Noname manuscript No.
(will be inserted by the editor)

A mixed-methods analysis of micro-collaborative coding practices in OpenStack

Armstrong Foundjem, Member, IEEE, ·
Eleni Constantinou ·
Tom Mens, Senior Member, IEEE ·
Bram Adams, Member, IEEE

the date of receipt and acceptance should be inserted later

Abstract Technical collaboration between multiple contributors is a natural phenomenon in distributed open source software development projects. Macro-collaboration, where each code commit is attributed to a single collaborator, has been extensively studied in the research literature. This is much less the case for so-called micro-collaboration practices, in which multiple authors contribute to the same commit. To support such practices, GitLab and GitHub started supporting social coding mechanisms such as the "Co-Authored-By:" trailers in commit messages, which, in turn, enable to empirically study such micro-collaboration. In order to understand the mechanisms, benefits and limitations of micro-collaboration, this article provides an exemplar case study of collaboration practices in the OpenStack ecosystem. Following a mixed-method research approach we provide qualitative evidence through a thematic and content analysis of semi-structured interviews with 16 OpenStack contributors. We contrast their perception with quantitative evidence gained by statistical analysis of the git commit histories ($\sim$1M commits) and Gerrit code review histories ($\sim$631K change sets and $\sim$2M patch sets) of 1,804 OpenStack project repositories over a 9-year period. Our findings provide novel empirical insights to practitioners to promote micro-collaborative coding practices, and to academics to conduct further research towards understanding and automating the micro-collaboration process.

Keywords collaborative software development · open source software · OpenStack · social coding · code reviews · mixed method research

Armstrong Foundjem
School of Computing, Queen's University, Kingston, Canada
E-mail: a.foundjem@queensu.ca

Eleni Constantinou
Department of Mathematics and Computer Science, Eindhoven University of Technology, Netherlands
E-mail: e.constantinou@tue.nl

Tom Mens
Software Engineering Lab, University of Mons, Belgium
E-mail: tom.mens@umons.ac.be

Bram Adams
School of Computing, Queen's University, Kingston, Canada
E-mail: bram.adams@queensu.ca

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 paper 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 [1, 2, 3, 4, 5]. 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 [6, 7, 8, 9, 10, 11, 12] and improving onboarding [13]. 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.

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-

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

[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

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 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 paper 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 [14].

This paper studies as an exemplar case the OpenStack ecosystem[8], a popular open-source platform for cyloud computing governed by the Open Infrastructure Foundation (OIF). To do so, we follow a mixed-methods research methodology [15] 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:

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 paper'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.

[6] https://wiki.openstack.org/wiki/GitCommitMessages
[7] https://www.infoq.com/news/2018/01/github-multiple-author-commit/
[8] https://www.openstack.org/

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 [16][9].

2 Background

The goal of this paper 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 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.

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 [13], 63 of which are so-called core projects [17]. Zhang et al. [18, 19] 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 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 [20], such as the official core projects.

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

[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/`

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 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 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 [21, 22] 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* [23] 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].

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 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

[13] https://launchpad.net/openstack

[14] https://storyboard.openstack.org

[15] https://docs.openstack.org/infra/system-config/zuul.html

[16] https://wiki.openstack.org/wiki/GitCommitMessages

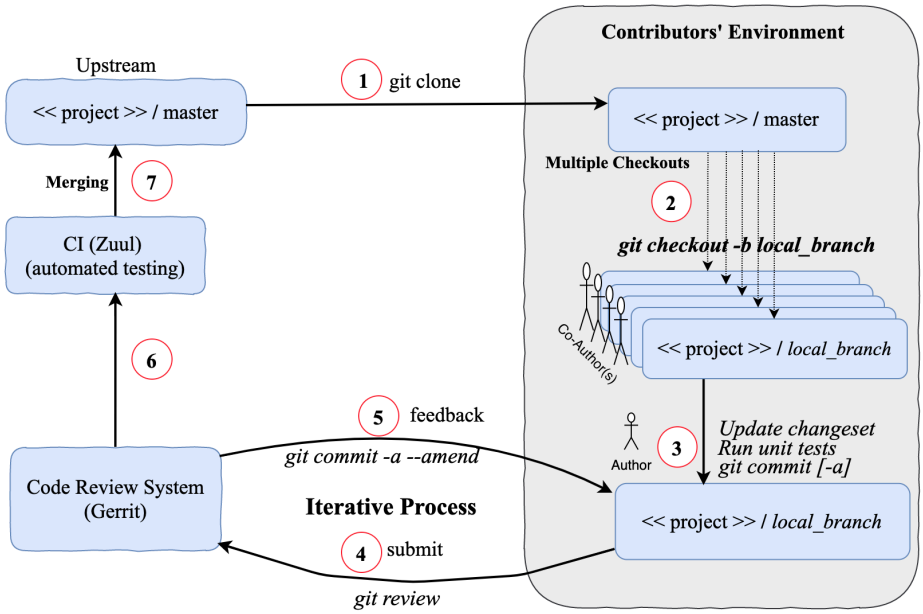
[17] https://www.openstack.org/videos/summits/berlin-2018/community-contributor-recognition-and-how-to-get-started

[18] https://superuser.openstack.org/articles/open-infrastructure-community-contributor-awards-denver-summit-edition/

[19] https://wiki.openstack.org/wiki/AUCRecognition

[20] https://superuser.openstack.org/articles/auc-community/

[21] https://wiki.openstack.org/wiki/Community/AmbassadorProgram



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

Fig. 1: Gerrit code contribution process in OpenStack (adapted from *).

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 paper will only focus on micro-collaboration to the code base, thus, we will only investigate attribution and co-authorship in the code base.

