Scaling Open Source Communities: An Empirical Study of the Linux Kernel

Xin Tan
Department of Computer Science and Technology, Peking University
Key Laboratory of High Confidence Software Technologies, Ministry of Education
Beijing, China
tanxin16@pku.edu.com

Minghui Zhou*
Department of Computer Science and Technology, Peking University
Key Laboratory of High Confidence Software Technologies, Ministry of Education
Beijing, China
zhmh@pku.edu.com

Brian Fitzgerald
Lero—the Irish Software Research Centre, University of Limerick
Limerick, Ireland
bf@ul.ie

ABSTRACT

Large-scale open source communities, such as the Linux kernel, have gone through decades of development, substantially growing in scale and complexity. In the traditional workflow, maintainers serve as "gatekeepers" for the subsystems that they maintain. As the number of patches and authors significantly increases, maintainers come under considerable pressure, which may hinder the operation and even the sustainability of the community. A few subsystems have begun to use new workflows to address these issues. However, it is unclear to what extent these new workflows are successful, or how to apply them. Therefore, we conduct an empirical study on the multiple-committer model (MCM) that has provoked extensive discussion in the Linux kernel community. We explore the effect of the model on the i915 subsystem with respect to four dimensions: pressure, latency, complexity, and quality assurance. We find that after this model was adopted, the burden of the i915 maintainers was significantly reduced. Also, the model scales well to allow more committers. After analyzing the online documents and interviewing the maintainers of i915, we propose that overloaded subsystems which have trustworthy candidate committers are suitable for adopting the model. We further suggest that the success of the model is closely related to a series of measures for risk mitigation—sufficient precommit testing, strict review process, and the use of tools to simplify work and reduce errors. We employ a network analysis approach to locate candidate committers for the target subsystems and validate this approach and contextual success factors through email interviews with their maintainers. To the best of our knowledge, this is the first study focusing on how to scale open source communities. We expect that our study will help the rapidly growing Linux kernel and other similar communities to adapt to changes and remain sustainable.

*Corresponding author

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380920

KEYWORDS

Open source communities; Scalability; Sustainability; Multiple committers; Maintainer; Workload; Linux kernel

ACM Reference Format:
Xin Tan, Minghui Zhou, and Brian Fitzgerald. 2020. Scaling Open Source Communities: An Empirical Study of the Linux Kernel. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380920

1 INTRODUCTION

Over the past several decades, open source software (OSS) projects have grown rapidly in popularity, owing to their unique advantages, e.g., low cost and high quality [27, 60]. During this time, many prominent projects emerged, such as the Linux kernel and Apache HTTP Server [44]. At this stage, many of these projects are mature and not merely hobbyist projects [54], but rather represent critical computing infrastructure for society. Both the scale and complexity of these projects have undergone considerable changes, e.g., the Linux kernel version 4.19 has over 25.58 million lines of code contributed by 1,710 contributors representing 230 corporations [13], whereas version 0.01 in 1991 had only 10,239 lines of code.[1] These projects established their own workflows in the early stages. In general, a small core group does most of the work, and coordinates with a considerably larger group of peripheral participants [16, 35, 44]. However, in recent years, an increasing number of contributors in several OSS projects (e.g., Node.js,[2] Dpdk,[3] and Fastlane[4]) have raised concerns about whether the original workflows can handle the current volume of patches, which continues to increase. This is especially the case in the Linux kernel, where it has triggered a heated discussion.[5]

The code base of the Linux kernel is logically organized into many subsystems, each of which has a designated maintainer—a contributor who possesses the commit right to the subsystem repository that s/he maintains (some subsystems have more than one maintainer). Almost all of the code patches are selected and transferred by the corresponding maintainers, layer by layer, until they are merged into the mainline repository.[6] It is an undeniable fact

[1] https://en.wikipedia.org/wiki/Linux_kernel
[2] https://medium.com/the-node-js-collection/healthy-open-source-967fa8be7951
[3] http://mails.dpdk.org/archives/moving/2016-November/000060.html
[4] https://krausefx.com/blog/scaling-open-source-communities
[5] https://lists.linuxfoundation.org/pipermail/ksummit-discuss/2018-September/005313.html
[6] https://www.kernel.org/doc/html/latest/process/index.html

that this hierarchical review workflow is a key factor in maintaining the high quality of the kernel [9]. However, as the scale of the project increases, these maintainers need to deal with burdensome review work, such as 487 patches in a two-week period [34] and hundreds of emails a day [53]. Worse still, many subsystems have only one maintainer, which means that if those sole maintainers take a vacation, become ill, or simply become busy at their day job, there is no one who can replace them. Meanwhile, new maintainers are not easy to train—they need to not only review patches, but also to shoulder the responsibility of sending that work upstream and acting as a liaison for complaints arriving from other subsystems [56]. This series of problems have raised widespread concern from both the community [11] and researchers [61]. People are concerned that the overworked maintainers may not sustain the subsystems, and they may become a single point of failure, resulting in the disruption of the hierarchical production.

There is a demand to adapt the original workflow to the growing workload and to sustain the community. The multiple-committer model (MCM) that distributes the workload to a group of contributors by giving commit rights has been created to address this problem. It is endorsed by the subsystems currently applying it [55]. Although this model is not rare in other projects, decentralization by suddenly introducing an increased number of committers is a high-risk operation for any community that experiences rapid scale-up. It may result in unexpected problems and complaints from other maintainers [56]. Many subsystems are reluctant to adjust/change the current workflow, even though it is widely accepted as problematic. Adopting an invention usually appears as a continuous and slow process, because decisions are rendered difficult by the lack of unbiased information sources [18, 26]. A deep understanding of the factors affecting the choice of a new workflow is essential for its adoption and therefore for resolving the workload crisis.

In this study, we attempt to achieve this understanding, and to help projects that are undergoing workload problems to make decisions as to how to adjust/change the current workflow. Decisions are often the result of a comparison of the uncertain benefits of the new invention with the uncertain costs of adopting it [26]. Therefore, we start by seeking to understand the effectiveness of the new model, and then investigate the factors most relevant to the adoption of the new model.

- RQ1 To what extent does the MCM work (or how exactly does the MCM help to reduce maintainer's workload)?
- RQ2 How to apply the MCM?
  - 2.1 What factors are crucial for successful implementation/operation of the MCM?
  - 2.2 How should candidate committers be selected when maintainers adopt the MCM?

To answer these questions, we analyze the commit history of the Linux kernel, read related online documents, and conduct interviews with maintainers. We focus on maintainers' main workload, i.e., reviewing patches, and define four categories of metrics to characterize their performance: pressure, latency, complexity, and quality assurance. We utilize these metrics to quantify the effect of adopting the MCM on the i915 subsystem, which has been using it since 2015. We find that after the adoption, pressure on maintainers, review latency, and the complexity of the review relationship between maintainers and patch authors were significantly reduced. At the same time, the new model enforces a strict process to ensure patch quality. After analyzing the online documents and interviewing the maintainers, we recommend overloaded subsystems having the following properties to adopt the MCM: 1) there are suitable candidate committers; 2) maintainers and committers trust each other; and 3) they already have or are willing to introduce measures to mitigate risks. Based on these findings, we also discuss insights for scaling OSS projects. We expect that our results will help maintainers of the Linux kernel and other OSS projects become more sustainable. This work builds upon an earlier 2-page summary [52].

2 BACKGROUND

We examine the related work in Section 2.1. The traditional workflow and the new workflow of the Linux kernel are introduced in Section 2.2 and Section 2.3 respectively.

2.1 Related Work

As software evolves over time, it becomes increasingly large and complex [38]. During the process of software evolution, successful OSS projects attract many contributors [7, 62]. These contributors complete various tasks in a project, which helps to establish and sustain the project. However, activities in OSS projects tend to be skewed, i.e, a small number of people do most of the work [44]. This group of people often act as maintainers of projects and take responsibility for the direction of projects [20]. As the scale of projects expands, they face increasing pressure that may challenge their capacity. For example, Zhou et al. [61] found that the distribution of work among the Linux kernel maintainers followed the 80:20 rule for most modules, suggesting that a few maintainers may bear the brunt of the increased workload. These issues have caused OSS projects to increasingly worry about their sustainability [17, 61].

Different innovations have been introduced to deal with workload and improve productivity in OSS projects. Notable ones include the pull request model [23–25, 63] and the use of continuous integration (CI) for automated quality assurance [28–30]. For example, Zhu et al. [63] found the pull request model to be associated with reduced review time. Hilton found that 70% of the most popular projects on GitHub heavily use CI. In comparison to projects that do not use CI, these projects (i) release twice as often, (ii) accept pull requests faster, and (iii) developers are less worried about breaking the build [28]. Recent studies investigated bots seeking to automate repetitive tasks in the social collaboration platform such as GitHub [36, 58, 59]. For example, the Hound bot was invented to verify code style violations [58].

