental Science and Information Application Technology*, volume 3, pages 11–14, July 2009.

[28] Bedir Tekinerdogan and Huub Scholten. Software ecosystems for the life sciences application domains. *SIGSOFT Softw. Eng. Notes*, 40(6):1–6, November 2015.

[29] K. A. Basit and M. Matskin. Towards an engineering of multi-agent eco-systems. In *4th IEEE International Conference on Digital Ecosystems and Technologies*, pages 340–345, April 2010.

[30] Jens Knodel, Matthias Naab, and Dominik Rost. Supporting architects in mastering the complexity of open software ecosystems. In *Proceedings of the 2014 European Conference on Software Architecture Workshops*, ECSAW '14, pages 13:1–13:6, New York, NY, USA, 2014. ACM.

[31] Gary Chastek and John D McGregor. It takes an ecosystem. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2012.

[32] Kevin Daniel André Carillo and Josianne Marsan. "the dose makes the poison"—exploring the toxicity phenomenon in online communities. *ICIS 2016 Proceedings*, 6, 2016.

[33] Kevin Daniel André Carillo, Josianne Marsan, and Bogdan Negoita. Towards developing a theory of toxicity in the context of free/open source software & peer production communities. *SIGOPEN 2016*, 2016.

[34] Colin C. Venters, Rafael Capilla, Stefanie Betz, Birgit Penzenstadler, Tom Crick, Steve Crouch, Elisa Yumi Nakagawa, Christoph Becker, and Carlos Carrillo. Software sustainability: Research and practice from a software architecture viewpoint. *Journal of Systems and Software*, 138:174–188, 2018.

[35] Rodrigo Pereira dos Santos, Cristiano Maciel, and José Viterbo. *Software Ecosystems, Sustainability and Human Values in the Social Web: 8th Workshop of Human-Computer Interaction Aspects to the Social Web, WAIHCWS 2017, Joinville, Brazil, October 23, 2017 and 9th Workshop, WAIHCWS 2018, Belém, Brazil, October 22, 2018, Revised Selected Papers*, volume 1081. Springer Nature, 2020.

[36] Jonas Gamalielsson and Björn Lundell. Sustainability of open source software communities beyond a fork: How and why has the libreoffice project evolved? *Journal of Systems and Software*, 89:128–145, 2014.

[37] Zhifang Liao, Libing Deng, Xiaoping Fan, Yan Zhang, Hui Liu, Xiaofei Qi, and Yun Zhou. Empirical research on the evaluation model and method of sustainability of the open source ecosystem. *Symmetry*, 10(12), 2018.

[38] Daniel Curto-Millet and Alberto Corsín Jiménez. The sustainability of open source commons. *European Journal of Information Systems*, 0(0):1–19, 2022.

[39] Zhixing Li, Yue Yu, Tao Wang, Gang Yin, ShanShan Li, and Huaimin Wang. Are you still working on this? an empirical study on pull request abandonment. *IEEE Transactions on Software Engineering*, 48(6):2173–2188, 2022.

[40] Sandra L. Ramírez-Mora and Hanna Oktaba. Team maturity in agile software development: The impact on productivity. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 732–736, 2018.

[41] K Zielinski and T Szmuc. Software process maturity and the success of free software projects. *Frontiers in Artificial Intelligence and Applications*, page 3, 2005.

[42] Suhaib Mujahid, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, Mohamed Aymen Saied, and Bram Adams. Toward using package centrality trend to identify packages in decline. *IEEE Transactions on Engineering Management*, pages 1–15, 2021.

[43] Song Wang, Junjie Wang, Jaechang Nam, and Nachiappan Nagappan. *Continuous Software Bug Prediction*. Association for Computing Machinery, New York, NY, USA, 2021.

[44] Xiaoqing Wang and Shannon Lantzy. A systematic examination of member turnover and online community health. In *A Systematic Examination of Member Turnover and Online Community Health.*, volume 4, 01 2011.

[45] Sami Hyrynsalmi, Marko Seppänen, Tiina Nokkala, Arho Suominen, and Antero Järvi. Wealthy, healthy and/or happy—what does 'ecosystem health'stand for? In *International Conference of Software Business*, pages 272–287. Springer, 2015.

[46] Slinger Jansen, Anthony Finkelstein, and Sjaak Brinkkemper. A sense of community: A research agenda for software ecosystems. In *2009 31st International Conference on Software Engineering - Companion Volume*, pages 187–190, 2009.

[47] Vasilis Boucharas, Slinger Jansen, et al. Formalizing software ecosystem modeling. In *Proceedings of the 1st International Workshop on Open Component Ecosystems*, page 41–50, NY, USA, 2009. ACM.

[48] JV Joshua, DO Alao, SO Okolie, and O Awodele. Software ecosystem: Features, benefits and challenges. *International Journal of Advanced Computer Science and Applications*, 2, 2013.

[49] Emad Shihab, Christian Bird, and Thomas Zimmermann. The effect of branching strategies on software quality. In *International Symposium on Empirical Software Engineering and Measurement*, pages 301–310. ACM/IEEE, 2012.

[50] Christian Bird and Thomas Zimmermann. Assessing the value of branches with what-if analysis. In *International Symposium on Foundations of Software Engineering*. ACM SIGSOFT, 2012.

[51] MEJ Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46(5):323–351, 2005.

[52] Gaurav G. Sharma and Klaas-Jan Stol. Exploring onboarding success, organizational fit, and turnover intention of software professionals. *Journal of Systems and Software*, 159:110442, 2020.

[53] Fabian Fagerholm, Alejandro S Guinea, Jürgen Münch, and Jay Borenstein. The role of mentoring and project characteristics for onboarding in open source software

projects. In *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*, page 55. ACM, 2014.

[54] Giovanni Viviani and Gail C. Murphy. Reflections on onboarding practices in mid-sized companies. In *Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '19, pages 83–84, Piscataway, NJ, USA, 2019. IEEE Press.

[55] Ricardo Britto, Daniela S Cruzes, Darja Smite, and Aivars Sablis. Onboarding software developers and teams in three globally distributed legacy projects: A multi-case study. *Journal of Software: Evolution and Process*, 30(4):e1921, 2018.

[56] Igor Steinmacher, Ana Paula Chaves, Tayana Uchoa Conte, and Marco Aurelio Gerosa. Preliminary empirical identification of barriers faced by newcomers to open source software projects. In *2014 Brazilian Symposium on Software Engineering*, pages 51–60. IEEE, 2014.

[57] Casey Casalnuovo, Bogdan Vasilescu, Premkumar Devanbu, and Vladimir Filkov. Developer onboarding in github: the role of prior social links and language experience. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 817–828. ACM, 2015.

[58] A. Labuschagne and R. Holmes. Do onboarding programs work? In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 381–385, May 2015.

[59] Yuxia Zhang, Minghui Zhou, Klaas-Jan Stol, Jianyu Wu, and Zhi Jin. How do companies collaborate in open source ecosystems? an empirical study of openstack.

2020 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2020.

[60] Jean-Gabriel Young, Amanda Casari, Katie McLaughlin, et al. Which contributions count? analysis of attribution in open source. In *International Working Conference on Mining Software Repositories*. IEEE, 2021.

[61] J. Whitehead. Collaboration in software engineering: A roadmap. In *Future of Software Engineering (FOSE '07)*, pages 214–225, May 2007.

[62] T. Mens, M. Cataldo, and D. Damian. The social developer: The future of software development. *IEEE Software*, 36(1):4, Jan 2019.

[63] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. Assessing code authorship: The case of the linux kernel. In Federico Balaguer, Roberto Di Cosmo, Alejandra Garrido, Fabio Kon, Gregorio Robles, and Stefano Zacchiroli, editors, *Open Source Systems: Towards Robust Practices*, pages 151–163, Cham, 2017. Springer International Publishing.