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 [24] with OpenStack practitioners during the Open Infrastructure Summit (OIS) in Berlin, Germany from 13 to 15 November 2018.

3.1 Selection and Demographics of Participants

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

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 first author knew personally. For broader coverage and response we drafted a recruitment screener [25] 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 on-site through snowball sampling [26] 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 [27, 28], similar to how previous studies [29, 30] 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 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.

3.2 Interview Recording and Transcription

All interviews were semi-structured and performed by the first author of this article 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 A. The interview guide was designed to include three main series of sections:

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.

Table 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

After the summit, the recorded interviews were distributed equally over the last three authors of this paper 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.

3.3 Interview Coding and Reviewing

Our qualitative methods consisted of: (1) a thematic analysis [31, 32, 33] from the transcribed data to uncover themes such as the expected benefits, challenges and communication mechanisms at OpenStack; and (2) a content analysis [34, 35] 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* [36] conducted by one author who used two transcripts (corresponding to $10-15\%$ of the total set of transcripts [37]). 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* [38].

Next, all authors were involved in a *deductive coding* process [36] 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. The purpose was to

Table 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 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.

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 [39] to calculate inter-rater reliability (IRR) [40]. We used Landis' agreement levels [41] to interpret the IRR as *poor*, *slight*, *fair*, *moderate*, *substantial* or *excellent*.

To iteratively improve the IRR, we performed three rounds of negotiated agreement [40]. 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 [16], 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 reports on these themes derived from the qualitative results together with their frequencies.

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 1. Numbered from **F1** to **F10**, the findings are summarised in Table 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 paper, 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 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 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.

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

Table 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

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 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 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

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

Table 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

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 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 ..."*

F4 Collaboration improves onboarding.

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 [13]. 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 enhances 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 improves 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 improves 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 5. Automation, tracking and promotion account for 71% of the total demands.

Table 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 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 [42], 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."*

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 2. We will empirically evaluate these qualitative findings on the basis of the quantitative study outlined in this section.

5.1 Extracting Micro-Collaborations

The qualitative analysis of Section 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.*

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 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:.

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

$\mathcal{P}_{init}$ = *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 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.

5.2 Data Cleaning

The initial datasets $\mathcal{C}_{init}$ and $\mathcal{P}_{init}$ presented in Section 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 [43] 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. [43]. These names and emails where excluded when comparing terms during identity merging.

Then, the first author acquired a partial ground-truth dataset containing the list of active contributor accounts, with names and username(s) of each contributor mapped 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. [44] 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

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

[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 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 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 [45, 46]. 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.*

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 6.*

[29] https://wiki.openstack.org/wiki/ThirdPartySystems

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

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 [8]. 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. [47]. 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 the collaborative or individual activity between the two platforms and are annotated as $\mathcal{C} \cap \mathcal{P}$ in the remainder of the paper.

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 [48, 49, 50]. Existing work on bug-inducing changes has focused especially on predicting whether a given change introduces a bug [51, 52], 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 [50, 53]). 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 [50, 51, 54]:

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

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

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 [55]. 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 [56, 57] 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.

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. [58] we carry out exploratory factor analysis (EFA) and confirmatory factor analysis (CFA) [59] 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 [60]. Full details are available in our replication package [16].

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

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

relationship, and (2) the data to understand these relationships should be interval-scaled or continuous data [61], even though Johnson et al. [62] argue that categorical data with more than five categories can still be considered for SEM models [62]. Our observed 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 [13, 63, 64, 65, 66, 67, 68, 69, 70]. We then filtered out all metrics not fit for SEM models, either (1) because they were not categorical with more than five categories [62], not interval-scaled or not continuous [61], (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 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. [58], 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 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 [71] 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 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 7 and 8. To determine the number of factors, we used the eigenvalue rule [59].

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

Table 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. [68]
juniorReviewers (q)	Number of junior developers (reviewers) with a +1 power [68]
reviewPriority (q)	Review priority of a patch set, the higher the more immediate attention required from experienced developers. [66] [65]
gating (q)	Number of times that CI/CD runs functional and integration testing jobs (successfully or not) [68]
verified (q)	Number of approval scores (+2) from automated testing, etc. [68]
workFlow (b)	Number of ±[1,2] scores for best practices including coding style, work-in-progress, etc. [70] [66]
comments (b)	Total number of comments on a change set. [64]
commenters (b)	Number of contributors posting comments in addition to the original author of the change set. [64]
churn (b)	Number of added and deleted lines of code. [67]
changedFiles (b)	Number of changed files in a change set. [65]
status (b)	Status of patch sets, which can either be "Merged" or "Abandoned". [67]
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. [64]
codeReviews (p^a)	Number of change sets under review. [63, 64]
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 [67] [69]. 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. [23] [64]
devProjects (p^o)	Number of projects a contributor commits code to. [13]

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 [59].

Table 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 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 7 and 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 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) [72, 73, 74] 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. 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 1 and 2 below:

$$\begin{aligned} socialInteraction &\sim automationEffort \\ reviewQuality &\sim automationEffort + socialInteraction \end{aligned} \quad (1)$$

$$reviewProductivity \sim socialInteraction \quad (2)$$

This allows us to see how *reviewQuality* is explained by *automationEffort* and *socialInteraction* (Equation 1), and how *reviewProductivity* is explained by *socialInteraction* (Equation 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 [58, 61, 73, 75, 75, 76, 77]. Cronbach's α [78] explains how closely related tested items are in a *group* [48], 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 [58] 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 [73, 76] ($0.07 \leq 0.08$), 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.

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. We would like to stress that, apart from the two SEM analyses [73, 77], 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.

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 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.