Although these inventions may accelerate maintainers' work, the effect is limited because a lot of manual intervention is still required in patch review, e.g., determining whether a patch is necessary, whether an implementation has shortcomings, or whether there are better ways to implement. This problem could be addressed by distributing the workload among members to achieve high bandwidth. As discovered, the successful evolution of OSS projects depends on the co-evolution of the team structure and the software structure [48]. The Linux kernel is organized as a structured hierarchy of modules, which allows flexible expansion of modules, together with their maintainers [3, 61]. This is the key to the success of Linux. However, as the pressure on maintainers continues to increase, it is clear that the current structure needs to evolve. While the community proposes a new multi-committer

model to adapt to the changes, it is unclear to what extent this model works, or how this model should be implemented. Therefore, in this study, we aim to investigate these two questions. The investigation can deliver insights into scaling of OSS communities, thus helping OSS ecosystems survive the constantly changing environment.

2.2 Traditional Workflow

The kernel development community is organized as a hierarchy, with contributors submitting patches to maintainers, who in turn commit those patches to a repository (with each maintainer being responsible for their own subsystem repository), and pushing them upstream to higher-level maintainers. This hierarchy logically resembles the directory hierarchy of the kernel source itself. The process requires all maintainers in the chain to take responsibility for the patches. The maintainers who introduce patches directly from authors take the main responsibility of reviewing patches. The higher-level maintainers who pull the patches from the lower-level maintainers are responsible for conducting integration tests—they usually do not review these patches. According to the online documents, reviewing patches requires considerable effort and represents the main workload for maintainers. This is further exacerbated by the fact that the number of patches continues to increase. Maintainers may worry about burnout when undertaking such heavy review work [11, 56, 61]. In this study, we focus on the main burden for maintainers, i.e., reviewing patches.

2.3 Multiple-Committer Model

To reduce the workload of maintainers, a few subsystems in the Linux kernel have attempted to apply new models. The MCM is the most discussed model in the community, and was applied by the i915 subsystem in October, 2015. As shown in Figure 1, in the traditional workflow, only the maintainers have the privilege to commit patches to the repositories they maintain, whereas in the MCM, many core contributors are given commit rights, allowing them to commit patches to the same repository as the maintainers. Because the committers[7] share a great deal of review work, the maintainers' workload is greatly reduced. In contrasted to the committers, the main job of the maintainers is to communicate externally, including coordinating with other subsystems, sending patches upstream, and being responsible for the faults in the subsystems.[8]

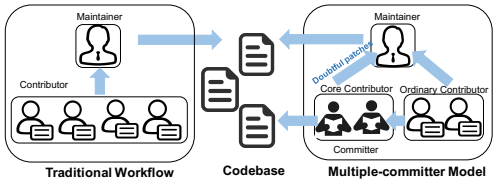

Figure 1: Two Types of Workflows of Subsystems

The MCM has been used by the i915 subsystem—which is also called INTEL DRM DRIVERS, supporting all integrated graphics chipsets with both Intel display and rendering blocks.[9] Its main files are under `drivers/gpu/drm/i915` of the kernel's code directory. The number of its maintainers fluctuates between 1–3 in the studied period. As shown in Figure 2, the numbers of commits, modified files, contributors, and committers appear to grow over time.

[7] In this study, we refer to the regular contributors (maintainers not included) who have commit right to the repositories as committers.

[8] https://kernel-recipes.org/en/2016/talks/maintainers-dont-scale/

[9] https://www.kernel.org/doc/html/v4.14/gpu/i915.html

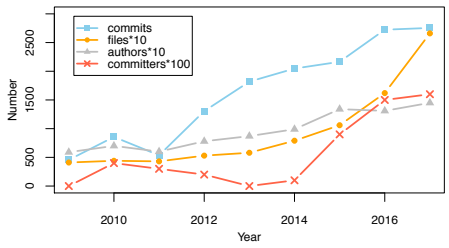

Figure 2: Growth of the i915 Subsystem

3 METHODOLOGY

3.1 Study Design

We applied a mixed-method approach [14] for this study. Figure 3 shows the overall design of the study, including the following steps.

Step 1. We started from collecting commit history data of the i915 subsystem and identifying the review workload of maintainers, as described in Section 3.2.

Step 2. We designed four categories of metrics to study the effect of the MCM, as described in Section 3.3.

Step 3. We reviewed the online documents related to the MCM and conducted interviews to understand which factors are important for the implementation of the MCM, as described in Section 3.4.

Step 4. As identified in the previous step, "committers" is a critical success factor of the MCM but difficult to identify, we borrowed the idea of collaboration network to propose an approach for selecting candidate committers, as described in Section 3.5.

Step 5. Finally, we conducted interviews within and outside (Node.js serves as an exemplar) the Linux kernel to validate the generality of the results, as described in Section 3.6.

3.2 Identification of the Maintainers' Workload

We cloned the mainline repository of the Linux kernel in June, 2018, and retrieved its commit history. Because the Linux kernel moved to Git in 2005, we ignored the pre-2005 history. We took steps to clean and standardize the raw data for further analysis. Each piece of data (i.e., a commit) records the hash of the patch,[10] the author of the patch, the person who committed the patch, the time at which the patch was committed, the description of the patch, and the affected files. For each affected file, we obtained the maintainer of the file and the repository for that file at that time. The Linux kernel contains a file named MAINTAINERS that records the name of each subsystem, the repository of the subsystem, the names of the individuals who maintain it, and the files associated with it. We obtained a version of this file from each month since April 2009, when the file was started.[11] For the i915 subsystem, we considered our observations to be only those commits that modified files belonging to the i915 subsystem (mainly in `drivers/gpu/drm/i915`).

The Linux kernel uses "signed-off-by" to track patches, which is a simple line at the end of the commit message that certifies the person who wrote it or has the right to pass it on. Through communicating with three maintainers, we found that generally, there are two types of contributors who can add this tag to a patch: the author of the patch, and the maintainer who accepts it from the author. The maintainers in the upper layers accept patches by a pull request from maintainers in the lower layers, and they

[10] In this paper, commit and patch are used interchangeably.

[11] For convenience, we obtained the first version on the 1st of each month.

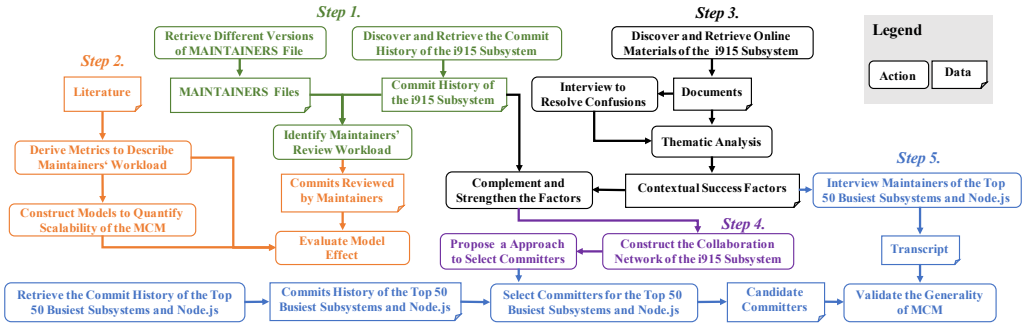

Figure 3: Overall Design of the Study

have no chance to modify the patches. Therefore, by analyzing this tag, we can identify the maintainers who initially introduce the patches to the hierarchical repositories of the Linux kernel. These maintainers undertake the main workload of patch review, as explained in Section 2.2. For each patch, if the individual who signed off the patch is not the author of the patch and her/his name is in the MAINTAINERS file, we consider that s/he actually reviewed the patch. This allowed us to obtain the review workload of each maintainer. If a contributor committed patches to the repositories, but her/his name does not appear in the corresponding version of the MAINTAINERS file, we consider her/him to be a committer. If a committer signed off a patch when s/he was not the author, s/he is considered to have reviewed this patch. This allowed us to obtain the review workload of each committer.

3.3 Evaluation of the Model Effects

Our sequential mixed method research approach [14] comprised two main stages. In the first stage, we conducted a literature review to derive metrics to describe maintainer workload.[12] Stage one concluded with the development of eight metrics across four categories. Based on these metrics, the second stage involved an evaluation of whether the metrics exhibited differences before and after adoption. Note that in the second stage, we did not seek to prove a *causal relationship*. We do show that although the number of patches has continued to increase, the burden of the maintainers is significantly reduced after adoption of MCM. This is also consistent with the perception of the kernel community.

In this section, we describe the metrics we used. The measurement of reviewers' workload is quite straightforward in the literature, typically the number of patches reviewed or in the pending queue [2, 33]. However, we cannot conduct a comprehensive evaluation of the MCM with just those metrics. Hence, we borrowed concepts and metrics that are often used to measure the performance of computer systems, i.e. throughput and latency [46, 57]. We established four high-level categories. All of the metrics are aimed at a particular subsystem and a particular timespan, i.e., a month. The related definitions and proposed metrics are as follows.