[64] Armstrong Foundjem, Ellis E. Eghan, and Bram Adams. Onboarding vs. diversity, productivity and quality – empirical study of the OpenStack ecosystem. In *International Conference on Software Engineering*, pages 1033–1045, 2021.

[65] Casey Casalnuovo, Bogdan Vasilescu, Premkumar Devanbu, and Vladimir Filkov. Developer onboarding in github: The role of prior social links and language experience. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 817–828, New York, NY, USA, 2015. ACM.

[66] Igor Steinmacher, Marco Aurélio Graciotto Silva, and Marco Aurélio Gerosa. Barriers faced by newcomers to open source projects: A systematic review. In Luis Corral, Alberto Sillitti, Giancarlo Succi, Jelena Vlasenko, and Anthony I. Wasserman, editors, *Open Source Software: Mobile Open Source Technologies*, pages 153–163, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[67] Austin Z. Henley, Kivanç Muçlu, Maria Christakis, et al. CFar: A tool to increase communication, productivity, and review quality in collaborative code reviews. In *CHI*, pages 1–13. ACM, 2018.

[68] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 192–201, New York, NY, USA, 2014. ACM.

[69] Laura Plonka, Helen Sharp, Janet van der Linden, and Yvonne Dittrich. Knowledge transfer in pair programming: An in-depth analysis. *International Journal of Human-Computer Studies*, 73:66–78, 2015.

[70] Kim Man Lui and Keith C.C. Chan. Pair programming productivity: Novice–novice vs. expert–expert. *International Journal of Human-Computer Studies*, 64(9):915–925, 2006.

[71] A. Heinzl K. Spohrer, T. Kude and C. T. Schmidt. Peer-based quality assurance in information systems and development: A transactive memory perspective. In *International Conference on Information Systems*, 2013.

[72] Bram Adams, Ryan Kavanagh, Ahmed E. Hassan, and Daniel M. German. An empirical study of integration activities in distributions of open source software. *Empirical Software Engineering*, 21(3):960–1001, 2016.

[73] B. Adams and S. McIntosh. Modern release engineering in a nutshell – why researchers should care. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 78–90, March 2016.

[74] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams. Do faster releases improve software quality? an empirical case study of mozilla firefox. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 179–188, June 2012.

[75] Guenther Ruhe. *Product Release Planning: Methods, Tools and Applications*. CRC Press, 2010.

[76] M. T. Rahman and P. C. Rigby. Release stabilization on linux and chrome. *IEEE Software*, 32(2):81–88, Mar.-Apr. 2015.

[77] Germán Poo-Caamaño, Leif Singer, Eric Knauss, and Daniel M. German. Herding cats: A case study of release management in an open collaboration ecosystem. In Kevin Crowston, Imed Hammouda, Björn Lundell, Gregorio Robles, Jonas Gamalielsson, and Juho Lindman, editors, *Open Source Systems: Integrating Communities*, pages 147–162, Cham, 2016. Springer International Publishing.

[78] E. Constantinou and T. Mens. Socio-technical evolution of the ruby ecosystem in github. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 34–44, Feb 2017.

[79] José Apolinário Teixeira and Helena Karsten. Managing to release early, often and on time in the openstack software ecosystem. *Journal of Internet Services and Applications*, 10(1):7, Apr 2019.

[80] A. Decan, T. Mens, and M. Claes. An empirical comparison of dependency issues in oss packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 2–12, Feb 2017.

[81] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an api: Cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 109–120, New York, NY, USA, 2016. ACM.

[82] Rashina Hoda. Socio-technical grounded theory for software engineering. *IEEE Transactions on Software Engineering*, pages 1–26, 2021.

[83] Rashina Hoda. Decoding grounded theory for software engineering. In *2021 IEEE/ACM 43rd ICSE-Companion*, pages 326–327, may 2021.

[84] Fabio Palomba et al. Beyond technical aspects: How do community smells influence the intensity of code smells? *IEEE TSE*, 47(1):108–129, January 2021.

[85] Zainab Masood, Rashina Hoda, and Kelly Blincoe. Real world scrum a grounded theory of variations in practice. *IEEE Transactions on Software Engineering*, pages 1–13, 2020.

[86] Suprateek Sarker, Francis Lau, and Sundeep Sahay. Using an adapted grounded theory approach for inductive theory building about virtual team development. *SIGMIS Database*, 32(1):38–56, dec 2001.

[87] Leonardo Sousa et al. Identifying design problems in the source code: A grounded theory. In *2018 IEEE/ACM 40th ICSE*, pages 921–931, 2018.

[88] Pilar Rodriguez, Cathy Urquhart, and Emilia Mendes. A theory of value for value-based feature selection in software engineering. *IEEE Transactions on Software Engineering*, pages 1–28, 2020.

[89] Todd Sedano, Paul Ralph, and Cécile Péraire. Software development waste. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 130–140, 2017.

[90] Yogeshwar Shastri, Rashina Hoda, and Robert Amor. The role of the project manager in agile software development projects. *Journal of Systems and Software*, 173:110871, 2021.

[91] Yogeshwar Shastri, Rashina Hoda, and Robert Amor. Spearheading agile: The role of the scrum master in agile projects. *Empirical Software Engineering*, 26(1):1–31, 2021.

[92] Rashina Hoda, James Noble, and Stuart Marshall. Self-organizing roles on agile software development teams. *IEEE Transactions on Software Engineering*, 39(3):422–444, 2013.

[93] Yuxia Zhang et al. How do companies collaborate in open source ecosystems? an empirical study of openstack. In *ACM/IEEE 42nd ICSE*, Icse '20, page 1196–1208, New York, NY, USA, 2020. ACM.

[94] Jose Teixeira. Understanding collaboration in the open-source arena: The cases of webkit and openstack. In *18th EASE*, EASE '14, New York, NY, USA, 2014. Association for Computing Machinery.

[95] Armstrong Foundjem. *Cross-Distribution Feedback in Software Ecosystems*, page 723–724. ACM, New York, NY, USA, 2020.

[96] Yuxia Zhang, Hao He, and Minghui Zhou. Commercial participation in openstack: Two sides of a coin. *Computer*, 55(2):78–84, 2022.

[97] Armstrong Foundjem, Ellis Eghan, and Bram Adams. Onboarding vs. diversity, productivity and quality — empirical study of the openstack ecosystem. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1033–1045, 2021.

[98] Denis Silakov. Abf: A farm for building cross-distribution linux software. In *10th Central and Eastern European Software Engineering Conference in Russia*, CEE-SECR '14. ACM, 2014.

[99] Vincent Boisselle and Bram Adams. The impact of cross-distribution bug duplicates, empirical study on debian and ubuntu. In *15th IEEE SCAM*, pages 131–140, Bremen, 2015. IEEE Computer Society.

[100] Muhammad Suleman et al. Empirical research and auxiliary tool for custom android roms. In *2020 ISCEIC*, pages 14–18, 2020.

[101] Hui Ding et al. An empirical study on downstream workarounds for cross-project bugs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 318–327, 2017.

[102] Germán Poo-Caamaño, Eric Knauss, Leif Singer, and Daniel German. Herding cats in a foss ecosystem: a tale of communication and coordination for release management. *Journal of Internet Services and Applications*, 8, 12 2017.

[103] Zhendong Wang, Yang Feng, Yi Wang, James A. Jones, and David Redmiles. Unveiling elite developers' activities in open source projects. *ACM Trans. Softw. Eng. Methodol.*, 29(3), June 2020.

[104] Shahab Bayati. Understanding newcomers success in open source community. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 224–225, New York, NY, USA, 2018. Association for Computing Machinery.

[105] C. Liu, D. Yang, X. Zhang, H. Hu, J. Barson, and B. Ray. Poster: A recommender system for developer onboarding. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 319–320, May 2018.

[106] I. Steinmacher, C. Treude, and M. A. Gerosa. Let me in: Guidelines for the successful onboarding of newcomers to open source projects. *IEEE Software*, 36(4):41–49, July 2019.