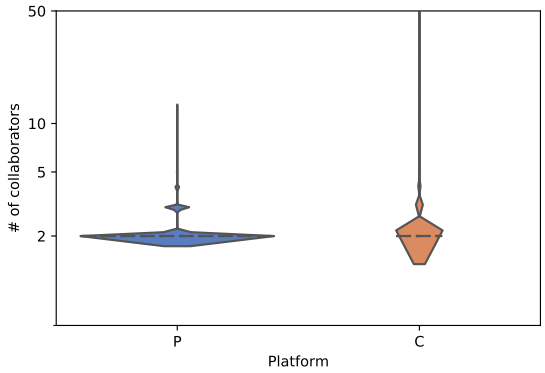


Fig. 2: Number of collaborators in collaborative git commits in $\mathcal{C}$; and in Gerrit change sets in $\mathcal{P}$.

> 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 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 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 [19]. In general, as observed by [19], collaborative coding doesn't happen frequently during OpenStack development cycles, but is used sparingly during complex features that involve multiple teams and contributors.

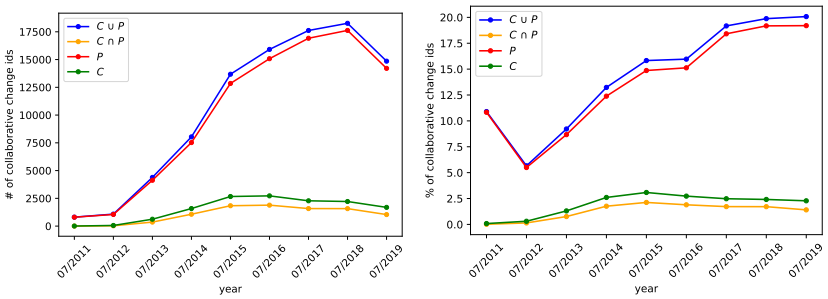
As micro-collaboration in OpenStack is observed in both git and Gerrit, we also investigate the overlap in micro-collaboration between both platforms. In other words, how many instances of collaboration are indicated both by git commit trailers and multiple Gerrit patch sets, and how many collaboration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

Table 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 $\mathcal{P}$,

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



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

Fig. 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).

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 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 3 years of development.

Figure 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 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 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 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.

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 2: **F4** *Collaboration improves onboarding*; **F6** *Collaboration improves soft-*

ware quality; **F7** *Collaboration improves productivity*; and **F8** *Collaboration enables recognition and accountability*.

F4 Collaboration improves onboarding

A recent study by Foundjem et al. [13] 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* [79, 80] 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 [80] 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 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 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 $p < 2.8e-5$.

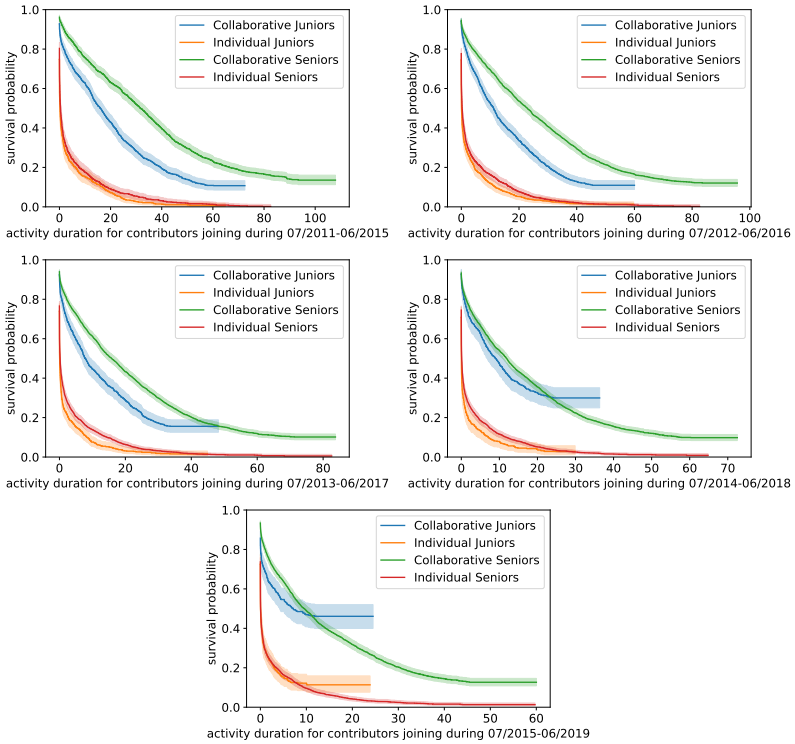


Fig. 4: Kaplan-Meier survival curves with 99% confidence interval (shaded areas) for sliding windows of 4-year periods, for junior/senior OpenStack contributors according to their collaboration status. The activity duration is measured in months.

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 improves 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 7. The latter analysis validates to what extent the review process is different for micro-collaborative and individual change sets.

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

Table 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 (14.5%)	13,678 (85.5%)	5,542.15 (34.6%)	10,455.85 (65.4%)
individual	315,690 (35.0%)	586,282 (65.0%)	312,467.85 (34.6%)	589,504.15 (65.4%)

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

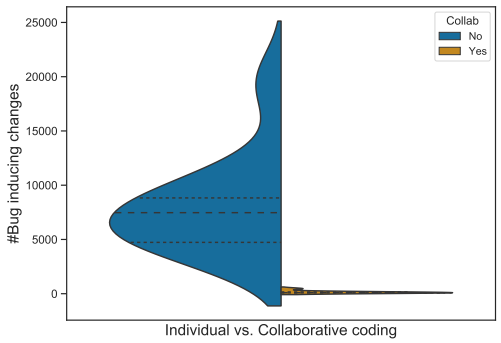


Fig. 5: Distribution of #BICs for individual (No) and collaborative (Yes) commits.