- $\boldsymbol{Mntr}$: the set of maintainers and committers;
- $\boldsymbol{Patch}$: the set of patches;
- $\boldsymbol{Patch_i}$: the set of patches signed off by contributor i;
- $\boldsymbol{Reviewer_p}$: the set of all the reviewers of patch p;
- $\boldsymbol{File_p}$: the set of files modified in patch p;
- $\boldsymbol{author_p}$: the author of patch p;
- $\boldsymbol{stime_p}$: the time that patch p was signed off by maintainer;
- $\boldsymbol{atime_p}$: the author time of patch p;

Pressure. Maintainers' pressure is measured by how much review work they have done. We define the workload of the busiest maintainer/committer as the bottleneck for processing patches in the subsystem. We also define two additional metrics that describe the distribution of the review workload among maintainers (and committers) that can reflect the risk of a single point of failure.

- M1: Maximum review workload: the number of the patches reviewed by the busiest maintainer/committer.

$$MAX\_RW = \max_{i \in Mntr} |Patch_i| \tag{1}$$

- M2: Intensity of review workload: the ratio of review workload done by the busiest maintainer/committer.

$$INTST\_RW = \frac{MAX\_RW}{|Patch|} \tag{2}$$

- M3: Entropy of review workload: the dispersion of review workload assignment, borrowed from information theory [50].

$$ENT\_RW = -\sum_{i \in Mntr} r_i \log_2 r_i, \ where \ r_i = \frac{|Patch_i|}{|Patch|}. \tag{3}$$

Latency. Latency is a measure of the time delay experienced by a system. In our study, we define latency as the length of time from a patch being submitted by the author to the time it is signed off by the first maintainer/committer.[13] We use this metric because a long latency implies that the maintainer may be overloaded, and the possibility of conflict is high. This metric is also used in practice by maintainers to judge whether they are overloaded [12].

- M4: Latency of review workload: the median review time of the patches.

$$LAT\_RW = \text{median}\ \{stime_p - atime_p \mid p \in Patch\} \tag{4}$$

Complexity. The more complex the work is, the more time and effort the maintainer must devote. The complexity of the code review is found to be related to the number of files [43], and the contact and communication with contributors [39]. Based on these two considerations, we define two metrics to characterize the complexity of maintainers' review work. We choose the maximum complexity of all of the maintainers (and committers) to reflect the bottleneck of the subsystem.

- M5: Complexity of review relationship: among all the maintainers (and committers), the maximum number of unique

[12] Note that because maintainers and committers are all reviewers, "maintainer" may refer to "committer" and "maintainers" may include "committers" in certain scenarios.

[13] A patch may go through several iterations (versions) before acceptance. Here we use the time when the last version of the patch is submitted.

authors reviewed by a certain maintainer/committer.

$$CPLX\_RR = \max_{i \in Mntr} \left|\{author_p \mid p \in Patch_i\}\right| \quad (5)$$

- M6: File complexity: among all the maintainers (and committers), the maximum number of unique files reviewed by a certain maintainer/committer.

$$CPLX\_F = \max_{i \in Mntr} \left|\{f \mid p \in Patch_i, f \in File_p\}\right| \quad (6)$$

Quality assurance. The purpose of patch review is to ensure that patches are of high quality. Instead of directly measuring quality, we define the following two metrics to measure the effort/strategy that is employed to deliver quality. This is justifiable given that patch quality is very high in this community.

- M7: The number of reviewers has been found to be important to ensure patch quality [41, 47, 63]. We use the average number of unique reviewers per patch (including the author) as the indicator of the effort/strategy for quality assurance. We retrieved all of the tags that indicated some sort of review, including "Signed-off-by," "Reviewed-by," and "Tested-by."[14]

$$QLTY\_ANR = \frac{\sum_{p \in Patch} |Reviewer_p|}{|Patch|} \quad (7)$$

- M8: Some maintainers pointed out that overload caused a high number of maintainers' commits to go unreviewed [56]. Therefore, we define the ratio of self-commits (commits authored by maintainers/committers) that were reviewed by others as an indicator of the strictness of review for patches contributed by contributors with commit rights.

$$QLTY\_RSRO = \frac{\left|\{p \mid p \in MPatch, Reviewer_p \neq \{author_p\}\}\right|}{|MPatch|} \quad (8)$$

where $MPatch = \{p \mid p \in Patch, author_p \in Mntr\}$.

To mitigate the effects of interference and explore the scalability of the MCM, that is, how the model works when more committers are introduced, we fitted regression models, with the response being maintainer burden (represented by M1–M8) and the predictor being the number of committers (#*C*). Because the burden on the maintainers is likely to be affected by the number of patches (#*P*) and maintainers (#*M*), we included these predictors in the models. Each observation represents a month from April, 2009 to January, 2018. We log-transformed skewed variables to satisfy the assumptions of the model. The final regression equation is:

$$(\ln)(M1 \ldots M8) \sim \ln(\#C) + \ln(\#P) + \ln(\#M) \quad (9)$$

3.4 Analysis of the Contextual Success Factors

The MCM has been stimulating wide discussion and attention in the Linux kernel community for some time, and many online documents have been created during this period. Therefore, we conducted a qualitative analysis to help identify the factors that are crucial for successful implementation and operation of the model, and then complemented and strengthened these results with a quantitative analysis on the i915 subsystem.

For the qualitative analysis, we utilized the following procedures. We first used Google to search for the keywords "multiple committers" and "Linux kernel," and used the top 50 most relevant returns as our initial dataset. We then expanded these documents by retrieving the embedded links, and iterated this process until no new document emerged. This process resulted in 73 documents. We read these documents and screened the 30 relevant documents that involve documents from official websites, emails from a mailing list, and articles and comments from maintainers' blogs. The documents we excluded were mainly those that were irrelevant to MCM or not about the Linux kernel. We also conducted email interviews with three maintainers, including two i915 maintainers and one higher-level maintainer, to resolve any confusion in the documents.

We then applied thematic analysis, a widely used technique for identifying and recording "themes" in textual documents [5, 8, 15], to derive success factors of the MCM. Firstly, we read and reread these documents to become familiar with them. Secondly, for each document we analyzed, we systematically generated initial codes related to the usage and preparation of the model. We then examined each initial code to see whether it could be related to a more inclusive code, and also whether there was an opportunity to merge codes. For example, for the sentence "*[c]ontributors who want to apply for committer status should have submitted a few non-trivial patches that have been merged already*," we generated the initial codes "non-trivial patches" and "been merged" that were mapped to the higher order code/category "suitable candidates." These were then mapped to the theme "committers". We also considered edge cases where initial codes could be mapped to multiple categories. For example, the sentence "*[y]ou have your standard issue – overloaded bottleneck*" was identified as relevant to both "overloaded maintainers" and "single point of failure." In this case following discussion among the authors, we decided to add it to the "overloaded maintainers" category, as the main point was the overloaded maintainers, and the bottleneck in this case might not lead to a single point of failure. This analysis was supported by the MAXQDA tool.[15] The analysis revealed different themes related to the contextual success factors. Based on these themes, we quantitatively analyzed the relevant characteristics of the i915 subsystem, and compared them with other subsystems to augment the former results. Eventually, we obtained 61 initial codes that clustered under seven higher order codes/categories. We then compared and further abstracted these into three major themes that represent the factors critical for implementing the MCM, as shown in Table 1.

Table 1: Contextual Success Factors for the MCM

Higher order codes/categories	Themes
Overloaded maintainers Single point of failure	Projects
Suitable candidates Trust among maintainers and committers	Committers
Sufficient precommit testing Strict review process Application of tools to simplify work and reduce errors	Risk mitigation

3.5 Selection of Candidate Committers

The analysis in Section 3.4 suggests that "committers" is one of the three critical success factors for the adoption of the MCM. Capable people are always the key to the success [32]. However, for large-scale projects it is difficult to identify developers with good potential to become committers [31], we therefore proposed an approach to select committers.

[14] The tags of "Reviewed-by" and "Tested-by" are usually used by ordinary contributors.

[15] https://www.maxqda.com/

The approach is based on collaboration network which has advantages in visualizing members' characteristics and relationships [4]. By analyzing the contextual success factors, we found that developers who are "suitable" and achieve "trust among maintainers and committers" are likely to be elected as committers (more details are in Section 4: RQ2.1). Therefore, we construct collaboration network for a subsystem to capture those elements of candidate committers. Any node i represents a contributor. Let w_i be the weight of node i, defined by the number of commits authored by contributor i in a particular timespan, which represents i's capability and suggests whether i is suitable. Let $c_{i,j}$ be the weight of the edge between nodes i and j, i.e., the number of times that the names of both contributor i and j appear in the tags from the same commit message, which represents the level of trust between them. For example, if the contributor i reviewed 10 patches from the contributor j, there will be 10 commits whose author are j and who are also tagged by i, so $c_{i,j} = 10$. In Section 4: RQ2.1, we construct the collaboration network for the i915 subsystem, which proves a close relationship between the committers and maintainers.