[107] Ifraz Rehman, Dong Wang, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Newcomer candidate: Characterizing contributions of a novice developer to github. *arXiv preprint arXiv:2008.02597*, 2020.

[108] Y. Zhang, M. Zhou, A. Mockus, and Z. Jin. Companies' participation in oss development - an empirical study of openstack. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[109] Swapnil Bhartiya. What open means to openstack. `https://www.linuxfoundation.org/blog/2017/12/what-open-means-openstack/`, December 2017.

[110] Yvonne Dittrich. Software engineering beyond the project – sustaining software ecosystems. *Information and Software Technology*, 56(11):1436–1456, 2014. Special issue on Software Ecosystems.

[111] OpenStack-Foundation. Links to oui archived materials. https://www.openstack.org, [OUI] https://docs.openstack.org/upstream-training/upstream-archives.html, 2020. Accessed: 2020-08-28.

[112] Daniel Izquierdo, Nicole Huesman, Alexander Serebrenik, and Gregorio Robles. Openstack gender diversity report. *IEEE Software*, 36(1):28–33, 2018.

[113] A. N. Meyer, G. C. Murphy, T. Zimmermann, and T. Fritz. Enabling good work habits in software developers through reflective goal-setting. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[114] Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, and Christian Kästner. Exploring differences and commonalities between feature flags and configuration options. In

Proc. Int'l Conf. Software Engineering–Software Engineering in Practice (ICSE-SEIP). ACM, 2020.

[115] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. Here we go again: Why is it difficult for developers to learn another programming language? In *Proceedings of the 42nd International Conference on Software Engineering, ICSE*, 2020.

[116] A. Rahman, C. Parnin, and L. Williams. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 164–175, 2019.

[117] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: A case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, page 181–190, New York, NY, USA, 2018. Association for Computing Machinery.

[118] Sigeru Mizuno. *Management for quality improvement: the 7 new QC tools*. CRC Press, 2020.

[119] J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1), 1977.

[120] Armstrong Foundjem. Replication packages for our study on seco onboarding – scripts/dataset. http://doi.org/10.5281/zenodo.4457683, 2021. Accessed: 2021-01-22.

[121] S. Yeasmin and L. A. Albabtain. Escape the countries: A vr escape room game. In *2020 3rd International Conference on Computer Applications Information Security (ICCAIS)*, pages 1–6, 2020.

[122] Christopher Mendez, Anita Sarma, and Margaret Burnett. Gender in open source software: What the tools tell. In *Proceedings of the 1st International Workshop on Gender Equality in Software Engineering*, GE '18, page 21–24, New York, NY, USA, 2018. Association for Computing Machinery.

[123] K. Kohl Silveira, S. Musse, I. H. Manssour, R. Vieira, and R. Prikladnicki. Confidence in programming skills: Gender insights from stackoverflow developers survey. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 234–235, 2019.

[124] A. Filippova, E. Trainer, and J. D. Herbsleb. From diversity by numbers to diversity as process: Supporting inclusiveness in software development teams with brainstorming. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 152–163, 2017.

[125] Mohsin Alvi. *A manual for selecting sampling techniques in research*. https://mpra.ub.uni-muenchen.de/70218/, 2016.

[126] CHAOSS Project Linux Foundation. Community health analytics open source software. https://chaoss.community/metrics/, 2020. Accessed: 2020-08-28.

[127] André N. Meyer, Gail C. Murphy, Thomas Fritz, and Thomas Zimmermann. *Developers' Diverging Perceptions of Productivity*, pages 137–146. Apress, Berkeley, CA, 2019.

[128] Slinger Jansen. Measuring the health of open source software ecosystems: Beyond the scope of project health. *Information and Software Technology*, 56(11):1508–1519, 2014.

[129] Paidamoyo Chapfuwa, Chunyuan Li, Nikhil Mehta, Lawrence Carin, and Ricardo Henao. Survival cluster analysis. In *Proceedings of the ACM Conference on Health, Inference, and Learning*, CHIL '20, page 60–68, New York, NY, USA, 2020. Association for Computing Machinery.

[130] Somanath D Pawar and Digambar T Shirke. Nonparametric tests for multivariate multi-sample locations based on data depth. *Journal of Statistical Computation and Simulation*, 89(9):1574–1591, 2019.

[131] Mike Allen. The SAGE Encyclopedia of Communication Research Methods. *The SAGE Encyclopedia of Communication Research Methods*, December 2019.

[132] F. Armstrong, F. Khomh, and B. Adams. Broadcast vs. unicast review technology: Does it matter? In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 219–229, March 2017.

[133] Corey Jergensen, Anita Sarma, and Patrick Wagstrom. The onion patch: Migration in open source ecosystems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 70–80, New York, NY, USA, 2011. Association for Computing Machinery.

[134] Sebastian Hönel, Morgan Ericsson, Welf Löwe, and Anna Wingkvist. Importance and aptitude of source code density for commit classification into maintenance activities. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 109–120. IEEE, 2019.

[135] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. Revisiting and improving szz implementations. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12. IEEE, 2019.

[136] Christopher Mendez, Hema Susmita Padala, Zoe Steine-Hanson, Claudia Hilderbrand, Amber Horvath, Charles Hill, Logan Simpson, Nupoor Patil, Anita Sarma, and Margaret Burnett. Open source barriers to entry, revisited: A sociotechnical perspective. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1004–1015, New York, NY, USA, 2018. Association for Computing Machinery.

[137] G. Avelino, E. Constantinou, M. T. Valente, and A. Serebrenik. On the abandonment and survival of open source projects: An empirical investigation. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, Sep. 2019.

[138] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the pypi ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*,

ESEC/FSE 2018, page 644–655, New York, NY, USA, 2018. Association for Computing Machinery.

[139] Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Assessing the bus factor of git repositories. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 499–503. IEEE, 2015.

[140] I. Steinmacher, I. Wiese, A. P. Chaves, and M. A. Gerosa. Why do newcomers abandon open source software projects? In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 25–32, May 2013.

[141] Jun Li, Gan Sun, Guoshuai Zhao, and H Lehman Li-wei. Robust low-rank discovery of data-driven partial differential equations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 767–774, 2020.

[142] Huilian Sophie Qiu, Alexander Nolte, Anita Brown, Alexander Serebrenik, and Bogdan Vasilescu. Going farther together: The impact of social capital on sustained participation in open source. In *Proceedings of the 41st International Conference on Software Engineering (ICSE) 2019, Montreal, Canada*. IEEE, 2019. to appear.

[143] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 908–911, New York, NY, USA, 2018. Association for Computing Machinery.

[144] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[145] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. Automatic identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 81–90. IEEE, 2006.

[146] Caitlin Sadowski, Chris Lewis, Zhongpeng Lin, Xiaoyan Zhu, and E James Whitehead Jr. An empirical analysis of the fixcache algorithm. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 219–222. ACM, 2011.

[147] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. The impact of refactoring changes on the szz algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 380–390. IEEE, 2018.

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

[149] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. Szz unleashed: An open implementation of the szz algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, MaLTeSQuE 2019, page 7–12, New York, NY, USA, 2019. Association for Computing Machinery.

[150] Shao-Fang Wen. Learning secure programming in open source software communities: A socio-technical view. In *Proceedings of the 6th International Conference on Information and Education Technology*, ICIET '18, page 25–32, New York, NY, USA, 2018. Association for Computing Machinery.

[151] Paige Rodeghero, Thomas Zimmermann, Brian Houck, and Denae Ford. Please turn your cameras on: Remote onboarding of software developers during a pandemic. *arXiv preprint arXiv:2011.08130*, 2020.

[152] Catarina de Souza Costa, Jose Jair Figueiredo, Joao Felipe Pimentel, et al. Recommending participants for collaborative merge sessions. *Trans. Softw. Eng.*, pages 1–1, 2019.