Using the approach of Section 5.4, we use a χ^2-test across the dimensions of micro-collaboration/individual coding style and BIC/non-BIC change set. Table 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 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 [81, 82] confirmed a *large* effect size for this difference ($\delta = 0.782$).[38]

Table 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.

[38] According to [82] 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 ($d \geq 0.474$).

Table 12: Top 10 OpenStack projects most affected by bug-inducing changes (in terms of their proportion and number).

Individual coding			Collaborative coding		
project	***proportion (%)***	***#BIC***	***project***	***proportion (%)***	***#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

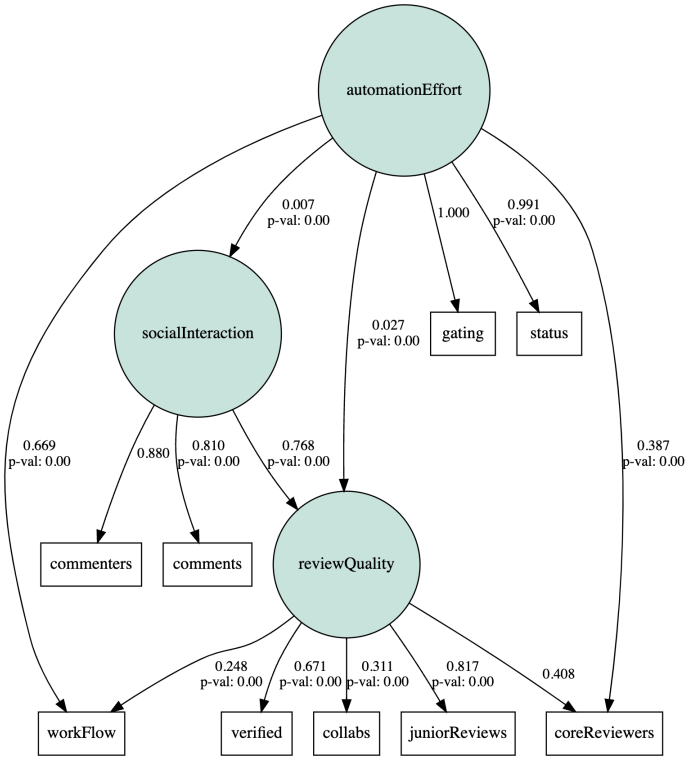


Fig. 6: Path diagram showing the structural analysis for the quality SEM model. Factors are shown as circles, observed variables as rectangles, and loadings as solid arrows.

> 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 1 in Section 5.5, we consider the model's path diagram (Figure 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 [83], which is the standardized (β) regression 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 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 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 work 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.

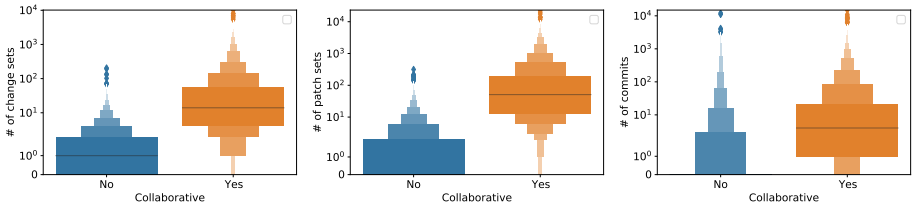
F7 Collaboration improves productivity

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 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 7.

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



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

Fig. 7: Overall productivity of collaborative vs non-collaborative contributors.

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 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.

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 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 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).

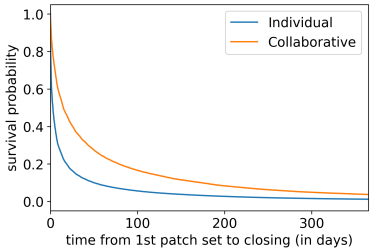


Fig. 8: Kaplan-Meier survival curves (with 99% confidence intervals) for the time between the first and the last patch set in a change set, focusing on the first year after initial patch submission.

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].

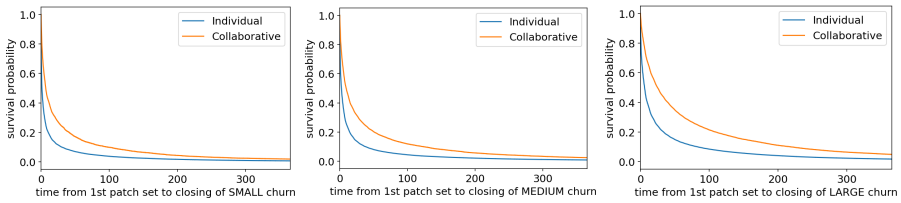
> Quantitative evidence contradicts contributor perception: collaborative change sets tend to require *more time* to finalise, and there is no evidence of a higher success rate.

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 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 ($p < 0.001$).

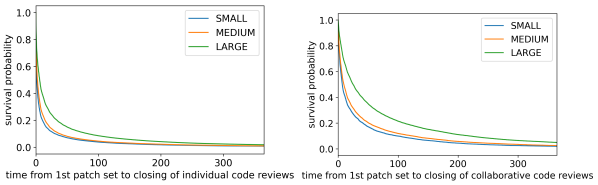
We performed a χ^2 test to verify if micro-collaborative patch sets are more likely to be *accepted* compared to individual patch sets. The null hypothesis, stating that collaboration strategy is independent from the final outcome of the change set (merged or abandoned), could not be rejected ($p = 0.078$). This suggests that the presence of collaboration is not related to the final outcome of the review (80% of individual and 87% of collaborative

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



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

Fig. 9: Time (in days) to finalize change sets for different code churn categories.