Once we construct the collaboration network for a subsystem, we can use the algorithm of identifying the densest subgraph in graph theory to locate the candidate committers. The details for the selecting approach are illustrated in Section 4: RQ2.2.

3.6 Approach for the Validation

To investigate the generality of the MCM, we validated the contextual success factors that were derived from thematic analysis, and the approach for selecting candidate committers. Specifically, we had two aims: (1) validate that the contextual success factors are applicable within the Linux kernel community and that the committers we selected are appropriate, i.e., they are recognized by the subsystems' maintainers; (2) investigate whether these findings can apply to other communities.

For validation within the Linux kernel, we chose the top 50 busiest subsystems among 1,427 driver subsystems (based on Eq. 1) and calculated their candidate committers using the approach proposed in Section 4: RQ2.2. [16] For seven subsystems, we did not screen out the appropriate candidates because these candidates are already overloaded maintainers in other subsystems. We emailed the maintainers of the remaining 43 subsystems—we chose the busiest maintainer of each subsystem to whom we sent the email. We introduced the MCM (with the quantified effect on i915), and asked the maintainers whether they were willing to try it in their subsystems. We also provided the list of candidate committers, and asked for their opinions. Eventually, 38 emails were delivered successfully and we received eight responses. See Section 5.1 for details of the responses.

In order to validate whether our results can apply to other communities, we chose to study Node.js[17] for two reasons. Firstly, Node.js has experienced rapid development, especially since 2015, and has become the most widely used development framework in the world today.[18] Hence it has experienced rapid scaling issues. Secondly, Node.js is different from the Linux kernel: it has typical characteristics of many GitHub projects, such as more loosely-controlled repository access. Also, it has a pull-request based model instead of a patch-based one. We read its official documents and interviewed via email one of its maintainers to obtain his view on scaling Node.js. Specifically, we showed him the contextual success factors and asked whether these factors were consistent with the scaling experience of Node.js. To validate the approach for selecting committers, we applied this approach on development data between January, 2015 and March, 2015 (rapid development period) to predict committer candidates, who we then compared with real committers between April, 2015 and June, 2015.

4 RESULTS

RQ1: To what extent does the MCM work?

1) Effect of the MCM. We compared the maintainers' workload before and after adoption of the MCM in the i915 subsystem based on the eight metrics described in Section 3.3. Fig. 4 shows the values of the different metrics over time.[19] Since the i915 subsystem started using this model in October, 2015, we calculated the monthly mean and median values of each metric both before and after this date, and removed the data before January, 2013 (inactive period) and the data of October, 2015 to reduce noise. To determine whether these two groups of data were significantly different, we utilized the Mann-Whitney U test to compare between two independent groups [45]. With the exception of file complexity (M6), the values of all of the metrics exhibited significant differences before and after the model was adopted. The results are shown in Table 2.

Pressure. As shown in Fig. 4 (a), although the number of commits in this subsystem increased, the number of patches reviewed by the busiest maintainer dropped significantly, especially when the new model was just adopted. Before this model was adopted, the busiest maintainer was responsible for over 90% of the workload (see M2 in Table 2), but after adoption, this value decreased to approximately 30%. M3 reflects the degree of the dispersion of the work, and indicates that the workload gradually distributes evenly among the maintainers and committers. This suggests that after the new workflow was adopted, the maintainers' maximal workload and the risk of a single point of failure were both reduced.

Latency. When maintainers are overloaded, they are unlikely to review patches in a timely manner, which may delay the merging of patches. The latency is shown in Fig. 4 (b). After the MCM was adopted, the latency decreased from approximately five days to one day. Before adoption, the latency of the i915 subsystem was longer than that of the kernel, but it became shorter after adoption.

Complexity. We measured the change in complexity based on the maximum number of contributors/files touched by the maintainers (and committers). As shown in Fig. 4 (c), the complexity of the review relationship (M5) appears to be halved after adoption of the new model, whereas there is no significant difference in the file complexity (M6) (see Table 2). This is reasonable, because maintainers tend to have focused areas.

Quality assurance. As shown in Table 2, both M7 and M8 are larger after adoption of the model. We also compared the i915 subsystem to the overall Linux kernel, as shown in Figs. 4 (d) and (e).

[16] We chose two years as the timespan of commit history to construct the collaboration network.
[17] https://github.com/nodejs
[18] https://insights.stackoverflow.com/survey/2017
[19] We calculated all the metrics for both i915 subsystem and the Linux kernel for comparison. For clarity, we only present the kernel comparison for a few metrics.

Table 2: Statistic Results of the Eight Metrics of i915 Subsystem

Metrics	Throughput			Latency	Complexity		Quality Assurance	
	M1	M2	M3	M4	M5	M6	M7	M8
Mean	154.76, 79.92	0.92, 0.35	0.43, 2.79	5.71, 1.94	22.39, 12.23	40.27, 49.12	2.26, 2.39	0.80, 0.96
Median	150.00, 63.50	0.93, 0.32	0.40, 2.81	5.00, 1.00	23.00, 12.00	39.00, 47.00	2.26, 2.39	0.87, 0.97
P-value/r	7.18E-08/0.69	3.13E-11/0.85	3.03E-11/-0.85	1.55E-08/0.68	4.06E-09/0.75	0.019/-0.27	6.45E-09/-0.74	1.44E-06/-0.61

In each cell, the first number is the mean/median of each metric per month when the traditional workflow is used. The second number is the mean/median when the MCM is used. "r" represents the effect size ($r = Z/\sqrt{N}$). According to [19], $abs(r) \geq 0.1$, $abs(r) \geq 0.3$, and $abs(r) \geq 0.5$ represent small, medium, and large effect sizes, respectively.

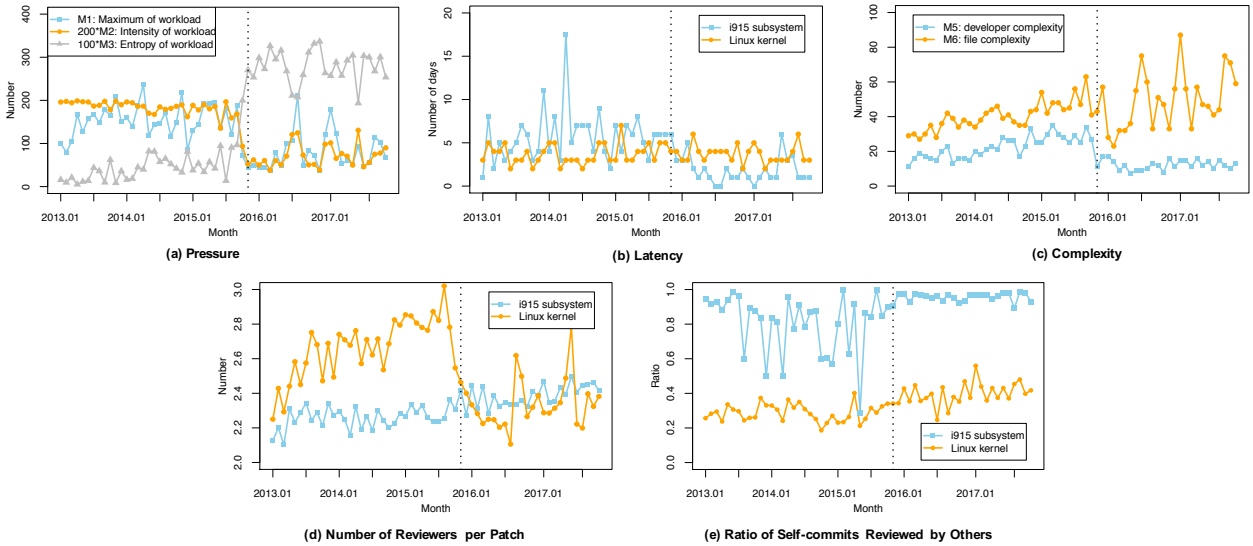

Figure 4: Changes in Measurements before and after Adopting the MCM

Before adoption of the new model, the number of unique reviewers per patch was lower than that of the kernel, but in recent years, it has been somewhat higher. The ratio of self-commits reviewed by others in the i915 subsystem is far higher than that of the kernel, and almost all of the self-commits were reviewed by others after October, 2015. It indicates that the subsystem enforces a strict review process for patches, while giving commit rights to more people.

2) Scalability of the MCM. We fitted regression models (see Equation 9) to analyze the scalability of the MCM. The results are presented in Table 3. In seven of the eight models, the adjusted R^2 was greater than 0.5, while in the remaining one, the R^2 was greater than 0.3. Except for the file complexity (M6), all of the metrics were significantly associated with $\#C$ ($p \approx 0$). It suggests that when more committers are introduced, the pressure, latency, and complexity of review relationship are reduced, whereas the entropy of the workload is increased. We further neglected $\#P$ and/or $\#M$ to exclude erroneous results caused by multicollinearity, while under this condition, the effects of $\#C$ are undiminished. These results indicate that the new model is scalable.