[153] Catarina Costa, Jair Figueiredo, Leonardo Murta, et al. TIPMerge: recommending experts for integrating changes across branches. In *International Symposium on Foundations of Software Engineering*, pages 523–534, 2016.

[154] Stephan Krusche, Mjellma Berisha, and Bernd Bruegge. Teaching code review management using branch based workflows. In *International Conference on Software Engineering*, pages 384–393. ACM, 2016.

[155] Md Tajmilur Rahman. Investigating modern release engineering practices. In *International Conference on Software Analysis, Evolution, and Reengineering*, pages 607–608. IEEE, 2015.

[156] Subhajit Datta. How does developer interaction relate to software quality? An examination of product development data. *Empirical Software Engineering*, 23(3):1153–1187, June 2018.

[157] Torgeir Dingsøyr, Nils Brede Moe, Tor Erlend Fægri, et al. Exploring software development at the very large-scale: A revelatory case study and research agenda for agile method adaptation. *Empirical Software Engineering*, 23:490–520, 2018.

[158] Eirini Kalliamvakou, Daniela Damian, Kelly Blincoe, et al. Open source-style collaborative development practices in commercial projects using GitHub. In *International Conference on Software Engineering*, pages 574–585. IEEE, 2015.

[159] S. Bick, K. Spohrer, R. Hoda, et al. Coordination challenges in large-scale software development: A case study of planning misalignment in hybrid settings. *Trans. Softw. Eng.*, 44(10):932–950, 2018.

[160] Thomas Neumayr, Hans-Christian Jetter, Mirjam Augstein, et al. Domino: A descriptive framework for hybrid collaboration and coupling styles in partially distributed teams. *Human-Computer Interaction*, page 24, November 2018.

[161] Fatemeh Mardi, Keith Miller, and Phyllis Balcerzak. Novice - expert pair coaching: Teaching Python in a pandemic. In *Technical Symposium on Computer Science Education*, pages 226–231. ACM, 2021.

[162] Helen Sharp and Hugh Robinson. Collaboration and co-ordination in mature extreme programming teams. *International Journal of Human-Computer Studies*, 66(7):506–518, 2008.

[163] Per Runeson, Martin Host, Austen Rainer, et al. *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley, 1st edition, 2012.

[164] John W. Creswell. *Research Design – Qualitative, Quantitative and Mixed Methods Approaches*. SAGE Publications, 3rd edition, 2009.

[165] Armstrong Foundjem, Eleni Constantinou, Tom Mens, et al. Replication package — v2.0.0). https://doi.org/10.5281/zenodo.5759968, December 2021. Online.

[166] Armstrong Foundjem and Bram Adams. Release synchronization in software ecosystems. *Empirical Software Engineering*, 26(34), 2021.

[167] Yuxia Zhang, Minghui Zhou, Audris Mockus, et al. Companies' participation in oss development–an empirical study of openstack. *IEEE Transactions on Software Engineering*, 47(10):2242–2259, 2021.

[168] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. How has forking changed in the last 20 years? A study of hard forks on GitHub. In *International Conference on Software Engineering*, pages 445–456. ACM, 2020.

[169] M. Mukadam, C. Bird, and P. C. Rigby. Gerrit software code review data from Android. In *International Working Conference on Mining Software Repositories*, pages 45–48, 2013.

[170] Xin Xia, David Lo, Xinyu Wang, et al. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 261–270, 2015.

[171] Christian Bird. Interviews. In *Perspectives on Data Science for Software Engineering*. Morgan Kaufmann, 2016.

[172] Carolyn D Egelman, Emerson Murphy-Hill, Elizabeth Kammer, et al. Predicting developers' negative feelings about code review. In *International Conference on Software Engineering*, pages 174–185. IEEE, 2020.

[173] Norsaremah Salleh, Rashina Hoda, Moon Ting Su, Tanjila Kanij, and John Grundy. Recruitment, engagement and feedback in empirical software engineering studies in industrial contexts. *Information and Software Technology*, 98:161–172, 2018.

[174] Greg Guest, Arwen Bunce, and Laura Johnson. How many interviews are enough? an experiment with data saturation and variability. *Field methods*, 18(1):59–82, 2006.

[175] Patricia I Fusch and Lawrence R Ness. Are we there yet? data saturation in qualitative research. *The qualitative report*, 20(9):1408, 2015.

[176] Miryung Kim, Thomas Zimmermann, Robert DeLine, et al. The emerging role of data scientists on software development teams. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 96–107. IEEE, 2016.

[177] André N Meyer, Earl T Barr, Christian Bird, et al. Today was a good day: The daily life of software developers. *IEEE Transactions on Software Engineering*, 47(5):863–880, 2019.

[178] Naundefineda Terzimehić, Renate Häuslschmid, Heinrich Hussmann, et al. A review & analysis of mindfulness research in HCI. In *ICHF in Computing Systems*, pages 1–13. ACM, 2019.

[179] Per Lenberg, Lars Göran Wallgren Tengberg, and Robert Feldt. An initial analysis of software engineers' attitudes towards organizational change. *Empirical Software Engineering*, 22(4):2179–2205, 2017.

[180] D. S. Cruzes and T. Dyba. Recommended steps for thematic synthesis in software engineering. In *International Symposium on Empirical Software Engineering and Measurement*, pages 275–284, 2011.

[181] D. Arya, W. Wang, J. L. C. Guo, et al. Analysis and detection of information types of open source software issue discussions. In *International Conference on Software Engineering*, pages 454–464, 2019.

[182] Julia Himmelsbach, Stephanie Schwarz, Cornelia Gerdenitsch, et al. Do we care about diversity in human computer interaction. In *International Conference on Human Factors in Computing Systems*, pages 1–16. ACM, 2019.

[183] H Russell Bernard, Amber Wutich, and Gery W Ryan. *Analyzing qualitative data: Systematic approaches*. SAGE publications, 2016.

[184] Marcia W. DiStaso and Denise Sevick Bortree. Multi-method analysis of transparency in social media practices: Survey, interviews and content analysis. *Public Relations Review*, 38(3):511–514, 2012. Public Relations History.

[185] Johnny Saldaña. *The coding manual for qualitative researchers*. Sage, 2015.

[186] Md Shariful Islam, Wael Khreich, and Abdelwahab Hamou-Lhadj. Anomaly detection techniques based on kappa-pruned ensembles. *IEEE Transactions on Reliability*, 67(1):212–229, 2018.

[187] John L. Campbell, Charles Quincy, Jordan Osserman, and Ove K. Pedersen. Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. *Sociological Methods & Research*, 42(3):294–320, 2013.

[188] Ons Mlouki, Foutse Khomh, and Giuliano Antoniol. On the detection of licenses violations in the android ecosystem. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 382–392. IEEE, 2016.

[189] Mathieu Goeminne and Tom Mens. A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 78:971–986, 2013.

[190] Christian Bird, Alex Gourley, Prem Devanbu, et al. Mining email social networks. In *International Working Conference on Mining Software Repositories*, pages 137–143. ACM, 2006.

[191] Mehdi Golzadeh, Alexandre Decan, Eleni Constantinou, et al. Identifying bot activity in github pull request and issue comments. In *2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE)*, pages 21–25, 2021.

[192] Nathan Cassee, Christos Kitsanelis, Eleni Constantinou, et al. Human, bot or both? a study on the capabilities of classification models on mixed accounts. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 654–658, 2021.

[193] Daniel M. German, Bram Adams, and Ahmed E. Hassan. Continuously mining distributed version control systems: An empirical study of how Linux uses git. *Empirical Software Engineering*, 21(1):260–299, February 2016.

[194] Y. Fan, X. Xia, D. Alencar da Costa, et al. The impact of changes mislabeled by SZZ on just-in-time defect prediction. *Trans. Softw. Eng.*, page 26, 2019.

