Reducing the Workload of the Linux Kernel Maintainers: Multiple-Committer Model

Anonymous

***Abstract*—Large-scale ecosystems, such as the Linux kernel, have gone through decades of development, substantially growing in scale and complexity. In the traditional development process, each maintainer servers as a the "gatekeeper" for the subsystem that s/he maintains. As the numbers of code patches (submissions) and authors significantly increase, the maintainers of these subsystems find themselves under considerable pressure, which may cause them to become a single point of failure. A few of subsystems have begun to use new workflows, e.g., multiple-committer model, to address these issues. However, it is unclear whether the new model works and which subsystems are suitable to apply it. Therefore, we conduct an empirical study on the i915 subsystem of the Linux kernel which currently uses the multiple-committer model. We explore the effect of the model on this subsystem with respect to five dimensions: throughput, latency, overwork, complexity and quality assurance. We find that this model significantly reduces the maintainers throughput, review latency and overwork, and simplifies their review network. The quality of the patches is also guaranteed due to a strict review mechanism. We propose that subsystems that have overloaded maintainers, that share code ownership and affiliations, that possess a stable core team and that practice continues integration are suitable for adoption of this model. The success of the model is closely related to sufficient precommit testing, strict review rules, necessary documentation, and the use of automatic development tools. We validate our findings via email interviews with Linux maintainers. We expect that our proposed framework of quantifying maintainers' workload and the prerequisites and practices of multiple-committer model may help the rapidly growing Linux kernel and other similar ecosystems to adapt to changes and remain sustain.**

I. Introduction

Over the past several decades, free/libre open source software (FLOSS) projects have grown rapidly in popularity due to their unique advantages, e.g., low cost and high quality [1, 2]. During this time, many prominent projects, such as the Linux kernel and Apache HTTP Server, emerged [3]. After decades of development, many of these projects are no longer merely hobbyist projects [4] but rather represent the critical computing infrastructure for our society. Both the scale and complexity of these projects have undergone considerable changes, e.g., the Linux kernel version 4.13 has more than 24.7 million lines of code contributed by over 1,600 contributors representing over 200 corporations,[1] whereas version 0.01 in 1991 has only 10,239 lines of code.[2] These communities established their own workflows in the early stages of the projects. In general, a small core group does most of the work and coordinates a considerable larger group of peripheral participants [3, 5, 6]. The authority of this core group arises naturally as a result of their contributions to the project, such as writing code or reviewing patches [7, 8]. However, recently, increasing numbers of contributors in numerous FLOSS projects are concerned regarding whether the traditional workflow can handle the current mass of patches or even more in the future, especially in the Linux kernel community [9].[3]

The code base of the Linux kernel is logically organized into many subsystems, each of which has a designated maintainer – a developer who is ultimately responsible for the code within that subsystem and possesses the commit right to the subsystem that s/he maintains. Almost all of the patches are selected and transferred by the corresponding maintainers, layer by layer, until they are merged into the mainline repository.[4] The maintainers of the higher level need to pull patches from the lower level maintainers. It is an undeniable fact that this hierarchical review workflow is a key factor in maintaining the high quality of the kernel [10].

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 [11] and hundreds of emails a day [12]. Worse still, many subsystems have only one maintainer, which means that if the maintainer takes a vacation, becomes ill, or simply becomes busy at their day job, there is no one who can replace her/him. This issue suggests that if you become a maintainer, you need to spend a substantial amount of time on the project and that you are less able to take a break, which makes developers hesitant to become maintainers [13]. Being a maintainer or training new maintainers is very difficult work since, as a maintainer, you should not only review patches but also shoulder the responsibility of sending that work upstream and acting as a liaison for complaints arriving from other subsystems [13]. This series of problems has aroused widespread concern from both the community [14] and researchers [15]. Therefore, it is necessary to modify the original workflow to adapt to the growing number of patches. Attempting to apply a new workflow (in a subsystem) is relatively difficult in practice, which may result in unexpected problems and complaints from other maintainers. Currently, only a handful of pioneering subsystems are using group models rather than the traditional model, while most other subsystems are on the sidelines. The multiple-committer model is one of these models, and it is endorsed by the subsystems currently applying it.[5] However, it

[1]https://www.linuxfoundation.org/2017-linux-kernel-report-landing-page/

[2]https://www.linuxcounter.net/statistics/kernel

[3]http://mails.dpdk.org/archives/moving/2016-November/000060.html

[4]https://www.kernel.org/doc/html/latest/process/index.html

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

is not clear to date whether this new model truly works (RQ1), which subsystems are suitable for the model if it does work (RQ2), and which practices to follow when using the model (RQ3). Many subsystems are reluctant to try a new workflow, even though the current workflow repeatedly present problems.

In this study, we aim to address these issues. We conduct an empirical study on the i915 subsystem, which currently utilizes a multiple-committer model in which many regular contributors are able to commit changes to the repository. We focus on the main burden of the maintainers' work: review patches. By defining five dimensions of metrics, including throughput, latency, overwork, complexity and quality assurance, we quantify the effect of the new workflow on the i915 subsystem based on project repository data. We find that the multiple-committer model can significantly reduce the maintainers' workload, review latency, and overwork, as well as simplifying the review network. The quality of patches is also guaranteed. By analyzing usage scenario and interviewing with the maintainers, we recommend overloaded subsystems that have mechanisms for ensuring patch quality and candidate committers to adopt the new model. The model's smooth operation requires the following practices: sufficient precommit testing, strict review rules, necessary documentation and automatic development tools. These results not only help the Linux kernel but also provide insights into expanding ecosystems in general.