In summary, we find that after the MCM was adopted, the burden of the i915 maintainers was significantly reduced and also, the model scales well to allow more committers. We also find that the number of committers is negatively correlated with the time needed for patch review and the complexity of the review relationship between committers and authors. The review workload is shared by multiple committers, which can reduce the risk of a single point of failure, but may require a strict process to ensure patch quality.

Table 3: Results of Modeling M1–M8

Metric	ln(#C)		ln(#P)		ln(#M)		f^2
	Est	*Std.Err*	*Est*	*Std.Err*	*Est*	*Std.Err*	
M1	-0.10	0.07	0.77	0.06	-0.08	0.06	7.03
M2	-0.92	0.06	0.14	0.06	-0.17	0.05	4.43
M3	0.92	0.04	-0.09	0.04	0.19	0.04	12.7
M4	-0.64	0.13	0.02	0.13	0.07	0.12	0.58
M5	-0.91	0.10	0.17	0.10	0.36	0.09	1.48
M6	-0.11	0.12	0.52	0.12	0.36	0.11	0.79
M7	0.57	0.12	0.07	0.11	0.19	0.10	1.00
M8	0.60	0.14	-0.12	0.14	-0.12	0.13	0.36

The data in the gray cells indicate p-value $< .001$. "f^2" represents the effect size ($Cohen's f^2 = r^2/(1-r^2)$). According to [49], $f^2 \geq 0.02$, $f^2 \geq 0.15$, and $f^2 \geq 0.35$ represent small, medium, and large effect sizes.

RQ2.1: What factors are crucial for successful implementation/operation of the MCM?

The above analysis suggests that the MCM works effectively on the i915 subsystem. To apply it in more projects, it is necessary to clarify the crucial factors for adoption. By applying the approach described in Section 3.4, we obtained seven contextual success factors under three higher-level factors, as shown in Table 1. We describe these factors in detail and refer to the analyzed documents as "DOC#" and the views from interviews with maintainers as "INT". For a deeper understanding of these factors, we present a quantitative analysis of these factors based on the development history data of the i915 subsystem.

Projects (F1). Here, we focus on the factors that suggest a project urgently needs this model. We identify two contextual success factors extracted from 20 initial codes. Before the adoption of the MCM, the maintainer of the i915 subsystem said that *as the*

single maintainer, he gave the subsystem "a bus factor of one" and when he wasn't available for any reason, things simply came to a stop (DOC#7). By distributing the commit right to more contributors, the MCM can effectively reduce the workload of maintainer(s), and handle the risk of a single point of failure that occurs when the sole maintainer takes a break. Naturally, the projects with **overloaded maintainers (F1.1)** and/or **single point of failure (F1.2)** should consider using this model (Note, this does not mean that other projects cannot apply the MCM).

We calculated the workload of the busiest maintainer of the i915 subsystem based on the number of patches signed off by him (M1). We found that before adoption of the new model, the workload of this maintainer was ranked in the top 2.3% of all the maintainers, whether considering all drivers or the overall kernel. In contrast, after just a month of adoption of the MCM, he was ranked in the top 9.8%. Other factors can also indicate that maintainers are overloaded: for example, *difficulty in finding time to take a break, or a significant increase in the review latency* (DOC#7). By analyzing the commit history data of the Linux kernel from June, 2017 to June, 2018, we found that out of 1,824 subsystems, 1,236 subsystems have only one maintainer, indicating that they may face the risk of a single point of failure. There are 75 subsystems with more than 100 active days (days with commits) per year, whereas 37% of them have only one maintainer. This suggests that a number of the subsystems are taking risks, so the MCM has a relatively large number of candidate projects.

Committers (F2). Under this theme, we identified 17 initial codes that map to two factors. The basic requirement for applying the MCM in a project is that *there exists a group of committers having the ability to commit changes to the repository* (DOC#8). This requires that the project has the following two characteristics.

Suitable candidates (F2.1). Commit right is a critical aspect of open source communities [37]. Communities are very cautious when giving someone a commit right [6]. As one of the large-scale, high-quality OSS projects, the success of the Linux kernel is closely related to the strictness of its commit access [10]. However, the effectiveness of the MCM is achieved through relaxing the commit right to a certain extent, which is a high-risk operation that may affect patch quality. To avoid this problem, the community of the i915 subsystem has established a set of strict rules to elect capable contributors as candidate committers.[20] For example, *contributors who want to apply for this right should have submitted a few (5–10 as a rule of thumb) non-trivial patches (not just simple spelling fixes and whitespace adjustment) that have already been merged* (DOC#28).

Trust among maintainers and committers (F2.2). Trust in open source communities is crucial to facilitating their success [1]. Becoming a committer means that the contributor has the maintainer's trust, indicating that they are not only technically competent, but also play a key role in the collaboration network, e.g., *actively participating in reviews and discussions* (DOC#28). Another trust-related issue is that the candidate committers must put the interests of the community ahead of their own personal interests. For example, some subsystems have multiple hardware vendors, that are usually competitors.[3] Modifications by a certain vendor may not be appropriate for other vendors, because their hardware and commercial objectives are somewhat different. *In this case, trust is obviously key within the group, regardless of background or employment. Maintainers must act as gatekeepers for balancing the interests of all parties* (DOC#19).

To understand whether the i915 subsystem satisfied the above requirements before adopting the MCM, we followed the approach in Section 3.5 to construct its collaboration network, as shown in Figure 5. We used the commit data of the i915 subsystem two years before it moved to the new model (October, 2013 to October, 2015). The size of a node characterizes a contributor's capability and the width of edges is positively related to the level of trust between two contributors. The nodes with labels are the contributors who were elected as committers immediately after the subsystem applied the MCM (October, 2015 to December, 2015). It is clear that contributors who have contributed substantial commits and played a key role in the network would be elected as committers, consistent with the results of the qualitative analysis. This observation also underpins the approach for selecting candidate committers proposed in Section 4: RQ2.2.

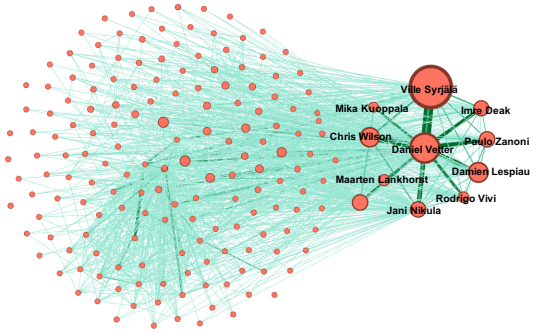

Figure 5: Collaboration Network of the i915 Subsystem

Risk mitigation (F3). When compared to the traditional workflow, which requires all patches be reviewed by the maintainers, the MCM may increase the risk of introducing bad patches—after all, more people means more risk, and capable people are always in short supply. Worse still, because a large number of patches are submitted to the same tree (repository), it is nearly impossible to rebase the tree. Therefore, the subsystem may have to formulate a series of measurements to reduce these risks. Under this theme, we found 25 initial codes that map to three contextual success factors.

Sufficient precommit testing (F3.1). There is almost no way to identify all embarrassing mistakes without testing. *This requires sufficient precommit testing to ensure that obscure corner cases do not break functionality* (DOC#7). For this purpose, the i915 subsystem mainly enforces the following two practices: supporting continuous integration (CI), and formulating detailed testing requirements. CI helps the subsystem to adapt to changes quickly, and to better ensure the quality of patches. In particular, for the drivers that support different types or versions of hardware, the speed of update makes it particularly challenging to ensure the quality and robustness of patches on various platforms. *For the i915 subsystem, Intel provides a CI system that consists of multiple Intel machines spanning as many generations and display types as possible* (DOC#10). *Strict rules are enforced for CI, e.g., a patch series must pass IGT Basic Acceptance Tests (BAT) on all CI machines without causing regressions* (DOC#29).

[20] https://drm.pages.freedesktop.org/maintainer-tools/drm-intel.html

Contributors can obtain the build results in real time (DOC#9). In addition to providing abundant hardware resources, the community of the i915 subsystem sets up detailed testing requirements. *For example, it requires that tests must fully cover user-space interfaces. The subsystem also does pre-merge testing of patch series on a multitude of platforms, so as to confirm that the changes strictly improve the state of the Linux. The results are automatically sent to contributors by emails* (DOC#2). During the interview with an i915 subsystem's maintainer, he specifically pointed out that *it is important to have a CI system integrated with the bug tracking system* (INT).

Strict review process (F3.2). The MCM has strict requirements for patch quality assurance (even stricter than the traditional process because it involves more people). *It requires committers to be confident in the patches they push, in proportion to the impact of those patches* (DOC#29). The confidence must be explicitly documented with special tags (e.g., Reviewed-by, Acked-by, and Tested-by) in the commit message. In one interview, a maintainer stated that "*having at least two people looking at the code drastically reduces the risk of bugs and regressions*" (INT). As shown in Fig. 4 (e), almost all self-commits are reviewed by others in the i915 subsystem, whereas this ratio is only 32% in the overall Linux kernel. *If the committers do not feel comfortable pushing a patch for any reason (technical concerns, conflicting feedback, management or peer pressure, etc.), it is recommended to defer to the maintainers* (DOC#29).