[195] E. C. Neto, D. A. d. Costa, and U. Kulesza. Revisiting and improving SZZ implementations. In *International Symposium on Empirical Software Engineering and Measurement*, pages 1–12, 2019.

[196] Markus Borg, Oscar Svensson, Kristian Berg, et al. SZZ unleashed: An open implementation of the SZZ algorithm. In *MaLTeSQuE*, pages 7–12. ACM, 2019.

[197] Le An, Foutse Khomh, and Yann-Gaël Guéhéneuc. An empirical study of crash-inducing commits in mozilla firefox. *Software Quality Journal*, 26(2):553–584, Jun 2018.

[198] Gema Rodriguez, Gregorio Robles, and Jesus Gonzalez-Barahona. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. *Information and Software Technology*, 03 2018.

[199] Ming Wen, Rongxin Wu, Yepang Liu, et al. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Joint Meeting on ESEC and FSE*, pages 326–337. ACM, 2019.

[200] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA, 2018. ACM Press.

[201] Mary L McHugh. The chi-square test of independence. *Biochemia medica: Biochemia medica*, 23(2):143–149, 2013.

[202] Albert Satorra and Peter M Bentler. A scaled difference chi-square test statistic for moment structure analysis. *Psychometrika*, 66(4):507–514, 2001.

[203] Janet Siegmund, Christian Kästner, Jörg Liebig, et al. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.

[204] Horst Treiblmaier and Peter Filzmoser. Exploratory factor analysis revisited: How robust methods support the detection of hidden multivariate data structures in is research. *Information & Management*, 47(4):197–207, 2010.

[205] Georgy Meshcheryakov, Anna A Igolkina, and Maria G Samsonova. semopy 2: A structural equation modeling package with random effects in python. *arXiv preprint arXiv:2106.01140*, 2021.

[206] Anandasivam Gopal, Tridas Mukhopadhyay, and Mayuram S Krishnan. The impact of institutional forces on software metrics programs. *Trans. Softw. Eng.*, 31(8):679–694, 2005.

[207] David Richard Johnson and James C Creech. Ordinal measures in multiple indicator models: A simulation study of categorization error. *American Sociological Review*, pages 398–407, 1983.

[208] Nicole Forsgren, Margaret-Anne Storey, Chandra Maddila, et al. The SPACE of developer productivity: There's more to it than you think. *Queue*, 19(1):20–48, February 2021.

[209] Daniel Izquierdo-Cortazar, Nelson Sekitoleko, Jesus M. Gonzalez-Barahona, et al. Using metrics to track code review performance. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*,

EASE'17, pages 214–223, New York, NY, USA, 2017. Association for Computing Machinery.

[210] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. Code review quality: How developers see it. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1028–1038, New York, NY, USA, 2016. Association for Computing Machinery.

[211] Vladimir Kovalenko and Alberto Bacchelli. Code review for newcomers: Is it different? In *Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '18, pages 29–32, New York, NY, USA, 2018. Association for Computing Machinery.

[212] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 192–201, New York, NY, USA, 2014. Association for Computing Machinery.

[213] Shane McIntosh, Yasutaka Kamei, Bram Adams, et al. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.

[214] Edson Oliveira, Eduardo Fernandes, Igor Steinmacher, et al. Code and commit metrics of developer productivity: a study on team leaders perceptions. *Empirical Software Engineering*, 25(4):2519–2549, 2020.

[215] Peter Rigby, Brendan Cleary, Frederic Painchaud, et al. Contemporary peer review in action: Lessons from open source development. *IEEE Software*, 29(6):56–61, 2012.

[216] Canon Tong, Stanley Kam-Sing Wong, and Ken Pui-Hing Lui. The influences of service personalization, customer satisfaction and switching costs on e-loyalty. *International Journal of Economics and Finance*, 4(3):105–114, 2012.

[217] Laura Klem. Structural equation modeling. In *Reading and understanding MORE multivariate statistics.*, pages 227–260. American Psychological Association, Washington, DC, US, 2000.

[218] Richard P Bagozzi and Youjae Yi. Specification, evaluation, and interpretation of structural equation models. *Journal of the academy of marketing science*, 40(1):8–34, 2012.

[219] Anna A. Igolkina and Georgy Meshcheryakov. semopy: A python package for structural equation modeling. *Structural Equation Modeling: A Multidisciplinary Journal*, 27(6):952–963, 2020.

[220] Judea Pearl. The causal foundations of structural equation modeling. Technical report, California Univ Los Angeles Dept of Computer Science, 2012.

[221] Stephen G West, Aaron B Taylor, Wei Wu, et al. Model fit and model selection in structural equation modeling. *Handbook of structural equation modeling*, 1:209–231, 2012.

[222] Aage B Sørensen. 10. theoretical mechanisms and the empirical study of social processes. *Social mechanisms: An analytical approach to social theory*, page 238, 1998.

[223] Hyuncheol Kang and Jung-Won Ahn. Model setting and interpretation of results in research using structural equation modeling: A checklist with guiding questions for reporting. *Asian Nursing Research*, 15(3):157–162, 2021.

[224] Reza Ghaiumy Anaraky, Yao Li, and Bart Knijnenburg. Difficulties of measuring culture in privacy studies. *Proc. ACM Hum.-Comput. Interact.*, 5(CSCW2), oct 2021.

[225] Tanya N Beran and Claudio Violato. Structural equation modeling in medical research: a primer. *BMC research notes*, 3(1):1–10, 2010.

[226] Raphael Vallat. Pingouin: statistics in python. *Journal of Open Source Software*, 3(31):1026, 2018.

[227] Jason T Rich, J Gail Neely, Randal C Paniello, et al. A practical guide to understanding kaplan-meier curves. *Otolaryngology–head and neck surgery*, 143(3):331–6, 2010.

[228] Michele Tufano, Fabio Palomba, Gabriele Bavota, et al. When and why your code starts to smell bad (and whether the smells go away). *Trans. Softw. Eng.*, 43(11):1063–1088, 2017.

[229] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):4994–509, November 1993.

[230] M. Hess and J. Kromrey. Robust confidence intervals for effect sizes: A comparative study of Cohen's d and Cliff's delta under non-normality and heterogeneous variances. *AERA*, pages 1–30, 2004.

[231] G David Garson. *Path analysis*. Statistical Associates Publishing Asheboro, NC, 2013.

[232] Frederick P. Brooks Jr. *The mythical man-month*. Addison-Wesley Reading, MA, United States, 1974.

[233] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4):531–577, August 2012.

[234] Chris Bogart, Christian Kästner, James Herbsleb, et al. When and how to make breaking changes: Policies and practices in 18 open source software ecosystems. *Trans. Softw. Eng. Methodol.*, 30(4), July 2021.

[235] Claes Wohlin, Per Runeson, Martin Höst, et al. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[236] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. Open source software peer review practices: A case study of the Apache server. In *International Conference on Software Engineering*, pages 541–550. ACM, 2008.

[237] Patricia I. Fusch and Lawrence R. Ness. Are we there yet? data saturation in qualitative research. *The Qualitative Report*, 20(9):1408–1416, Sep 07 2015. Copyright - Copyright The Qualitative Report Sep 7, 2015; Document feature - ; Last updated - 2015-10-19.

[238] D. M. German, B. Adams, and A. E. Hassan. The evolution of the r software ecosystem. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 243–252, March 2013.

[239] Konstantinos Plakidas, Srdjan Stevanetic, Daniel Schall, Tudor B. Ionescu, and Uwe Zdun. How do software ecosystems evolve? a quantitative assessment of the r ecosystem. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 89–98, New York, NY, USA, 2016. ACM.

[240] Dr. Marco Iansiti and Gregory L. Richards. The information technology ecosystem: Structure, health, and performance. *The Antitrust Bulletin*, 51(1):77–110, 2006.