(a) Individual (b) Collaborative

Fig. 10: Time (in days) to finalize change sets for different churn sizes.

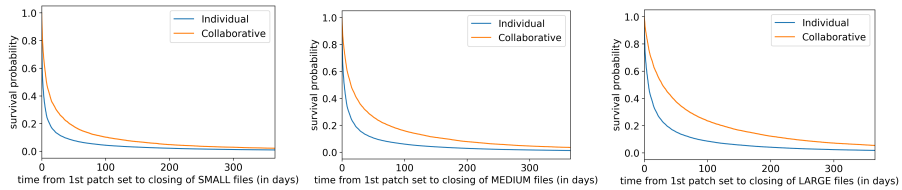
change sets are merged). Therefore, there is no quantitative evidence that collaboration is related to a higher success rate in merging change sets.

> 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 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.

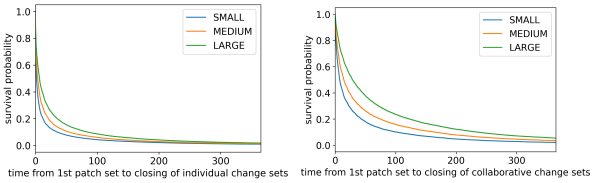
Figure 9 presents the survival curves corresponding to the time it took to finalize each change set, grouped by churn category and collaboration practice. The figure suggests that individual change sets take less time to finalize, regardless of the churn size category. This was statistically confirmed through log-rank tests ($p < 0.001$). Similarly, Figure 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 9) and collaboration categories (Figure 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 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*



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

Fig. 11: Time (in days) to finalize change sets for different file categories.



(a) Individual (b) Collaborative

Fig. 12: Time (in days) to finalize change sets for different file sizes.

category contained change sets touching 2 to 4 files, and the *large* category all change sets touching more than four files. Figure 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 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.

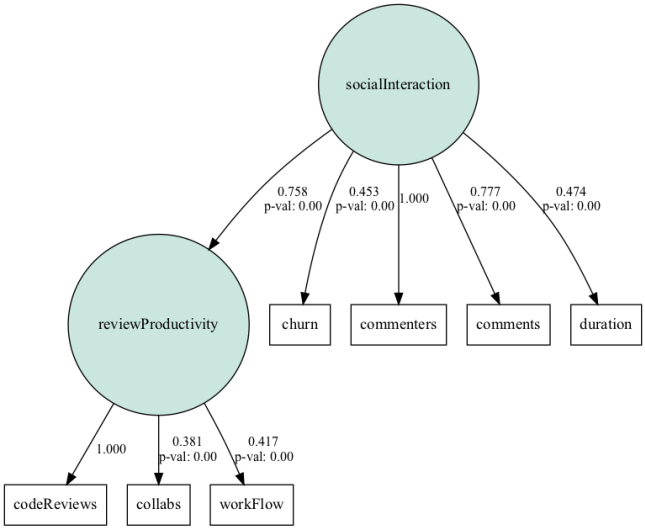


Fig. 13: Path diagram showing the structural analysis for the change-set level productivity SEM model. Factors are shown as circles, observed variables as rectangles, and loadings as solid arrows.

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

The SEM path analysis diagram in Figure 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 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 (∼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 OpenStack's repositories. We counted 497,442 such merged change sets, of which 55,794 were effectively micro-collaborative (see Section 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

[41] `https://superuser.openstack.org/articles/auc-community/`

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

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.

7 Related Work

7.1 Collaboration

Young et al. [84] 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. [19] 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, this work sheds light on the complexities of collaboration within such a large ecosystem. In contrast to studying collaboration at the company level, our study focuses on micro-collaboration between individuals in code reviews.

Whitehead [85] 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. [86] 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. [87] 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.

7.2 Developer Onboarding

Foundjem et al. [13] 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 paper 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. [88] 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. [89] 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.

7.3 Code Review and Quality

Henley et al. [90] 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 [22] 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*.

7.4 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. [91] 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 [92] 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. [93] 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 inves-

tigated in this paper 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.

8 Discussion

8.1 The Good

This paper 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.

8.2 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 [94], 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 [95]). 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 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.

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

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

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

8.3 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 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 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 [96]. 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.

8.4 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 first author 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 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"*.

9 Threats to Validity

We discuss the main threats that may affect the validity of our findings, following the structure recommended by Wohlin et al. [97].

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 [63]. 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 [98], 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 [99]. 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 [96].
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 over-estimations, the second author of the paper 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 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

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

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.

10 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 work 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.

Declarations

Funding: This research is partially supported by the F.R.S.-FNRS under Grant numbers T.0017.18, J.0151.20 and O.0157.18F-RG43 (Excellence of Science project SECO-Assist), as well as by the FRQ-F.R.S.-FNRS under Grant number 264544 (bilateral Québec-Wallonia project SECOHealth).

Conflict of interests/Competing interests: The authors have no relevant financial or non-financial interests to disclose.

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

Ethics Approval: The interviews performed during this study were subject to ethics certificate CER-1617-40, governed by the ethics board of Polytechnique Montreal.