II. Methodology

We applied a mixed-method approach [16] to understanding how maintainers work in the Linux kernel community, especially in the i915 subsystem. We used qualitative methods to analyze the general workflow and developed five dimensions by which to characterize the main burden on maintainers' work. We also carefully analyzed the multiple-committer model based on the i915 subsystem. To avoid interfering with the work of busy contributors and considering that the Linux kernel is a prominent FLOSS project with a plethora of online documentation, our qualitative study is based primarily on artifacts recorded in the project web pages, technical blogs, and related websites. To obtain information that could not be obtained from existing resources, we conducted small interviews via email.

For quantitative analysis, we focused on the workflow of the i915 subsystem and obtained and prepared data from the mainline Git repository. We defined ten metrics based on the aforementioned five dimensions to evaluate the effect of the multiple-committer model.

We start by describing the general development process of the Linux kernel and its main problems in Section II-A, and we subsequently introduce the multiple-committer model and the context for the i915 subsystem in Section II-B. We explain the workflow analysis and metrics in Section II-C.

A. General Workflow

To understand the general workflow of the Linux kernel, and the duties of maintainers, we searched the relevant documents. In particular, we utilized the following procedures: 1) We read the online documentation on the official websites of the Linux kernel[4] to obtain a comprehensive understanding of the kernel development process. 2) We used Google to search for the keywords "the Linux kernel" and "maintainers", and we read documentation related to maintainers' workflow. 3) After the two previous steps, we had a reasonable understanding of the development process and the roles of the maintainers. Then, we interviewed five maintainers, who manage the subsystems at different scales and for different functionalities, by email to resolve any confusion from the documentation and to better understand problems with the general workflow. Eventually, we obtained the following information.

The kernel development community is organized as a hierarchy, with developers submitting patches to maintainers who, in turn, commit those patches to a repository (with each maintainer maintaining their own repository), and push them upstream to higher-level maintainers. This hierarchy logically resembles the directory hierarchy of the kernel source itself. This process requires all maintainers in the chain to take responsibility for the patches, especially the maintainers who introduce patches from authors into the repositories. After all, the maintainers who pull patches from lower level maintainers in the hierarchy have little time to carefully review each patch. Instead, they usually conduct integration tests, a fact that was confirmed by the maintainers during the interviews. The maintainers in our interviews indicated that reviewing patches requires a considerable amount of energy while the number of patches is still increasing. We also found from online blogs that certain maintainers worry that they may be burned out when undertaking such heavy review work [13, 14]. Therefore, in this study, we focus on the main burden on maintainers: reviewing the patches submitted directly by the authors.

In addition to the difficulty of dealing with the increasing review workload, the general workflow also brings the risk of a single point of failure. Maintainers, who act as gatekeepers for the portion of the kernel they manage, are not only responsible for the subsystems they manage but also need to coordinate with the wider kernel community. If the maintainer leaves, the development of the subsystem will be affected. Training a new maintainer to replace her/him is also not an easy task [13].

Fig.1 shows the growth of the Linux kernel. The numbers of commits, modified files, contributors, and maintainers appear to grow over time. Although the number of maintainers continues to grow at a pace similar to that of the number of patches, significant evidences – for example, complaints by maintainers' [17], large number of maintainers' commits unreviewed [18], and lengthy review time – indicate that the community fails to mentor and train new maintainers at a pace sufficient to keep up with the growth of the kernel [13]. Zhou et al. explained that this is because the practice of assigning multiple maintainers to a file yields only a 1/2-power increase in productivity [15].

[6]From April, 2009, the Linux kernel has included a file named MAINTAINERS that contains information of maintainers. Therefore, we obtained the statistics of maintainers after 2009.

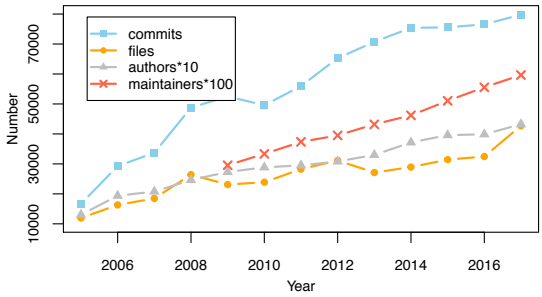

Fig. 1. Growth of the Linux Kernel[6]

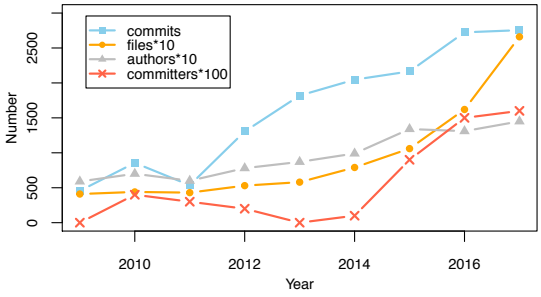

Fig. 2. Growth of i915 Subsystem[9]

B. Multiple-Committer Model

To reduce the workload of maintainers and the latency of kernel development, a few subsystems have attempted to apply new models. The multiple-committer model is one of the group maintainership models that was proposed and applied by i915 subsystem in October 2015. We searched for the relevant blogs written by the maintainers of the subsystem and browsed the official website to understand how the model works.