[241] Tom Mens and Mathieu Goeminne. Analysing the evolution of social aspects of open source software ecosystems. In *IWSECO@ ICSOB*, pages 1–14, 2011.

[242] Cleidson R.B. de Souza, Fernando Figueira Filho, Müller Miranda, Renato Pina Ferreira, Christoph Treude, and Leif Singer. The social side of software platform ecosystems. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 3204–3214, New York, NY, USA, 2016. ACM.

[243] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 102–112, May 2017.

[244] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010.

[245] Akond Ashfaque Ur Rahman, Eric Helms, Laurie Williams, and Chris Parnin. Synthesizing continuous deployment practices used in software development. In *Proceedings of the 2015 Agile Conference*, AGILE '15, pages 1–10, Washington, DC, USA, 2015. IEEE Computer Society.

[246] Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. Release practices in mobile apps -– users and developers perception. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 552–562, Osaka, Japan, March 2016.

[247] E. Shihab, Zhen Ming Jiang, and A. E. Hassan. On the use of internet relay chat (irc) meetings by developers of the gnome gtk+ project. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 107–110, May 2009.

[248] Daniel M. German. The gnome project: a case study of open source, global software development. *Software Process: Improvement and Practice*, 8(4):201–215, 2004.

[249] Forbes Gina Longoria. 3 reasons openstack may cost you less than aws. Technical report, www.forbes.com/, Oct 24, 2016.

[250] Anselm Strauss and Juliet Corbin. *Basics of qualitative research*. Sage publications, 1990.

[251] Marie C Hoepfl et al. Choosing qualitative research: A primer for technology education researchers. *Volume 9 Issue 1 (fall 1997)*, 1997.

[252] Marcia W. DiStaso and Denise Sevick Bortree. Multi-method analysis of transparency in social media practices: Survey, interviews and content analysis. *Public Relations Review*, 38(3):511–514, 2012. Public Relations History.

[253] M. Drouhard, N. Chen, J. Suh, R. Kocielnik, V. Peña-Araya, K. Cen, Xiangyi Zheng, and C. R. Aragon. Aeonium: Visual analytics to support collaborative qualitative coding. In *2017 IEEE Pacific Visualization Symposium (PacificVis)*, pages 220–229, 2017.

[254] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice. *Proc. ACM Hum.-Comput. Interact.*, 3(CSCW), November 2019.

[255] Dedoose. Dedoose version 8.0.35, web application for managing, analyzing, and presenting qualitative and mixed method research data (2018). los angeles, ca: Sociocultural research consultants, llc. `https://www.dedoose.com/`, 2018.

[256] E. Laukkanen, M. Paasivaara, J. Itkonen, C. Lassenius, and T. Arvonen. Towards continuous delivery by reducing the feature freeze period: A case study. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 23–32, May 2017.

[257] Jan Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In Gary J. Chastek, editor, *Software Product Lines*, pages 257–271, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[258] F. Armstrong, F. Khomh, and B. Adams. Broadcast vs. unicast review technology: Does it matter? In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 219–229, March 2017.

[259] Mika V Mäntylä, Bram Adams, Foutse Khomh, Emelie Engström, and Kai Petersen. On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering*, 20(5):1384–1425, 2015.

[260] A. Dyck, R. Penners, and H. Lichter. Towards definitions for release engineering and devops. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, pages 3–3, 2015.

[261] Jonathan Grudin and John Pruitt. Personas, participatory design and product development: An infrastructure for engagement. In *Proc. PDC*, volume 2, 2002.

[262] D. Ford, T. Zimmermann, C. Bird, and N. Nagappan. Characterizing software engineering work with personas based on knowledge worker actions. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 394–403, 2017.

[263] Xiang Zhang, Hans-Frederick Brown, and Anil Shankar. Data-driven personas: Constructing archetypal users with clickstreams and user telemetry. In *Proceedings of the 2016 CHI conference on human factors in computing systems*, pages 5350–5359, 2016.

[264] Santiago Dueñas, Valerio Cosentino, Gregorio Robles, and Jesus M Gonzalez-Barahona. Perceval: software project data at your will. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 1–4. ACM, 2018.

[265] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. Automating large-scale data quality verification. *Proceedings of the VLDB Endowment*, 11(12):1781–1794, 2018.

[266] F. Liu and Y. Deng. A fast algorithm for network forecasting time series. *IEEE Access*, 7:102554–102560, 2019.

[267] H. Nguyen and C. K. Hansen. Short-term electricity load forecasting with time series analysis. In *2017 IEEE International Conference on Prognostics and Health Management (ICPHM)*, pages 214–221, 2017.

[268] Robert K Yin. *Case study research: Design and methods*. Sage publications, 2013.

[269] Blake M. DiCosola III and Gina Neff. Using social comparisons to facilitate healthier choices in online grocery shopping contexts. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI EA '20, page 1–8, New York, NY, USA, 2020. Association for Computing Machinery.

[270] D. A. Tamburri, R. Kazman, and H. Fahimi. The architect's role in community shepherding. *IEEE Software*, 33(6):70–79, Nov.-Dec. 2016.

[271] Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems – a systematic literature review. *Journal of Systems and Software*, 86(5):1294–1306, 2013.

[272] Armstrong Foundjem and Bram Adams. Release synchronization in software ecosystems. *Empirical Software Engineering*, 26(3):1–50, March 2021.

[273] Wanwangying Ma, Lin Chen, Xiangyu Zhang, et al. Impact analysis of cross-project bugs on software ecosystems. In *ICSE*, Icse '20, page 100–111, USA, 2020. ACM.

[274] Joyce Oliveira and Carina Alves. Software ecosystems governance – an analysis of sap and gnome platforms. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 296–299, 2021.

[275] Christopher Bogart, Christian Kästner, James Herbsleb, et al. How to break an api: Cost negotiation and community values in three software ecosystems. In *FSE*, Fse 2016, page 109–120, USA, 2016. ACM.

[276] Daniela Lettner, Florian Angerer, et al. A case study on software ecosystem characteristics in industrial automation software. In *Proceedings of the 2014 International Conference on Software and System Process*, page 40–49, NY, USA, 2014. ACM.

[277] James Tizard. Requirement mining in software product forums. In *2019 IEEE 27th International Requirements Engineering Conference (RE)*, pages 428–433, 2019.

[278] David Johnson, James Tizard, Daniela Damian, Kelly Blincoe, and Tony Clear. Open crowdre challenges in software ecosystems. In *2020 4th International Workshop on Crowd-Based Requirements Engineering (CrowdRE)*, pages 1–4, 2020.

[279] Z. Abou Khalil, E. Constantinou, T. Mens, et al. A longitudinal analysis of bug handling across eclipse releases. In *ICSME*, pages 1–12, Los Alamitos, October 2019. IEEE Computer Society.

[280] Bram Adams, Ryan Kavanagh, Ahmed E. Hassan, et al. An empirical study of integration activities in distributions of open source software. *EMSE*, 21(3):960–1001, June 2016.

[281] W. Ma et al. How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem. In *2017 IEEE/ACM 39th ICSE*, ICSE '17, pages 381–392, May 2017.

[282] Frederick P. Brooks. *The Mythical Man-Month: Essays on Softw*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1978.

[283] José Apolinário Teixeira and Helena Karsten. Managing to release early, often and on time in the openstack software ecosystem. *Journal of Internet Services and Applications*, 10(1):1–22, 2019.

[284] Daniel M. German, Bram Adams, and Ahmed E. Hassan. Continuously mining the use of distributed version control systems: An empirical study of how linux uses git. *Empirical Software Engineering*, 21(1):260–299, 2015.

[285] Forrest Shull, Janice Singer, and Dag IK Sjøberg. *Guide to advanced empirical software engineering*. Springer, 2007.

[286] Juliet M Corbin and Anselm Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative sociology*, 13(1):3–21, 1990.