References

1. C. de Souza Costa, J. J. Figueiredo, J. F. Pimentel, A. Sarma, and L. G. P. Murta, "Recommending participants for collaborative merge sessions," *Trans. Softw. Eng.*, pp. 1–1, 2019.
2. C. Costa, J. Figueiredo, L. Murta, and A. Sarma, "TIPMerge: recommending experts for integrating changes across branches," in *International Symposium on Foundations of Software Engineering*, 2016, pp. 523–534.
3. S. Krusche, M. Berisha, and B. Bruegge, "Teaching code review management using branch based workflows," in *International Conference on Software Engineering*. ACM, 2016, pp. 384–393.
4. M. T. Rahman, "Investigating modern release engineering practices," in *International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2015, pp. 607–608.
5. C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *International Symposium on Foundations of Software Engineering*. ACM SIGSOFT, 2012.
6. S. Datta, "How does developer interaction relate to software quality? An examination of product development data," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1153–1187, Jun. 2018.
7. T. Dingsøyr, N. B. Moe, T. E. Fægri, and E. A. Seim, "Exploring software development at the very large-scale: A revelatory case study and research agenda for agile method adaptation," *Empirical Software Engineering*, vol. 23, pp. 490–520, 2018.
8. E. Kalliamvakou, D. Damian, K. Blincoe, L. Singer, and D. M. German, "Open source-style collaborative development practices in commercial projects using GitHub," in *International Conference on Software Engineering*. IEEE, 2015, pp. 574–585.
9. S. Bick, K. Spohrer, R. Hoda, A. Scheerer, and A. Heinzl, "Coordination challenges in large-scale software development: A case study of planning misalignment in hybrid settings," *Trans. Softw. Eng.*, vol. 44, no. 10, pp. 932–950, 2018.
10. T. Neumayr, H.-C. Jetter, M. Augstein, J. Friedl, and T. Luger, "Domino: A descriptive framework for hybrid collaboration and coupling styles in partially distributed teams," *Human-Computer Interaction*, p. 24, Nov. 2018.
11. F. Mardi, K. Miller, and P. Balcerzak, "Novice - expert pair coaching: Teaching Python in a pandemic," in *Technical Symposium on Computer Science Education*. ACM, 2021, pp. 226–231.
12. H. Sharp and H. Robinson, "Collaboration and co-ordination in mature extreme programming teams," *International Journal of Human-Computer Studies*, vol. 66, no. 7, pp. 506–518, 2008.
13. A. Foundjem, E. E. Eghan, and B. Adams, "Onboarding vs. diversity, productivity and quality – empirical study of the OpenStack ecosystem," in *International Conference on Software Engineering*, 2021, pp. 1033–1045.
14. P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*, 1st ed. Wiley, 2012.
15. A. A. Al-Subaihin, F. Sarro, S. Black, L. Capra, and M. Harman, "App store effects on software engineering practices," *Trans. Softw. Eng.*, vol. 47, no. 2, pp. 300–319, 2021.

16. A. Foundjem, E. Constantinou, T. Mens, and B. Adams, "Replication package — v2.0.0)," https://doi.org/10.5281/zenodo.5759968, December 2021, online.
17. A. Foundjem and B. Adams, "Release synchronization in software ecosystems," *Empirical Software Engineering*, vol. 26, no. 34, 2021.
18. Y. Zhang, M. Zhou, A. Mockus, and Z. Jin, "Companies' participation in oss development–an empirical study of openstack," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2242–2259, 2021.
19. Y. Zhang, M. Zhou, K.-J. Stol, J. Wu, and Z. Jin, "How do companies collaborate in open source ecosystems?" in *International Conference on Software Engineering*. ACM, 2020, pp. 1196–1208.
20. S. Zhou, B. Vasilescu, and C. Kästner, "How has forking changed in the last 20 years? A study of hard forks on GitHub," in *International Conference on Software Engineering*. ACM, 2020, pp. 445–456.
21. M. Mukadam, C. Bird, and P. C. Rigby, "Gerrit software code review data from Android," in *International Working Conference on Mining Software Repositories*, 2013, pp. 45–48.
22. S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality," in *Working Conference on Mining Software Repositories*. ACM, 2014, pp. 192–201.
23. X. Xia, D. Lo, X. Wang, and X. Yang, "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)*, 2015, pp. 261–270.
24. C. Bird, "Interviews," in *Perspectives on Data Science for Software Engineering*. Morgan Kaufmann, 2016.
25. C. D. Egelman, E. Murphy-Hill, E. Kammer, M. M. Hodges, C. Green, C. Jaspan, and J. Lin, "Predicting developers' negative feelings about code review," in *International Conference on Software Engineering*. IEEE, 2020, pp. 174–185.
26. N. Salleh, R. Hoda, M. T. Su, T. Kanij, and J. Grundy, "Recruitment, engagement and feedback in empirical software engineering studies in industrial contexts," *Information and Software Technology*, vol. 98, pp. 161–172, 2018.
27. G. Guest, A. Bunce, and L. Johnson, "How many interviews are enough? an experiment with data saturation and variability," *Field methods*, vol. 18, no. 1, pp. 59–82, 2006.
28. P. I. Fusch and L. R. Ness, "Are we there yet? data saturation in qualitative research," *The qualitative report*, vol. 20, no. 9, p. 1408, 2015.
29. M. Kim, T. Zimmermann, R. DeLine, and A. Begel, "The emerging role of data scientists on software development teams," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 96–107.
30. A. N. Meyer, E. T. Barr, C. Bird, and T. Zimmermann, "Today was a good day: The daily life of software developers," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 863–880, 2019.
31. N. Terzimehić, R. Häuslschmid, H. Hussmann, and m. schraefel, "A review & analysis of mindfulness research in HCI," in *ICHF in Computing Systems*. ACM, 2019, pp. 1–13.
32. P. Lenberg, L. G. W. Tengberg, and R. Feldt, "An initial analysis of software engineers' attitudes towards organizational change," *Empirical Software Engineering*, vol. 22, no. 4, pp. 2179–2205, 2017.
33. D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *International Symposium on Empirical Software Engineering and Mea-*