In the general workflow, only the maintainers have the right to commit patches to the repositories they maintain, whereas the multiple-committer model gives commit rights to regular contributors, allowing them to commit patches to the same repository as the maintainer(s). Because the committers[7] share a great deal of review work with the maintainer(s), the workload of the maintainer(s) is greatly reduced. Contrasted to the committers, the main job of the maintainer(s), at this point, is to communicate with the outside, including coordinating with other subsystems, sending patches to the upstream, and taking task blames for all the faults.[8]

To our knowledge, only the i915 and drm-misc subsystems in the Linux kernel are currently using a multiple-committer model to review and commit patches. However, for the drm-misc subsystem, there are far fewer committers than maintainers. In this case, the effects of the multiple-committer model are not obvious. Therefore, We choose to study the i915 subsystem. The i915 subsystem is INTEL DRM DRIVERS, which supports all integrated GFX chipsets with both Intel display and rendering blocks, with the exception of a few very early models.[8] All patches related to this subsystem are eventually submitted to the repository at "git://anongit.freedesktop.org/drm-intel". Its main files are under "drivers/gpu/drm/i915" of the Linux kernel's code directory. The number of maintainers of this subsystem has fluctuated between one and three. As shown in Fig. 2, the numbers of commits, modified files, contributors, and committers appear to grow over time.

C. Workflow Analysis and Metrics

1) Data Preparation: 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 records the hash of patch, the author of the patch, the person who committed the patch, the time at which the patch was committed, a description of the patch, and the names of the affected files. We also obtained the maintainer of the file and the repository for that file. 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. 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").

2) Identification of the Maintainers' Workload: The Linux kernel uses "signed-off-by" to track the patches. This tag is a simple line at the end of the commit message of each patch that certifies the person wrote it or has the right to pass it on as an open-source patch.[4] In the interviews with the maintainers, we found that in general, there are two types of contributors that can add this tag to a patch: the authors of the patch and maintainers who accepted it from the author. Because maintainers in the upper layers accept patches by "pull request" from maintainers in the lower layers, there is no chance for them to modify the patches. Therefore, by analyzing this tag, we can find the maintainers who first introduce (commit) the patches to the hierarchical repositories of the Linux kernel. In this study, we focus on the review workload during this process, which is consistent with the view that "*these maintainers take on the main workload of review,*" expressed by our interviewee. For each patch, if the individual is not the author of the patch and her/his name is in the MAINTAINERS file (The version of the MAINTAINERS file for a given month corresponds to the month during which the patch was committed.), we consider that he actually reviewed the patch. Eventually, the review workload of the maintainers are obtained. We did not use patch's "committer" domain because some contributors do not use GIT, therefore "committer" does not record the name of the individual who accepted a patch from the author [19].

We extracted the contributors who committed patches to the repositories, but whose names do not appear in the corresponding version of MAINTAINERS file. We treat these individuals as committers.

3) Metrics for Maintainers' Workload: We focus on the maintainers' review work and, more accurately, the review

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

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

[9]Although the community indicated that the new workflow was used in Oct. 2015, the data shows that committers started to appear as early as 2010. However, the number of committers has increased rapidly since Oct. 2015.

efforts that maintainers exert when introducing the patches from authors into their own repositories. In this sense, we can treat maintainers (and committers) as computer systems that process (review) patches. The performance of a computer system represents the amount of work it accomplishes, which can be evaluated by the metrics, e.g., throughput and latency [20]. Borrowing the concepts from computer systems [21], we evaluate the performance of maintainers in a subsystem with ten metrics from five dimensions. All of the metrics are aimed at a particular subsystem and a particular timespan. The related definitions and proposed metrics are as follows.

- $\boldsymbol{Mntr}$: the set of maintainer(s) and committers;
- $\boldsymbol{Patch}$: the set of patches;
- $\boldsymbol{i}$: any maintainer/committer, $\forall i \in Matn$;
- $\boldsymbol{p}$: any patch, $\forall p \in Patch$;
- $\boldsymbol{Patch_i}$: the set of patches signed off by contributor i;
- $\boldsymbol{author_p}$: the author of patch p;
- $\boldsymbol{Reviewer_p}$: the set of the reviewers of patch p;
- $\boldsymbol{File_p}$: the set of files modified in patch p;
- $\boldsymbol{c\_t_p}(\boldsymbol{a\_t_p})$: the commit (author) time of patch p;
- $\boldsymbol{t1_i}(\boldsymbol{t2_i})$: the eight (consecutive) productive hours per day of maintainer/committer i;
- $\boldsymbol{r_i}$: the ratio of patches signed off by maintainer/committer i, $r_i = \frac{\#Patch_i}{\#Patch}$.

Throughput. Throughput is the maximum rate of production or the maximum rate at which something can be processed. Borrowing this concept, We define the workload of the busiest maintainer as the bottleneck for the subsystem processing patches. We also define two additional metrics that describe the distribution of the review work among maintainers (committers), which can reflect the risk of a single point of failure.