[287] Barney G Glaser and Anselm L Strauss. *Discovery of grounded theory: Strategies for qualitative research*. Routledge, 2017.

[288] Armstrong Foundjem et al. A grounded theory of open sourcecross-communities feedback mechanisms.

[289] Suzanne Nobrega et al. Feasibility of virtual focus groups in program impact evaluation. *International Journal of Qualitative Methods*, 20, 2021.

[290] Catarina Gralha et al. The evolution of requirements practices in software startups. In *2018 IEEE/ACM 40th ICSE*, pages 823–833. IEEE, 2018.

[291] Anselm Strauss and Juliet M. Corbin. *Basics of qualitative research: Grounded theory procedures and techniques.* Basics of qualitative research: Grounded theory procedures and techniques. 1990. Pages: 270.

[292] Martin Bulmer. Concepts in the analysis of qualitative data. *The Sociological Review*, 27(4):651–677, 1979.

[293] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 78–88, USA, 2009. IEEE Computer Society.

[294] Gerardo Canfora, Luigi Cerulo, Marta Cimitile, and Massimiliano Di Penta. How changes affect software entropy: an empirical study. *Empirical Software Engineering*, 19(1):1–38, 2014.

[295] Jianxiong Guo and Weili Wu. Influence maximization: Seeding based on community structure. *ACM Trans. Knowl. Discov. Data*, 14(6), sep 2020.

[296] Aric A. Hagberg et al. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference*, pages 11–15, Pasadena, CA USA, 2008.

[297] Ming Cheung et al. Prediction of virality timing using cascades in social media. *ACM Trans. Multimedia Comput. Commun. Appl.*, 13(1), dec 2016.

[298] Suman Banerjee et al. Combim: A community-based solution approach for the budgeted influence maximization problem. *Expert Systems with Applications*, 125:1–13, 2019.

[299] Jungseul Ok, Youngmi Jin, Jinwoo Shin, and Yung Yi. On maximizing diffusion speed in social networks: Impact of random seeding and clustering. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, page 301–313, New York, NY, USA, 2014. Association for Computing Machinery.

[300] Dmitri Goldenberg et al. Timing matters: Influence maximization in social networks through scheduled seeding. *IEEE Transactions on Computational Social Systems*, 5(3):621–638, 2018.

[301] David Kempe et al. Maximizing the spread of influence through a social network. In *Ninth ACM SIGKDD international conference on KDD*, pages 137–146, 2003.

[302] Bin Liu et al. Recognition and vulnerability analysis of key nodes in power grid based on complex network centrality. *TCAS-II*, 65(3):346–350, 2018.

[303] Ahmet Erdem Sariyuce, Kamer Kaya, Erik Saule, and Umit V. Çatalyiirek. Incremental algorithms for closeness centrality. In *2013 IEEE International Conference on Big Data*, pages 487–492, 2013.

[304] Anand Bihari and Manoj Kumar Pandia. Eigenvector centrality and its application in research professionals' relationship network. In *2015 IEEE — ABLAZE*, pages 510–514, 2015.

[305] Jun Ai et al. Decentralized collaborative filtering algorithms based on complex network modeling and degree centrality. *IEEE Access*, 8:151242–151249, 2020.

[306] Junlong Zhang and Yu Luo. Degree centrality, betweenness centrality, and closeness centrality in social network. In *IEEE MSAM2017*, volume 132, pages 300–303, 2017.

[307] Kai Sheng and Zhi Zhang. Research on the influence maximization based on community detection. In *2018 13th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, pages 2797–2801, 2018.

[308] R Jaya Krishna et al. Analysis of community detection algorithms. In *2018 Second IEEE – ICICCT*, pages 669–674, 2018.

[309] Sayan Ghosh et al. Distributed louvain algorithm for graph community detection. In *2018 IEEE – IPDPS*, pages 885–895, 2018.

[310] Guilherme Avelino et al. On the abandonment and survival of open source projects: An empirical investigation. In *2019 ACM/IEEE ESEM*, pages 1–12, 2019.

[311] Igor Steinmacher et al. Almost there: A study on quasi-contributors in open-source software projects. In *2018 IEEE/ACM 40th ICSE*, pages 256–266, 2018.

Appendices

.1 Questions for guiding the OpenStack interviews on co-authoring

.1.1 Demographics

This first set of questions allow us to determine the profile and role of each interviewee within OpenStack.

1. How and why did you start to get involved in OpenStack?
2. What is your role in OpenStack? (And how did your role evolve over time?)
3. Which and how many OpenStack projects have you been involved in?
4. For how long have you been involved in OpenStack (and in these specific projects)?

.1.2 Generic questions

[These questions will be asked to each interviewee, regardless of his or her profile.]

1. Which mechanisms are you aware of (or have personal experience with) for making joint contributions to OpenStack projects with other persons?
   [If the question is too unclear, provide concrete examples to the interviewee, e.g., internally visible branch, externally visible branch, emails, slack, same commit, IRC ...]
2. Are you aware of (or familiar with) the possibility to co-author commits in OpenStack projects?
3. If yes:
   - Are co-authored commits common in the OpenStack projects you are involved in?

- What value, if any, does commit co-authoring bring to the OpenStack projects your are involved in?
- What are the drawbacks, if any, commit co-authoring brings to the ecosystem?
- Did/does the practice of co-authored commits improve onboarding experience? Go to C1 or D1

4. If no:

- For Foundation members, continue at question C2.
- For all other interviewees: end of interview.

.1.3 Questions for OpenStack foundation members

[These questions will only be asked to OpenStack foundation members.]

1. If the interviewee is aware of the possibility to co-author commits:

   (a) In general, why do co-authored commits happen in OpenStack?

   (b) Does OpenStack actively encourage co-authored commits? Why (not)?

   (c) Are OpenStack ecosystem members satisfied by the way in which co-authored commits are supported process-wise or tool-wise? Do you see room for improvement? How?

2. How is OpenStack (or the specific projects you are or have been involved in) dealing with contributor onboarding, i.e., trying to attract and retain new contributors? Which techniques and/or processes are used to support this?

3. How is OpenStack trying to reduce contributor turnover, and more specifically how is it trying to avoid key contributors from abandoning?

4. Apart from the above issues, according to your personal experience, what are other social, technical or organizational health problems OpenStack is confronted with, including its community and its open source code base?

5. How are these problems being addressed? For those problems that are not addressed yet, how should they be addressed?

.1.4 Questions for OpenStack practitioners

[These questions will only be asked to software developers involved in OpenStack projects.]

1. Have you yourself been involved in co-authoring commits? For which projects (within and beyond OpenStack)?

2. If yes:

   (a) How frequently have you co-authored commits?

   (b) What were the reasons for, and goals of, co-authoring the commits you were involved in (as opposed to individually authoring them)?

   (c) Are you aware of other reasons/goals of co-authored commits?

   (d) How much experience did you have in OpenStack when you started co-authoring commits?

   (e) What were the characteristics of the persons you co-authored with (juniors, seniors, experts in specific topics, . . .)?

   (f) What process do you use for co-authoring commits with other contributors (communication, division of tasks, ...)?

   (g) Who becomes the “principal author” (i.e., the author recorded in Git)?

(h) Do you explicitly mark your co-authored commits using "co-authored trailers" in commit messages? Why (not)?

(i) Are you satisfied by the way in which co-authored commits are supported by OpenStack, both process-wise or tool-wise? Do you see room for improvement? How?

3. If not at all:

(a) Was it an explicit decision not to get involved in co-authoring?

(b) If yes, why?

(c) If not, do you see:

i. any value that commit co-authoring could bring to you?

ii. any drawbacks that commit co-authoring could bring to you?

.2 Extended summary of the straussian GT data analysis

Figure 1 Extended summary of the Straussian GT data analysis, showing (A) **Open coding**.