Application of tools to simplify work and reduce errors (F3.3). The responsibility of a maintainer/committer takes considerable time and effort. Therefore, it is necessary to apply development tools to automate and simplify work. For example, the i915 subsystem employs "dim" to provide rich functionality to simplify the work of maintainers and committers. *It can help maintainers automatically test patches, and it provides options for error filtering. It also allows one to conveniently view and manage branches, and send pull requests to the upstream branch* (DOC#30). In an interview, an i915 maintainer pointed out that *"dim" is one of the key factors in the effective operation of the MCM* (INT). *Using tools to avoid (rather than correcting) errors is also key to the model. When someone makes a mistake, a check should be put into the tools to prevent the mistake from recurring, if possible* (DOC#7).

RQ2.2: How should candidate committers be selected when maintainers adopt the MCM?

In order to select candidate committers for a project, we first need to construct its collaboration network as described in Section 3.5, and then locate the featured densest subgraph as described below.

For an induced subgraph of a set of nodes $\mathbf{P} = \{p_1, p_2, \cdots, p_k\}$, we define its density as

$$Density = \frac{1}{k}\left(\alpha \sum_{i \in \mathbf{P}} w_i + (1-\alpha) \sum_{i,j \in \mathbf{P}} c_{i,j}\right) \tag{10}$$

where k is the number of selected nodes, and α is a factor to balance the capacity of individuals and the trust among the group.

The key step of our approach is to determine the densest one among all possible sets of nodes $\mathbf{P}$. This problem can be solved using fractional programming and maximum-flow approaches, first proposed by Goldberg [22]. Apart from the density requirement, we add some extra conditions to select appropriate candidates. We force the algorithm to search for the densest subgraph containing the maintainers of the subsystem, because committers have to work with maintainers. However, if contributors are already overloaded maintainers of other subsystems, we force the algorithm to exclude them. If contributors did not do anything in the past three months (in the selected date range), we consider them to be inactive and not willing to review patches, and thus they are also excluded.

To evaluate the effect of this approach, we employed it to select committers for the i915 subsystem using the data before October, 2015 and compared with the actual committers selected in October, 2015. We found that our approach can select five out of eight true committers, indicating our algorithm is of high accuracy (precision: 100.00%; recall: 62.50%). We did not identify the other three because they were not active (our algorithm cannot see information beyond commit history).

5 VALIDATION

We validated that the results of RQ2 are applicable within the Linux kernel community. Based on the analysis of Node.js community, we found that the results can also apply to other communities.

5.1 Applying the MCM in the Linux Kernel

Our first aim was to validate that the success factors and the approach for selecting candidate committers are significant within the Linux kernel community. We also wanted to gauge maintainers' responses regarding their willingness to accept this model, and any concerns they might have. Based on the approach described in Section 3.6, we eventually received eight responses.[21] Six subsystems (No. 1 – 4, 6, 8) expressed interest in the model, indicating that we identify the **projects (F1)** fairly well. Five subsystems (No. 1, 2, 3, 7, 8) clearly stated that the selected candidate committers were appropriate, indicating not only that the **approach for selecting committers** is effective but also that the requirements for **committers (F2)** we identified are reasonable. As for the other three subsystems, they did not comment on these candidate committers because they did not need the MCM due to the scaling challenge. Below we report the views of the maintainers to help illustrate the applicability of the model for other subsystems.

The maintainers of the No. 2 and No. 8 subsystems clearly indicated that they were applying the MCM and confirmed that the candidate committers that we identified are real committers. The maintainers of the No. 3 subsystem were preparing to apply this model, but still lacked tooling, suggesting that the practice of "**application of tools to simplify work and reduce errors (F3.3)**" is essential. The maintainers of the No. 4 subsystem (the busiest maintainer forwarded the email to another maintainer) showed interest in this model. However, they have concerns regarding the candidates. Because their contributors belong mainly to two competitors, Mellanox and Intel, they worried that the decisions made by the committers from one company might harm the other. Their concerns show clearly that **trust among maintainers and committers (F2.2)** is important for this model. We suggested them to set stricter review rules, e.g., the patches submitted by one company should be reviewed by their competitors, to which they agreed. The No. 1 and No. 6 subsystems had attempted to apply this model, but

[21] No.1–8 subsystems represent ACPI, DRM DRIVERS, GPIO SUBSYSTEM, INFINIBAND SUBSYSTEM, INTEL ETHERNET DRIVERS, MELLANOX ETHERNET SWITCH DRIVERS, MULTIFUNCTI ON DEVICES (MFD), and RADEON AND AMDGPU DRM DRIVERS respectively.

eventually failed, albeit for different reasons. The maintainer of the former explained,"*I respect all of these candidates and I collaborate with all of them, but I don't think they will be interested in the role you are proposing for them.*" This indicates the process of selecting committers is a two-way choice between maintainers and committers, and that motivation is a key factor for candidate committers to become real committers. The maintainers of the No. 4 and No. 7 subsystems also had a similar concern. As for the No. 6 subsystem, the reason for the failed attempt was the heavy workload for the verification and QA teams, which demonstrates the importance of **testing and review (F3.1, F3.2)**, and the necessity of **application of tools to simplify work(F3.3)**. The maintainers of the No. 5 and No. 7 subsystems stated that they would not use this model. We learned that the former subsystem was applying another kind of group model that introduced a verification team to share the review workload of the maintainer, and that our approach for selecting candidate committers could afford insights. Although we found that the maintainer of the No. 7 subsystem was busy (the 29th among 1,427 driver subsystems), he did not think he was overloaded at present. Research suggests that innovation is impeded by both the routine of an existing practice and perceived risks associated with the innovation [51]. Applying this model requires practice and some time to reveal benefits. Therefore, if maintainers are accustomed to the current workflow, they may not want to be the first ones to change and to face new problems.

In conclusion, we observed that although many kernel subsystems are interested in the MCM, the application of the model is not easy. The approach we proposed for selecting candidate committers can help maintainers identify suitable and trustworthy candidate committers, and their motivation to become committers is also a key factor for actually applying the MCM.

5.2 Applying the MCM in Other Communities

Based on the approach described in Section 3.6, we validated that the MCM is applicable to Node.js.

Node.js has 38 subsystems and is currently applying a "3 tier" governance structure similar to the MCM: new contributors are at the bottom providing support for more than 100 active committers (often referred to as "collaborators" using GitHub's terminology). The top-layer comprises Technical Steering Committee members (similar to maintainers in the MCM). However, before 2015, there were fewer than five developers who had commit privileges. The contextual success factors of the MCM we obtained in the i915 subsystem are also applicable to Node.js. In particular, it stipulates a series of rules to guarantee the quality of patch: a pull request may land if it **has approval from more than two developers (F3.1)** and **passes CI (F3.2)**. It **applies tools to simplify work and avoid errors (F3.3)**, e.g., node-core-utils, a CLI (command line interface) toolkit for Node.js core collaborators to facilitate development.[22] In order to select suitable and trustworthy **committers (F2)**, this community stipulates that existing collaborators can nominate someone who has made valuable contributions to become a committer. In contrast to the i915 subsystem, being nominated as a committer in the Node.js community is much easier. It may just require a number of contributions, including submitting pull requests, reviewing code, reporting bugs, etc. To validate our approach for selecting committers, we applied it on the development data between January, 2015 and March, 2015. Compared with real committers between April, 2015 and June, 2015, the approach selected 13 out of 35 true committers (precision: 100.00%; recall: 37.14%).

In conclusion, the contextual success factors we obtained are applicable to other communities, e.g., the Node.js community, and our approach can accurately identify candidate committers.

[22] https://github.com/nodejs/node-core-utils

6 DISCUSSION

We discuss the insights for scaling OSS communities by comparing similarities and differences between the kernel and other communities —with Node.js serving as an exemplar.

6.1 Similarities

For large-scale development teams, the highly experienced developers are the few who are on the top of the pyramid [61], and have commit rights to directly modify the main repository. As the number of patches increases dramatically, these developers come under considerable pressure. In order to reduce the burden on them and effectively utilize their capability, an intuitive approach is to take advantage of the developers in the middle of the pyramid to alleviate the pressure on the upper maintainers. This hierarchical workflow is not rare in OSS projects—Node.js, Apache, and Eclipse, for example. One way to truly scale a community is to distribute control to community members and adopt modular design. This concept is as old as civilization. In the agricultural revolution, instead of each of us gathering and hunting our own food, we created farms that would specialize in growing specific crops. Every empire or nation has a system of delegated authority to local powers. This is the case even in highly regimented systems, such as the crew of a nuclear submarine. In his book *Turn the Ship Around!*, Captain David Marquet showed that the distribution of control and ownership is the most efficient method for improving execution and developing leaders [40]. As for OSS communities which are structured hierarchically, re-routing code review tasks originally destined for top-layer to middle-layer developers by giving them commit rights is a good option. This allows senior developers to focus on complex issues rather than more trivial matters.