- M1. Maximum review workload: the number of the patches reviewed by the busiest maintainer.
$$Max\_Workload = max_i \#Patch_i \tag{1}$$
- M2. Intensity of review workload: the ratio of review work done by the busiest maintainer.
$$Intensity = \frac{Max\_Workload}{\#Patch} \tag{2}$$
- M3. Entropy of review workload: the dispersion of review workload assignment, borrowed from information theory [22].
$$Entropy = -\sum_i r_i log_2 r_i \tag{3}$$

Latency. Latency is a measure of the time delay experienced by a system. In our study, we define latency as the length of time that a patch (last version) being submitted by the author to the time being accepted by the first maintainer. We consider this metric because a long latency means 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 [23].

- M4. Latency of review workload: the median review time of the patches.
$$Delay = median_p\{c\_t_p - a\_t_p\} \tag{4}$$

Overwork. This concept is borrowed from load testing. In our study, it represents the ratio of the work that was completed outside the normal working time. We define eight hours a day as normal working time. Considering that the contributors of FLOSS projects include both volunteers and employees from companies and that volunteers have flexible working hours, we define the following two metrics.

- M5: Ratio of workload outside the eight most productive hours per day.
$$Overwork_1 = \frac{\sum_i \#Patch_i^{(outside\ t1_i)}}{\#Patch} \tag{5}$$
- M6: Ratio of workload outside the eight most productive consecutive hours per day.
$$Overwork_2 = \frac{\sum_i \#Patch_i^{(outside\ t2_i)}}{\#Patch} \tag{6}$$

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 [24], and the contact and communication with developers [25]. Based on these two considerations, we define two metrics to characterize the complexity of the maintainers' review work. We choose the maximum complexity of all the maintainers to reflect the bottleneck of the subsystem.

- M7. Maximum developer complexity: Among all the maintainers, the maximum number of authors reviewed by a certain maintainer.
$$Max\_DC = max_i \#author_j,\ j \in Patch_i \tag{7}$$
- M8. Maximum file complexity: Among all the maintainers, the maximum number of files reviewed by a certain maintainer.
$$Max\_FC = max_i \#File_j,\ j \in Patch_i \tag{8}$$

Quality assurance. Patch review ensures that the patches are of high quality. The number of reviewers has been found important to ensure patch quality [26, 27, 28].

- M9. We use the average number of unique reviewers per patch as the indicator of the quality assurance. We retrieved all the tags that indicate some sort of review, e.g., "reviewed-by", "signed-off-by", or "tested-by."
$$Q_1 = \frac{\sum_p \#Reviewer_p}{\#Patch} \tag{9}$$
- M10. Some maintainers pointed out that the overload caused a high number of maintainers' commits to go unreviewed [13]. Therefore, we define the "ratio of self-commits (commits authored by maintainers or committers) reviewed by others" as an indicator of the strictness of the review for the patches contributed by contributors with commit rights.
$$Q_2 = \frac{\#p_1}{\#p_2} \tag{10}$$
$$p_1,\ p_2 \in Patch,\ author_{p_1},\ author_{p_2} \in Mntr,$$
$$\{author_{p_1}\} \neq Reviewer_{p_1}$$

III. Results

RQ1: Does the multiple-committer model truly work?

We study the i915 subsystem and measure the effect of the multiple-committer model, based on the ten metrics described in Section II-C3. Fig. 3-8 shows the values of the different metrics for the i915 subsystem over time. Since we know that the i915 subsystem started to use 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 from October 2015 to reduce noise. To determine whether these two groups of data were significantly different, we utilized t-test. With the exception of file complexity (M8), the values of all of the metrics before the model was adopted exhibited significant differences relative to the values after the model was adopted. This indicates that the effects of using the multiple-committer model are quite conspicuous. The results are shown in Table I.

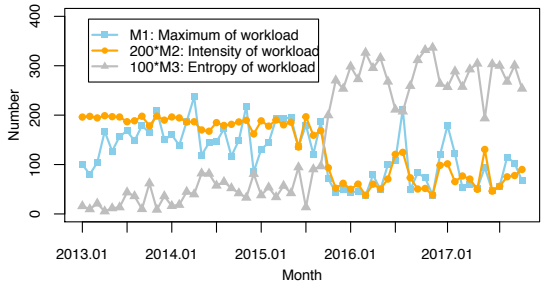

Fig. 3. Throughput of i915 Subsystem

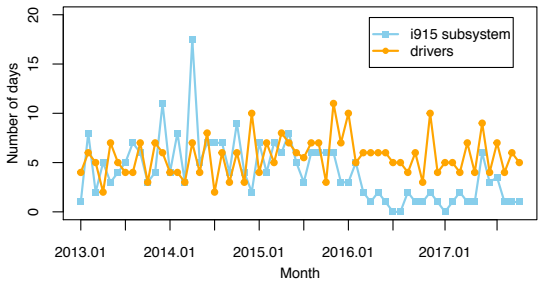

Fig. 4. Latency of i915 Subsystem

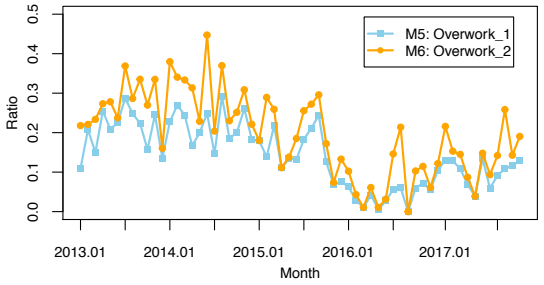

Fig. 5. Overwork of i915 Subsystem

Throughput. As shown in Fig. 3, though the number of commits (i.e., patches accepted into the mainline repository) in this subsystem has been increasing, the number of patches reviewed by the busiest maintainers has dropped significantly, especially in October 2015. Before the multiple-committer model was adopted, the busiest maintainer was responsible for over 90% of the workload (see Table I), but after the adoption