surement, 2011, pp. 275–284.
34. D. Arya, W. Wang, J. L. C. Guo, and J. Cheng, "Analysis and detection of information types of open source software issue discussions," in *International Conference on Software Engineering*, 2019, pp. 454–464.
35. J. Himmelsbach, S. Schwarz, C. Gerdenitsch, B. Wais-Zechmann, J. Bobeth, and M. Tscheligi, "Do we care about diversity in human computer interaction," in *International Conference on Human Factors in Computing Systems*. ACM, 2019, pp. 1–16.
36. H. R. Bernard, A. Wutich, and G. W. Ryan, *Analyzing qualitative data: Systematic approaches*. SAGE publications, 2016.
37. M. W. DiStaso and D. S. Bortree, "Multi-method analysis of transparency in social media practices: Survey, interviews and content analysis," *Public Relations Review*, vol. 38, no. 3, pp. 511–514, 2012.
38. J. Saldaña, *The coding manual for qualitative researchers*. SAGE publications, 2015.
39. M. S. Islam, W. Khreich, and A. Hamou-Lhadj, "Anomaly detection techniques based on kappa-pruned ensembles," *IEEE Transactions on Reliability*, vol. 67, no. 1, pp. 212–229, 2018.
40. J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews," *Sociological Methods & Research*, vol. 42, no. 3, pp. 294–320, 2013.
41. J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, 1977.
42. O. Mlouki, F. Khomh, and G. Antoniol, "On the detection of licenses violations in the android ecosystem," in *International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016, pp. 382–392.
43. M. Goeminne and T. Mens, "A comparison of identity merge algorithms for software repositories," *Science of Computer Programming*, vol. 78, pp. 971–986, 2013.
44. C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *International Working Conference on Mining Software Repositories*. ACM, 2006, pp. 137–143.
45. M. Golzadeh, A. Decan, E. Constantinou, and T. Mens, "Identifying bot activity in github pull request and issue comments," in *2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE)*, 2021, pp. 21–25.
46. N. Cassee, C. Kitsanelis, E. Constantinou, and A. Serebrenik, "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)*, 2021, pp. 654–658.
47. D. M. German, B. Adams, and A. E. Hassan, "Continuously mining distributed version control systems: An empirical study of how Linux uses git," *Empirical Software Engineering*, vol. 21, no. 1, pp. 260–299, Feb. 2016.
48. Y. Fan, X. Xia, D. Alencar da Costa, D. Lo, A. E. Hassan, and S. Li, "The impact of changes mislabeled by SZZ on just-in-time defect prediction," *Trans. Softw. Eng.*, p. 26, 2019.
49. E. C. Neto, D. A. d. Costa, and U. Kulesza, "Revisiting and improving SZZ implementations," in *International Symposium on Empirical Software Engineering and Measurement*, 2019, pp. 1–12.
50. M. Borg, O. Svensson, K. Berg, and D. Hansson, "SZZ unleashed: An open implementation of the SZZ algorithm," in *MaLTeSQuE*. ACM, 2019, pp. 7–12.
51. J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005.

52. L. An, F. Khomh, and Y.-G. Guéhéneuc, "An empirical study of crash-inducing commits in Mozilla Firefox," *Software Quality Journal*, vol. 26, pp. 553–584, 2018.
53. G. Rodriguez, G. Robles, and J. Gonzalez-Barahona, "Reproducibility and credibility in empirical software engineering," *Information and Software Technology*, vol. 99, pp. 164–176, 2018.
54. M. Wen, R. Wu, Y. Liu, Y. Tian, X. Xie, S.-C. Cheung, and Z. Su, "Exploring and exploiting the correlations between bug-inducing and bug-fixing commits," in *Joint Meeting on ESEC and FSE*. ACM, 2019, pp. 326–337.
55. D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Joint Meeting on ESEC and FSE*. ACM, 2018, p. 3.
56. M. L. McHugh, "The chi-square test of independence," *Biochemia medica: Biochemia medica*, vol. 23, no. 2, pp. 143–149, 2013.
57. A. Satorra and P. M. Bentler, "A scaled difference chi-square test statistic for moment structure analysis," *Psychometrika*, vol. 66, no. 4, pp. 507–514, 2001.
58. J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring and modeling programming experience," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1299–1334, 2014.
59. H. Treiblmaier and P. Filzmoser, "Exploratory factor analysis revisited: How robust methods support the detection of hidden multivariate data structures in is research," *Information & Management*, vol. 47, no. 4, pp. 197–207, 2010.
60. G. Meshcheryakov, A. A. Igolkina, and M. G. Samsonova, "semopy 2: A structural equation modeling package with random effects in python," *arXiv preprint arXiv:2106.01140*, 2021.
61. A. Gopal, T. Mukhopadhyay, and M. S. Krishnan, "The impact of institutional forces on software metrics programs," *Trans. Softw. Eng.*, vol. 31, no. 8, pp. 679–694, 2005.
62. D. R. Johnson and J. C. Creech, "Ordinal measures in multiple indicator models: A simulation study of categorization error," *American Sociological Review*, pp. 398–407, 1983.
63. N. Forsgren, M.-A. Storey, C. Maddila, T. Zimmermann, B. Houck, and J. Butler, "The SPACE of developer productivity: There's more to it than you think." *Queue*, vol. 19, no. 1, pp. 20–48, Feb. 2021.
64. D. Izquierdo-Cortazar, N. Sekitoleko, J. M. Gonzalez-Barahona, and L. Kurth, "Using metrics to track code review performance," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE'17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 214–223. [Online]. Available: https://doi.org/10.1145/3084226.3084247
65. O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality: How developers see it," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1028–1038. [Online]. Available: https://doi.org/10.1145/2884781.2884840
66. V. Kovalenko and A. Bacchelli, "Code review for newcomers: Is it different?" in *Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering*, ser. CHASE '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 29–32. [Online]. Available: https://doi.org/10.1145/3195836.3195842
67. S. McIntosh, Y. Kamei, B. Adams, and A. 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*, ser. MSR 2014. New York, NY,