Decentralization also needs supervision. For large-scale OSS projects, quality is always paramount, all the more so since these systems now provide the critical infrastructure for society. Both the i915 subsystem and Node.js reduce the review burden by distributing commit rights to the lower level, who do surface quality risks. Consistent with our findings, Node.js has formulated a series of measurements to mitigate these risks, e.g., any committer is authorized to approve any other contributor's work but they are not permitted to approve their own pull requests. This suggests that when delegating maintainer authority, code review and testing must be stricter, for which our analysis provides specific recommendations.

6.2 Differences

Although the i915 subsystem and Node.js both introduced multiple committers, the requirement for committer capability is different. Linux is the world's largest and most pervasive OSS project. The

Linux kernel is the largest component of the Linux operating system and is charged with managing the hardware, running user programs, and maintaining the security and integrity of the whole system. Compared to other projects, contributing to the Linux kernel is more difficult because developers have to understand the basic principles of a computer system to understand the intricacies of the hardware and software, which is also confirmed by the maintainer of Node.js. He said, "*for most projects on GitHub, most contributions are not very complex and only need a few adjustments (e.g., compliance with basic style requirements) that represent one of the largest burdens during review. However, the bar for contributing to the Linux kernel is so high, it provides a natural filtration mechanism*". It is easier to become a committer in Node.js, after several contributions, which can explain why there are more than 100 active committers conducting reviews in Node.js, whereas in the i915 subsystem, becoming a committer is not so easy. Furthermore, committers may be unwilling to review code because it takes considerable time and effort (see Section 5.1). These are the main reasons why the phenomenon of multiple committers is common in other OSS communities but has aroused a heated discussion from the kernel community. In addition, the involvement of many hardware vendors with conflicting interests in the Linux kernel makes it more difficult to select appropriate committers. One possible solution is training candidate committers as early as possible and encouraging developers to integrate code review into their daily routine. Any time a change occurs, it should be communicated to the different stakeholders who can make the appropriate adjustments in accordance with the objectives of the project instead of personal interests.

7 LIMITATIONS

We identified the review workload of the maintainers by looking at the "signed-off-by" field in the commit messages and the MAINTAINERS file, which enables us to confirm that the individual named by the signed-off-by field is indeed a maintainer. However, some problems are difficult to avoid, e.g., individuals who do not wish their names to appear in the MAINTAINERS file are excluded. Also, because the time of "signed-off-by" of a patch is not recorded, we used the commit time instead when calculating the review latency. This may not accurately reflect the review time of maintainers who do not use Git, since, in this case, the patches are usually committed by higher level maintainers. Because we analyzed only the mainline repository of the Linux kernel, a large number of patches that did not gain acceptance into the mainline repository were excluded [21]. Reviewing these patches also consumes maintainers' time. However, according to Zhou et al. [61], "*it is easier for the maintainer to not accept the code at all. To accept code, it takes time to review it, apply it, handle problems that may ensue, possibly fix any problems that happen later on when contributors disappear, and maintain it for the next 20 years.*" Therefore, it is reasonable to regard the review workload of the accepted patches as the main burden on the maintainers.

To increase construct validity, we interviewed Linux kernel maintainers and inspected various online resources. Also, the qualitative analysis was complemented and strengthened by quantitative validation. For example, we attained a reasonable understanding of the main burden on maintainers by reading relevant online documentation and by communicating with the maintainers. After deriving the contextual success factors for the MCM, we also explored whether the i915 subsystem satisfied these factors quantitatively, and confirmed our findings based on the feedback from the maintainers of the kernel community and Node.js community.

Threats to external validity mainly come from two sources. 1) Studying only the i915 subsystem limits external validity. The MCM was first adopted by i915 in the Linux kernel, and while the model has been applied by a few subsystems since, the i915 is the one that has used the model for a relatively long time, which enables us to explore the effect by analyzing the development history data. However, precisely because there are only a handful of subsystems attempting to use the new workflow, studying the model is necessary and timely. 2) The uniqueness of the Linux kernel limits external validity. As a prominent OSS project, the Linux kernel has many unique practices that have been referenced and used by many other projects [42]. We found that OSS projects that emulated the traditional kernel review process also encountered the pressure on maintainers, e.g., the dpdk project.[3] The implication of delegating commit rights and suggestions for risk mitigation are applicable to general OSS projects, as discussed in Section 5.2 and Section 6. Our framework for quantifying the maintainers' workload and our analysis of the MCM may benefit both the subsystems of the Linux kernel and other projects encountering similar problems.

8 CONCLUSIONS

The scaling of OSS projects is a common and challenging problem that determines whether projects which are growing can be sustained. However, no in-depth study of this topic has been published. In order to address this problem, we conducted an empirical study of the Linux kernel which is struggling with scaling issues: many developers have raised concerns regarding whether the current development workflow can handle the increasing number of contributors and patches. While the current workflow guarantees a strict review process, it places significant pressure on maintainers, who bear the risk of a single point of failure.

We investigated a new model—the MCM—that has been adopted by the i915 subsystem in the Linux kernel to relieve the burden on maintainers. We found that after adoption, the maintainers' workload were significantly reduced. A review process is strictly enforced to guarantee the quality of patches, and the effect of the model increases when more committers are introduced. For other projects with overloaded maintainers, if there are trustworthy candidate committers and measures for risk mitigation, they are suitable for this model. Our proposed framework to quantify the review workload of maintainers may lead to a better understanding of the factors that impede rapidly growing projects. Knowledge of the factors for implementing the MCM, and the approach for selecting candidate committers, can help other suitable subsystems to optimize their workflow to achieve a more efficient review process while scaling up. To facilitate replication of our work or other types of future work, we provide the data, scripts and retrieved materials used in this study in https://archive.softwareheritage.org/swh:1:dir:33d1a7540c8dc449b90d4771f6dd52ba45e26879/.

ACKNOWLEDGMENTS

This work is supported by the National key R&D Program of China Grant 2018YFB1004201, the National Natural Science Foundation of China Grants 61825201.

REFERENCES

[1] Maria Antikainen, Timo Aaltonen, and Jaani Väisänen. 2007. The role of trust in OSS communities — Case Linux Kernel community. In *Open Source Development, Adoption and Innovation*, Joseph Feller, Brian Fitzgerald, Walt Scacchi, and Alberto Sillitti (Eds.). Springer US, Boston, MA, 223–228.

[2] Olga Baysal, Reid Holmes, and Michael W Godfrey. 2013. Developer dashboards: The need for qualitative analytics. *IEEE software* 30, 4 (2013), 46–52.

[3] Andrea Bonaccorsi and Cristina Rossi. 2002. Why Open Source software can succeed. *Research Policy* 32, 7 (2002), 1243–1258.

[4] Ulrik Brandes, Patrick Kenis, Jürgen Lerner, and Denise Van Raaij. 2009. Network analysis of collaboration structure in Wikipedia. In *Proceedings of the 18th international conference on World wide web*. ACM, 731–740.

[5] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.

[6] Brett Cannon. Accessed January, 2019. Becoming an OSS project maintainer is about trust. https://snarky.ca/becoming-an-oss-project-maintainer-is-about-trust/.

[7] Andrea Capiluppi, Patricia Lago, and Maurizio Morisio. 2003. Evidences in the evolution of OS projects through Changelog Analyses. (01 2003).

[8] Jailton Coelho and Marco Tulio Valente. 2017. Why modern open source projects fail. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, NY, USA, 186–196.

[9] Robert E. Cole. 2003. From a Firm-Based to a Community-Based Model of Knowledge Creation: The Case of the Linux Kernel Development. *Organization Science* 14, 6 (2003), 633–649.

[10] Jonathan Corbet. 2008. How to participate in the Linux community. *A guide to the kernel development process. The Linux Foundation.[52] viitattu* 22 (2008), 2017.

[11] Jonathan Corbet. Accessed January, 2019. Group maintainership models. https://lwn.net/Articles/705228/.

[12] Jonathan Corbet. Accessed January, 2019. On Linux kernel maintainer scalability. https://lwn.net/Articles/703005/.

[13] Jonathan Corbet. Accessed January, 2019. Some numbers from the 4.19 development cycle. https://lwn.net/Articles/767635/.

[14] John W Creswell and J David Creswell. 2017. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications.

[15] Daniela S Cruzes and Tore Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, Banff, AB, Canada, 275–284.

[16] Nicolas Ducheneaut. 2005. Socialization in an Open Source Software Community: A Socio-Technical Analysis. *Computer Supported Cooperative Work* 14, 4 (2005), 323–368.

[17] Nadia Eghbal. 2016. *Roads and bridges: The unseen labor behind our digital infrastructure*. Ford Foundation.