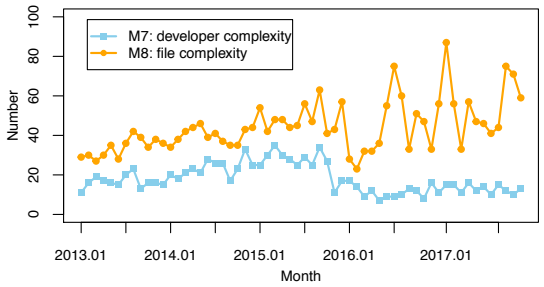

Fig. 6. Complexity of i915 Subsystem

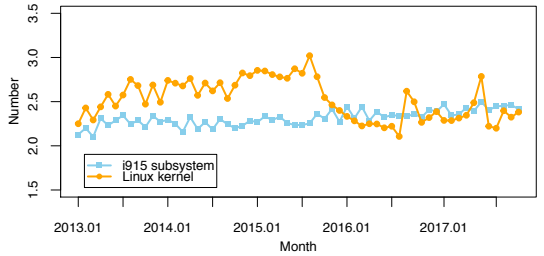

Fig. 7. Different Reviewers Per Patch of i915 Subsystem

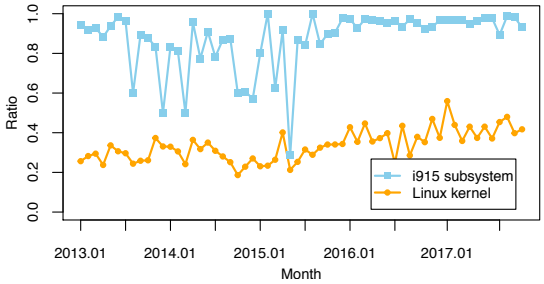

Fig. 8. Ratio of Self-Commits Reviewed by Others of i915 Subsystem

of the new model, this value decreased to approximately 30%. M3 reflects the degree of the dispersion of the work, which indicates that the workload gradually distributes evenly among the maintainers and committers. This suggests that the new workflow can not only reduce the maintainers' maximal workload but can also reduce the risk of a single point of failure.

Latency. When the maintainer is overloaded, s/he is unlikely to review patches in a timely manner, which may delay the merging of patches with the mainline repository. The latency is shown in Fig. 4. After the new workflow as adopted, the latency decreased from approximately five days to one day. Compared to the median latency of maintainers of all drivers, the decline in the latency of the i915 subsystem is apparent.

Overwork. As shown in Fig. 5, after the multiple-committer model was adopted, the ratio of work completed outside the working hours is reduced from 0.20 to 0.07 for M5 (0.27 to 0.11 for M6). Although the ratios appear to increase slightly in recent years, which may be a result of the increasing number of patches, they are still lower than the values before adoption of the new model.

Complexity. We measure the change in complexity based on the maximum number of developers/files touched by the maintainer. As shown in Fig. 6, developer complexity (M7) appears to be halved after adoption of the new model. There is no significant difference in file complexity (M8) after adoption of the new model (see Table I). This means that the effect of

TABLE I
STATISTIC RESULTS OF THE TEN METRICS OF I915 SUBSYSTEM

Metrics	Throughput			Latency	Overwork		Complexity		Quality	
	M1	*M2*	*M3*	*M4*	*M5*	*M6*	*M7*	*M8*	*M9*	*M10*
Mean	154.76, 79.92	0.92, 0.35	0.43, 2.79	4.88, 1.94	0.20, 0.07	0.27, 0.11	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	0.20, 0.06	0.27, 0.11	23.00, 12.00	39.00, 47.00	2.26, 2.39	0.87, 0.97
P-value	7.86E-09	3.54E-21	1.44E-29	1.95E-07	1.23E-15	4.08E-12	1.09E-10	0.015	6.23E-11	1.69E-05

In each cell, the fist number is the mean/median of each metric per month when the general workflow is used, and the second number is the mean/median after multiple-committer model is used.

the new model on mitigating the complexity of the review network is significant, but the effect on the complexity of the review work itself is not obvious.

Quality assurance. As shown in Table I, both M9 and M10 are larger after adoption of the model. We also compared the i915 subsystem to the overall Linux kernel, as shown in Fig. 7 and Fig. 8. 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. This finding indicates that the subsystem has a strict review process for patches while giving commit rights to more people (i.e., committers).