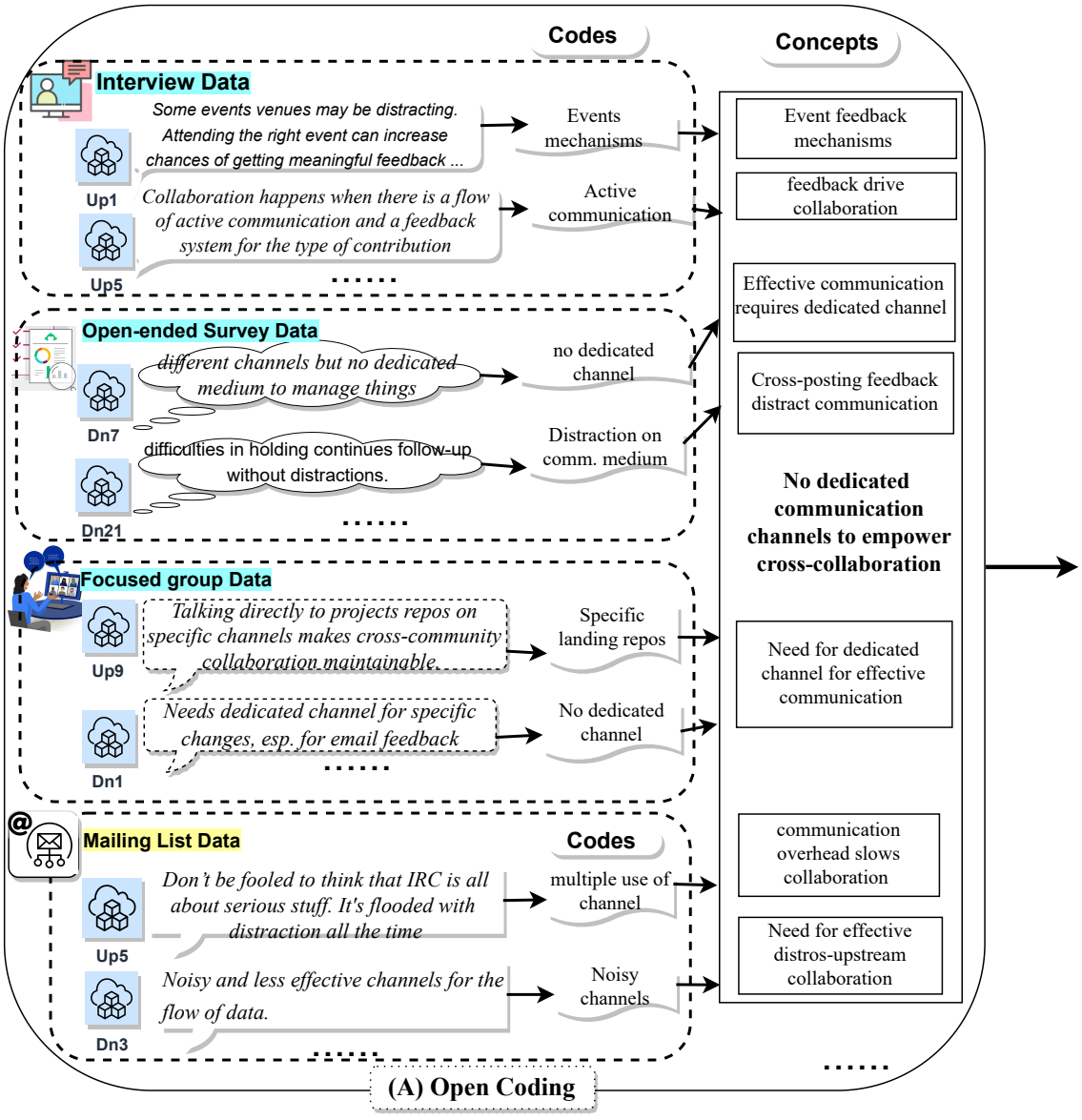


Figure 2 Extended summary of the Straussian GT data analysis, showing (B) **Axial coding**.

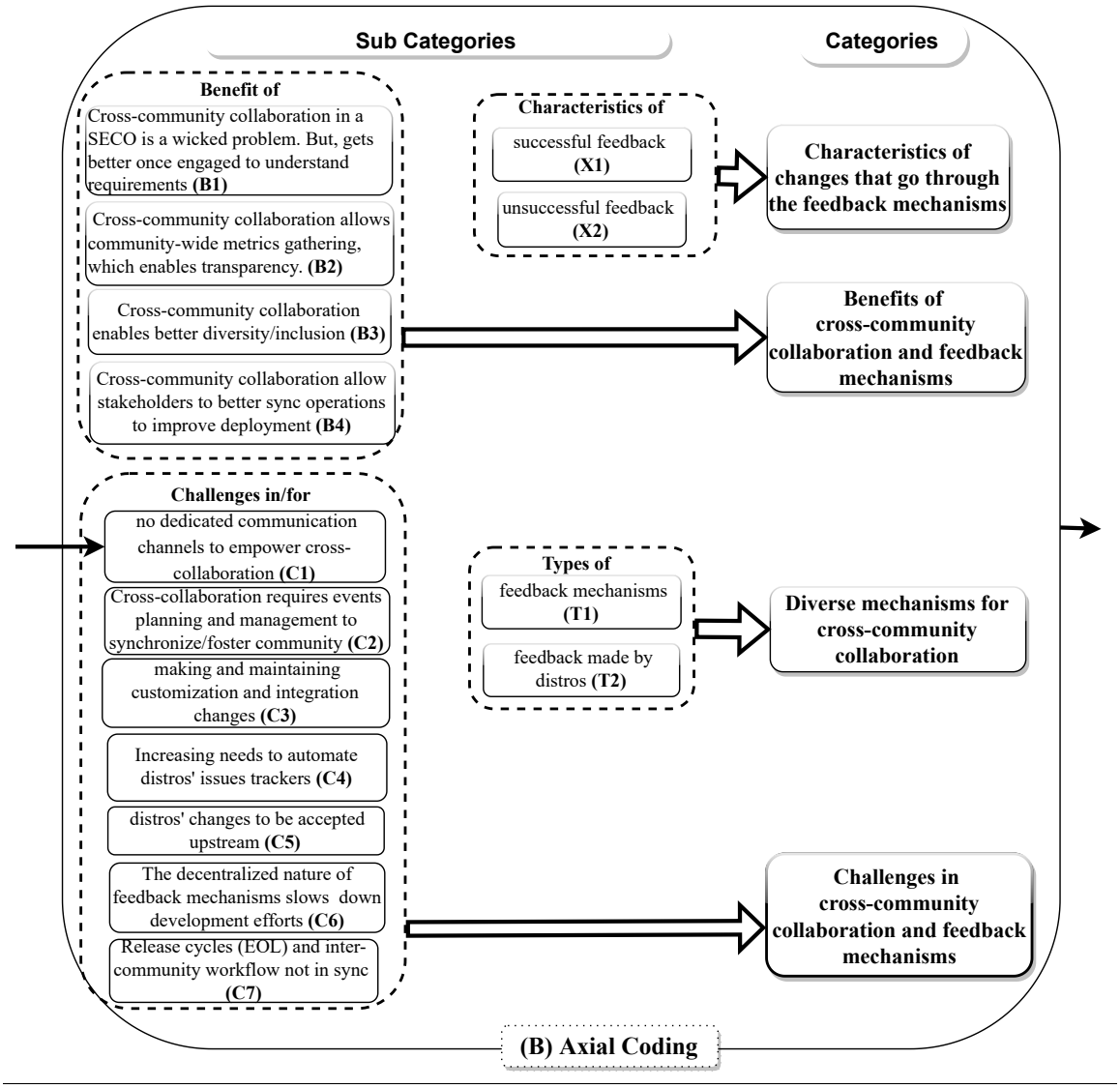


Figure 3 Extended summary of the Straussian GT data analysis, showing (C) **Selective coding**