USA: Association for Computing Machinery, 2014, pp. 192–201. [Online]. Available: https://doi.org/10.1145/2597073.2597076
68. ——, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.
69. E. Oliveira, E. Fernandes, I. Steinmacher, M. Cristo, T. Conte, and A. Garcia, "Code and commit metrics of developer productivity: a study on team leaders perceptions," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2519–2549, 2020.
70. P. Rigby, B. Cleary, F. Painchaud, M.-A. Storey, and D. German, "Contemporary peer review in action: Lessons from open source development," *IEEE Software*, vol. 29, no. 6, pp. 56–61, 2012.
71. C. Tong, S. K.-S. Wong, and K. P.-H. Lui, "The influences of service personalization, customer satisfaction and switching costs on e-loyalty," *International Journal of Economics and Finance*, vol. 4, no. 3, pp. 105–114, 2012.
72. L. Klem, "Structural equation modeling." 2000.
73. R. P. Bagozzi and Y. Yi, "Specification, evaluation, and interpretation of structural equation models," *Journal of the academy of marketing science*, vol. 40, no. 1, pp. 8–34, 2012.
74. A. A. Igolkina and G. Meshcheryakov, "semopy: A python package for structural equation modeling," *Structural Equation Modeling: A Multidisciplinary Journal*, vol. 27, no. 6, pp. 952–963, 2020. [Online]. Available: https://doi.org/10.1080/10705511.2019.1704289
75. H. Kang and J.-W. Ahn, "Model setting and interpretation of results in research using structural equation modeling: A checklist with guiding questions for reporting," *Asian Nursing Research*, vol. 15, no. 3, pp. 157–162, 2021.
76. R. Ghaiumy Anaraky, Y. Li, and B. Knijnenburg, "Difficulties of measuring culture in privacy studies," *Proc. ACM Hum.-Comput. Interact.*, vol. 5, no. CSCW2, oct 2021. [Online]. Available: https://doi.org/10.1145/3479522
77. T. N. Beran and C. Violato, "Structural equation modeling in medical research: a primer," *BMC research notes*, vol. 3, no. 1, pp. 1–10, 2010.
78. R. Vallat, "Pingouin: statistics in python," *Journal of Open Source Software*, vol. 3, no. 31, p. 1026, 2018. [Online]. Available: https://doi.org/10.21105/joss.01026
79. J. T. Rich, J. G. Neely, R. C. Paniello, C. C. J. Voelker, B. Nussenbaum, and E. W. Wang, "A practical guide to understanding kaplan-meier curves," *Otolaryngology–head and neck surgery*, vol. 143, no. 3, pp. 331–6, 2010.
80. M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1063–1088, 2017.
81. N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," *Psychological Bulletin*, vol. 114, no. 3, pp. 4994–509, November 1993.
82. 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*, pp. 1–30, 2004.
83. G. D. Garson, *Path analysis*. Statistical Associates Publishing Asheboro, NC, 2013.
84. J.-G. Young, A. Casari, K. McLaughlin, M. Z. Trujillo, L. Hébert-Dufresne, and J. P. Bagrow, "Which contributions count? analysis of attribution in open source," in *International Working Conference on Mining Software Repositories*. IEEE, 2021.
85. J. Whitehead, "Collaboration in software engineering: A roadmap," in *Future of Software Engineering*, May 2007, pp. 214–225.

86. T. Mens, M. Cataldo, and D. Damian, "The social developer: The future of software development," *IEEE Software*, vol. 36, no. 1, p. 4, Jan 2019.
87. G. Avelino, L. Passos, A. Hora, and M. T. Valente, "Assessing code authorship: The case of the Linux kernel," in *Open Source Systems: Towards Robust Practices*. Springer, 2017, pp. 151–163.
88. C. Casalnuovo, B. Vasilescu, P. Devanbu, and V. Filkov, "Developer onboarding in GitHub: The role of prior social links and language experience," in *Joint Meeting on ESEC and FSE*. ACM, 2015, pp. 817–828.
89. I. Steinmacher, M. A. G. Silva, and M. A. Gerosa, "Barriers faced by newcomers to open source projects," in *Open Source Software: Mobile Open Source Technologies*. Springer, 2014.
90. A. Z. Henley, K. Muçlu, M. Christakis, S. D. Fleming, and C. Bird, "CFar: A tool to increase communication, productivity, and review quality in collaborative code reviews," in *CHI*. ACM, 2018, pp. 1–13.
91. L. Plonka, H. Sharp, J. van der Linden, and Y. Dittrich, "Knowledge transfer in pair programming: An in-depth analysis," *International Journal of Human-Computer Studies*, vol. 73, pp. 66–78, 2015.
92. K. M. Lui and K. C. Chan, "Pair programming productivity: Novice–novice vs. expert–expert," *International Journal of Human-Computer Studies*, vol. 64, no. 9, pp. 915–925, 2006.
93. A. H. 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.
94. F. P. Brooks Jr., *The mythical man-month*. MA, United States: Addison-Wesley Reading, 1974.
95. M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4, pp. 531–577, Aug. 2012.
96. C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "When and how to make breaking changes: Policies and practices in 18 open source software ecosystems," *Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, Jul. 2021.
97. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.
98. P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: A case study of the Apache server," in *International Conference on Software Engineering*. ACM, 2008, pp. 541–550.
99. P. Fusch and L. Ness, "Are we there yet? data saturation in qualitative research," *The Qualitative Report*, 2015.

A Questions for guiding the OpenStack interviews on co-authoring

A.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)?

A.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.

A.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?

A.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?