To further explore how the model works when more committers are introduced, we fitted regression models with the response being maintainer's burden (represented by M1–M10) and the number of committers (*#Cmtr*) being the predictor. Because the burden on the maintainers is likely to be affected by the number of patches (*#Patch*) and maintainers (*#Mntr*), we included these predictors in the models. Each observation represents a month from January 2013 to January 2018. We log-transformed skewed variables to satisfy the assumptions of the model. The final regression equation is:

$$(ln)(M1...M10) \sim ln(\#Cmtr) + ln(\#Patch) + ln(\#Mntr) \quad (11)$$

The results are presented in Table II. In seven of the ten models, the adjusted R^2 was greater than 0.5, while in the remaining three, the R^2 was greater than 0.3. With the exception of file difficulty (M8), all of the metrics were significantly associated with #Cmtr (p≈0). This suggests that when more committers are introduced, the workload of the busiest maintainer, as well as the throughput (M2), the latency of patch review, the overwork, and the task complexity, are reduced. If we neglect *#Patch* and/or *#Mntr*, the effects of *#Cmtr* are undiminished. These results are consistent with our previous analysis.

In summary, we quantified the effect of the multiple-committer model and found that it works well. The model appears to effectively reduce the burden on the busiest maintainer as well as reduce the ratio of overwork for the subsystem. It reduces the time needed for patch review and simplifies the working relationship between maintainers (committers) and authors. The quality of patches is guaranteed by the strict review mechanism. The review workload is shared by multiple committers, which can reduce the risk of a single point of failure.

TABLE II
RESULTS OF MODELING M1–M10

Metric	ln(#Cmtr)		ln(#Patch)		ln(#Mntr)	
	Est	*Std.Err*	*Est*	*Std.Err*	*Est*	*Std.Err*
M1	-0.10	0.07	0.77	0.06	-0.08	0.06
M2	-0.92	0.06	0.14	0.06	-0.17	0.05
M3	0.92	0.04	-0.09	0.04	0.19	0.04
M4	-0.64	0.13	0.02	0.13	0.07	0.12
M5	-0.89	0.09	0.23	0.09	-0.07	0.08
M6	-0.88	0.10	0.33	0.10	-0.06	0.09
M7	-0.91	0.10	0.17	0.10	0.36	0.09
M8	-0.11	0.12	0.52	0.12	0.36	0.11
M9	0.57	0.12	0.07	0.11	0.19	0.10
M10	0.60	0.14	-0.12	0.14	-0.12	0.13

The data in the blue cells indicate p-value $< .001$.

RQ2: Which subsystems are suitable for this model?

Through the above analysis, we find that the multiple-committer model has been working effectively for the i915 subsystem. To implement this new model in more subsystems, it is necessary to clarify its target audience. By investigating the workflow of the multiple-committer model and the characteristics of the i915 subsystem, we recommend that a subsystem check to see if it meets the following criteria to help decide whether it should adopt the new model.

1) Overloaded maintainer. The primary advantage of the multiple-committer model is that it can reduce the burden on maintainers. Therefore, subsystems whose maintainers are overloaded should consider adopting the model. We calculated the workload of the busiest maintainers 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 maintainers, whether considering all drivers or the whole kernel. In addition to the comparison with other maintainers, the following phenomena may also indicate that the maintainers are overloaded: difficulty in finding time to take a break, a significant increase in the latency of review or in the ratio of work completed outside of the working time.

2) Sharing code ownership. Subsystems that share code ownership may well fit this model due to the needs for coordination. In general, developers modify private working copies of the system to ensure stability during programming. However, this prevents them from knowing what their coworkers are doing which may affect private work [29]. A maintainer from the i915 subsystem indicated that conflicts can easily occur if multiple contributors frequently modify the same code file. For each month, we calculated the average number of unique contributors who modified the same file in i915 subsystem. We found that on average, 2.65 contributors modify a file per month versus only 1.31 and 1.29 for the overall drivers and the Linux kernel, respectively. Considering that the possibility of conflict is related to the numbers of

patches submitted by different contributors for the same file, we draw on the idea of code ownership [30] to characterize the possibility of conflict. Note that, if most modifications of the modifications to a file are performed by the same contributor, the risk of conflict is lower than that of files with more disperse contributions [31]. Code ownership is defined as $Ownership_{i,f} = \frac{\#patch_{i,f}}{\#patch_f}$, which represents the proportion of modifications that contributor i makes to file f. Then, we define the code ownership of the subsystem in Equation 12, where N is the number of contributors that modified file f, and F represents the number of files modified for the subsystem.

$$C\_O = \frac{\sum_{f=1}^{F} max_i(Ownership_{i,f})}{F},\ i \in [1, N] \qquad (12)$$