[18] Robert Fichman and Chris Kemerer. 1993. Adoption of software engineering process innovations: The case of object orientation. *Sloan Management Review* 34 (01 1993).

[19] Catherine O Fritz, Peter E Morris, and Jennifer J Richler. 2012. Effect size estimates: current use, calculations, and interpretation. *Journal of experimental psychology: General* 141, 1 (2012), 2.

[20] Daniel M. German. 2003. The GNOME project: a case study of open source, global software development. *Software Process Improvement & Practice* 8, 4 (2003), 201–215.

[21] Daniel M German, Bram Adams, and Ahmed E Hassan. 2016. Continuously mining distributed version control systems: an empirical study of how Linux uses Git. *Empirical Software Engineering* 21, 1 (2016), 260–299.

[22] Andrew V Goldberg. 1984. *Finding a maximum density subgraph*. University of California Berkeley, CA, California, USA.

[23] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An Exploratory Study of the Pull-based Software Development Model. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 345–355. https://doi.org/10.1145/2568225.2568260

[24] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work Practices and Challenges in Pull-based Development: The Contributor's Perspective. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 285–296. https://doi.org/10.1145/2884781.2884826

[25] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. 2015. Work Practices and Challenges in Pull-based Development: The Integrator's Perspective. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 358–368. http://dl.acm.org/citation.cfm?id=2818754.2818800

[26] Bronwyn H Hall and Beethika Khan. 2003. *Adoption of new technology*. Technical Report. National bureau of economic research.

[27] Dietmar Harhoff, Joachim Henkel, and Eric Von Hippel. 2003. Profiting from voluntary information spillovers: how users benefit by freely revealing their innovations. *Research policy* 32, 10 (2003), 1753–1769.

[28] Michael Hilton. 2016. Understanding and Improving Continuous Integration. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 1066–1067. https://doi.org/10.1145/2950290.2983952

[29] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 197–207. https://doi.org/10.1145/3106237.3106270

[30] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-source Projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 426–437. https://doi.org/10.1145/2970276.2970358

[31] A. Jongyindee, M. Ohira, A. Ihara, and K. Matsumoto. 2011. Good or Bad Committers? A Case Study of Committers' Cautiousness and the Consequences on the Bug Fixing Process in the Eclipse Project. In *2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*. IEEE, 116–125. https://doi.org/10.1109/IWSM-MENSURA.2011.24

[32] Minnesh Kaliprasad. 2006. The human factor I: Attracting, retaining, and motivating capable people. *Cost Engineering* 48, 6 (2006), 20.

[33] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. 2016. Code review quality: how developers see it. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, Austin, TX, USA, 1028–1038.

[34] Greg Kroah-Hartman. Accessed January, 2019. I don't want your code: Linux Kernel Maintainers, why are they so grumpy? https://github.com/gregkh/presentation-linux-maintainer/blob/master/maintainer.pdf.

[35] Georg Von Krogh and Eric Von Hippel. 2003. Special issue on open source software development. *Research Policy* 32, 7 (2003), 1149–1157.

[36] C. Lebeuf, M. Storey, and A. Zagalsky. 2018. Software Bots. *IEEE Software* 35, 1 (January 2018), 18–23. https://doi.org/10.1109/MS.2017.4541027

[37] John Boaz Lee, Akinori Ihara, Akito Monden, and Ken-ichi Matsumoto. 2013. Patch reviewer recommendation in oss projects. In *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific*, Vol. 2. IEEE, Williamsburg, VI, USA, 1–6.

[38] M M Lehman, J F Ramil, P D Wernick, D E Perry, and W M Turski. 1997. Metrics and Laws of Software Evolution - The Nineties View. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*. IEEE, 20–32.

[39] Laura Macleod, Michaela Greiler, Margaret Anne Storey, Christian Bird, and Jacek Czerwonka. 2017. Code Reviewing in the Trenches: Understanding Challenges and Best Practices. *IEEE Software* PP, 99 (2017), 1–1.

[40] L David Marquet. 2013. *Turn the ship around!: A true story of turning followers into leaders*. Penguin.

[41] Shane Mcintosh, Bram Adams, Bram Adams, and Ahmed E. Hassan. 2014. The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects. In *Working Conference on Mining Software Repositories*. ACM, New York, NY, USA, 192–201.

[42] Tom Mens, Maálick Claes, Philippe Grosjean, and Alexander Serebrenik. 2014. Studying evolving software ecosystems based on ecological models. In *Evolving Software Systems*. Springer, 297–326.

[43] Rahul Mishra and Ashish Sureka. 2014. Mining Peer Code Review System for Computing Effort and Contribution Metrics for Patch Reviewers. In *Mining Unstructured Data*. IEEE, Victoria, BC, Canada, 11–15.

[44] Audris Mockus, Roy T. Fielding, and James Herbsleb. 2000. A Case Study of Open Source Software Development: The Apache Server. In *International Conference on Software Engineering*. ACM, Limerick, Ireland, 263–272.

[45] Nadim Nachar et al. 2008. The Mann-Whitney U: A test for assessing whether two independent samples come from the same distribution. *Tutorials in Quantitative Methods for Psychology* 4, 1 (2008), 13–20.

[46] Theodore S Rappaport. 2002. Wireless Communications–Principles and Practice, (The Book End). *Microwave Journal* 45, 12 (2002), 128–129.

[47] Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret Anne Storey. 2014. Peer Review on Open-Source Software Projects:Parameters, Statistical Models, and Theory. *Acm Transactions on Software Engineering & Methodology* 23, 4 (2014), 1–33.

[48] Walt Scacchi. 2003. Understanding open source software evolution. Applying, breaking and rethinking the laws of software evolution.

[49] Arielle S Selya, Jennifer S Rose, Lisa C Dierker, Donald Hedeker, and Robin J Mermelstein. 2012. A practical guide to calculating Cohen's f2, a measure of local effect size, from PROC MIXED. *Frontiers in psychology* 3 (2012), 111.

[50] Claude Elwood Shannon, Warren Weaver, Bruce Hajek, and Richard E Blahut. 1950. The mathematical theory of communication. *Physics Today* 3, 9 (1950), 31–32.

[51] Jagdish N Sheth and Walter H Stellner. 1979. *Psychology of innovation resistance: The less developed concept (LDC) in diffusion research*. Number 622. College of Commerce and Business Administration, University of Illinois.

[52] Xin Tan. 2019. Reducing the Workload of the Linux Kernel Maintainers: Multiple-Committer Model. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 1205–1207. https://doi.org/10.1145/3338906.3342490

[53] Xin Tan and Minghui Zhou. 2019. How to Communicate when Submitting Patches: An Empirical Study of the Linux Kernel. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 108 (Nov. 2019), 26 pages. https://doi.org/10.1145/3359210

[54] Linus Torvalds and David Diamond. 2001. Just for Fun: The Story of an Accidental Revolutionary. *Harperbusiness* 238, 6-7 (2001), S87.

[55] Daniel Vetter. Accessed January, 2019. Maintainers Don't Scale. https://kernel-recipes.org/en/2016/talks/maintainers-dont-scale/.

[56] Daniel Vetter. Accessed January, 2019. Vetter: Linux Kernel Maintainer Statistics. https://lwn.net/Articles/752563/.

[57] Bob Wescott. 2013. *Every Computer Performance Book: How to Avoid and Solve Performance Problems on The Computers You Work With.* CreateSpace Independent Publishing Platform.

[58] Mairieli Wessel, Bruno Mendes de Souza, Igor Steinmacher, Igor S. Wiese, Ivanilton Polato, Ana Paula Chaves, and Marco A. Gerosa. 2018. The Power of Bots: Characterizing and Understanding Bots in OSS Projects. *Proc. ACM Hum.-Comput. Interact.* 2, CSCW, Article 182 (Nov. 2018), 19 pages. https://doi.org/10.1145/3274451

[59] Mairieli Wessel, Igor Steinmacher, Igor Wiese, and Marco A. Gerosa. 2019. Should I Stale or Should I Close?: An Analysis of a Bot That Closes Abandoned Issues and Pull Requests. In *Proceedings of the 1st International Workshop on Bots in Software Engineering (BotSE '19)*. IEEE Press, Piscataway, NJ, USA, 38–42. https://doi.org/10.1109/BotSE.2019.00018

[60] Joel West and Scott Gallagher. 2006. Challenges of open innovation: the paradox of firm investment in open-source software. *R&d Management* 36, 3 (2006), 319–331.

[61] Minghui Zhou, Qingying Chen, Audris Mockus, and Fengguang Wu. 2017. On the Scalability of Linux Kernel Maintainers' Work. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, New York, NY, USA, 27–37.

[62] Minghui Zhou and Audris Mockus. 2015. Who Will Stay in the FLOSS Community? Modelling Participant's Initial Behaviour. *Software Engineering IEEE Transactions on* 41, 1 (2015), 82–99.

[63] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. 2016. Effectiveness of Code Contribution: From Patch-based to Pull-request-based Tools. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, 871–882.