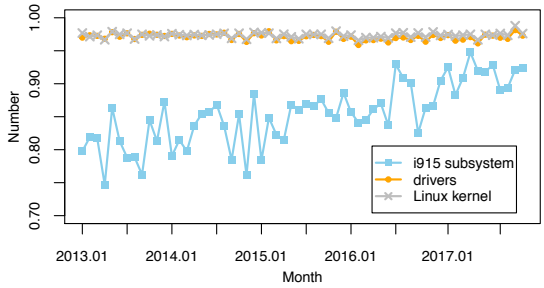

Fig. 9. Code Ownership of the Subsystems

As shown in Fig. 9, the code ownership of the i915 subsystem is lower than that of the overall drivers and the Linux kernel, especially before adoption of the new model. The smaller code ownership implies a greater possibility of conflict. As proposed by Bird et al. [29], awareness of others' activities in a timely fashion may help developers detect conflicts earlier. The general workflow requires that each maintainer manage her/his own repository. However, in the multiple-committer model, it is not forced to split the subsystem into several tiny subsystems. A maintainer of the i915 subsystem said that, sometimes, this split was not necessary, and that doing so may cause high latency. In the new model, each committer can directly submit patches to the same repository, which ensures that all of the contributors can obtain the status of the repository in a timely manner. A subsystem can determine whether or not to adopt this model by comparing its own code ownership to that of similar subsystems, by observing how the ownership changes over time, or by directly observing the occurrence of merge conflicts.

3) Sharing affiliations. When contributors have shared affiliations, it is easier for maintainers to find candidate committers, primarily for the following reasons. Some subsystems have multiple hardware vendors, which are usually competitors.[3] Modifications by a certain vendor may not be appropriate for other vendors, because their commercial objectives/needs and hardware are different. In this case, maintainers must act as gatekeepers for balancing the interests of all parties. For a regular contributor, thinking about the generality of the design is difficult (or not one of their priorities). This issue was confirmed by a maintainer from the "INFINIBAND SUBSYSTEM" during an interview. The combination of contributors of i915 subsystem avoids this issue because nearly all contributions are from its only vendor: Intel. We calculated the top three (the forth only contributes less than 2.8% of patches) email address domains belonging to authors, based on their patch contributions, in the i915 subsystem. We compared the top three domains from the i915 subsystem to those of other subsystems from three other modules.[10] [11] As shown in Table III, the top three domain names in the i915 subsystem all belonged to Intel and contributed over 80% of the patches. On the other hand, the affiliations of contributors from other subsystems were more complex. Contributors sharing affiliations can also facilitate communication and coordination, and maintainers are more likely to find committers they can trust. Other subsystems intending to use this model should consider whether the candidate committers can represent the common interests of different stakeholders – a concern made easier by shared affiliations.

TABLE III
TOP THREE AUTHORS' DOMAINS OF EMAIL ADDRESSES AND THEIR CONTRIBUTION PROPORTIONS

i915	mm	kernel	net
intel.com 40.2%	gamil.com 9.2%	linux.vnet.ibm.com 8.2%	gamil.com 15.6%
chris-wilson.co.uk 29.2%	linux.intel.com 8.5%	linutronix.de 8.0%	google.com 10.1%
linux.intel.com 21.7%	suse.com 7.7%	kernel.org 7.9%	redhat.com 6.9%

4) Stable core team. Committers must have extensive development experience [32], so they are usually chosen from the ranks of core developers. If turnover of core developers is high, it is difficult to find trusted contributors who can serve as committers. We define three metrics to characterize the stability of a subsystem. First, we define long-term contributors [33] as contributors who contribute to more than three releases.[12] The ratio of the long-term contributors (R_LTC) can reflect the stability of the team. Second, we define core contributors as the contributors whose contributions (#patches) are ranked in the top 10% of a release, which is borrowed from the idea that 80% of the work is accomplished by only 10% of the maintainers of the drivers [15]. We define the following two metrics to describe the stability of the core team. 1) retention time (R_T), which represents the median number of releases that the core contributors participate in (the contribution of at least one release must be ranked in the top 10%). 2) active time (A_T), which represents the median number of releases that the core contributors actively participate in (the developer's contribution to each release must be ranked in the top 10%). Based on these metrics, we evaluate the stability of the team for the i915 subsystem and compare it to that of the overall drivers and other subsystems from three core modules. We chose the development data from October 2015 to January 2018, which included 16 releases[11]. To reduce noise, we removed contributors who only participated in the first release or the last release.

[10] mm/kernel/net are the core modules as reflected by the folder structure of the Linux kernel. We choose these modules instead of three subsystems (for a total of more than 1500 subsystems), because modules are more representative.

[11] The results are based on the number of patches submitted from October 2015 to January 2018. The elimination of earlier data is to reduce noise.

[12] A new major kernel release happens every two or three months.

TABLE IV
STABILITY OF THE TEAM

subsystem	R_LTC	R_T	A_T
i915	0.35	11.60, 14.50	4.18, 3.00
drivers	0.34	9.05, 9.00	3.06, 2.00
mm	0.27	8.50, 9.00	2.44, 1.00
kernel	0.28	8.26 9.00	2.87 1.00
net	0.26	9.20, 9.00	3.30, 2.00

For the columns have two values, the first one is the mean and the other is the median.

As shown in Table IV, the stability of the contributors of the i915 subsystem is very similar to that of the overall drivers but is higher than that of other subsystems. This finding may be observed because the drivers are usually managed by companies, a process that is relatively stable relative to subsystems with more volunteers. The core team of the i915 subsystem is more persistent than that of other subsystems, reflected in R_T and A_T, which ensures trust between maintainer(s) and committers. A subsystem may calculate the stability of its current core team based on this set of metrics and compare itself to other subsystems to evaluate whether use of the model is appropriate.

5) Supporting continuous integration (CI). This factor helps the subsystem to adapt to fast changes and better ensure quality of patches. In the multiple-committer model, changes to repository are fast, which makes it nearly impossible to roll back. For the drivers that support different types/versions of hardware, the speed of iteration makes it particularly challenging to ensure the quality and robustness of patches on various platforms. In an interview, one maintainer indicated that a prerequisite for the new model is the ability to provide a wide range of hardware and bandwidth for automatic testing. For the i915 subsystem, Intel provides a CI system that consists of multiple Intel machines spanning as many generations and display types as possible.[13] Strict rules are enforced for CI, e.g., patch series must pass IGT Basic Acceptance Tests (BAT) on all of the CI machines without causing regressions.[14] Contributors can obtain the build results in real time.[15] Considering that this factor is essential but requires a large investment, we would say that the subsystems that have been supporting CI are more suited to the new model.

In summary, if a subsystem with overloaded maintainer intends to employ the multiple-committer model, it is essential to have candidate committers and ensure quality of patches. The following features provide insights: sharing code ownership and affiliations, having a stable core team, and supporting CI.

RQ3: What practices should be followed in applying the model?

The multiple-committer model delegates commit rights to a number of regular contributors, who are called committers. They can directly modify the subsystem repository. Compared to the traditional workflow, which requires all patches to be reviewed by the maintainers, the new model may increase the risk of introducing bad patches. Worse still, because a large number of patches is submitted to the same tree (repository), it is nearly impossible to rebase said tree. Therefore, the subsystem must follow certain practices to reduce these risks. To understand which practices ought to be followed when applying the model, we read the relevant online documentation belonging to the i915 subsystem. We also interviewed two maintainers of the subsystem to obtain more information. The results of our findings are as follows.

1) Sufficient precommit testing. Good testing is crucial to this model because there is almost no way to remove embarrassing mistakes. Such mistakes must be avoided in the first place – this requires sufficient precommit testing to ensure that obscure corner cases do not break functionality [23]. In addition to providing abundant hardware resources, the community of the i915 subsystem sets up detailed testing requirements [34]. For example, for the coverage of test, it requires that tests must fully cover user-space interfaces. The subsystem also does pre-merge testing of patch series sent by contributors. This allows verification, on a multitude of platforms, that the changes strictly improve the state of Linux. The results are automatically sent to contributors by emails. During an interview, one maintainer especially pointed out that it is important to have a CI system integrated with the bug tracking system.

2) Strict review process. The multiple-committer model has strict requirements for patch quality assurance because it is difficult to rebase the subsystem's tree. It requires that committers be confident in the patches they push, proportional to the impact of those patches.[14] 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 said, "*Having at least two people looking to the code and agreeing on the approach drastically reduces the risk of bugs and regressions.*" As shown in Fig. 8, 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, unresolved or conflicting feedback, management or peer pressure, or anything else), it is best to defer to the maintainers.

3) Necessary documentation. Open source documentation is a crucial part of open source management and is essential to growing and sustaining the community [35]. This is especially true for communities that quickly transition to a new workflow, which may cause contributors to be confused about the specifics of the new requirements. The i915 subsystem has its own website,[14] which provides a set of articles documenting the necessary information. These documents provide an overview of the development process, guidelines for committers and contributors, and an introduction to the tools, to name a few things. The documentation clearly outlines with text the responsibilities of committers, which is more effective than oral communication.

4) Applying tools to automate and simplify work as much as possible. A committer must review others' patches,

[13]https://intel-gfx-ci.01.org/hardware.html

[14]https://01.org/linuxgraphics/gfx-docs/maintainer-tools/drm-intel.html

[15]https://patchwork.freedesktop.org/project/intel-gfx/series/?ordering=-last_updated

communicate with the authors, and coordinate with multiple people, which takes considerable time and effort. Therefore, it is necessary to apply development tools to simplify the work of committers. For example, the Linux kernel drm subsystem (drm-intel and drm-misc Git repositories) employs "dim" to provide rich functionality to simplify the maintainers' (committers') work. It can help maintainers automatically test patches and it provides the options for error filtering. It also conveniently allow one to manage branches, view branches, and send the pull requests to the upstream branch. In an interview, a maintainer pointed out that "dim" is one of the key factors in the effective operation of the multiple-committer model. Using tools to avoid rather than correct errors is also a key to the model. When someone makes a mistake, a check should be put into the tools to prevent the mistake from occurring again, if possible [23].

In summary, if a subsystem plans to use the multiple-committer model, the following practices should be implemented, at the very least: sufficient precommit testing, strict review process, necessary documentation, and application of tools to automate and simplify work as much as possible.

IV. Validation

We interviewed two maintainers of i915 subsystem and four maintainers from three other subsystems via email to validate the results of RQ1 to RQ3.

To validate the results of RQ1, we conducted an interview with a maintainer of the i915 subsystem. We showed him the performance of the i915 subsystem, as measured by M1 to M10. He said that the results matched his experience with working on this subsystem over the past six years. Appropriate modifications were made to several metrics based on his suggestions, e.g., the fact that latency represents time between submission of the final version of a patch and merging of the patch, which, previously, we had not explicitly defined.

For RQ2, our aim is to validate the effectiveness of the features that we propose to screen out the subsystems that are suitable to adopt the new model. Based on the Linux kernel's commit history for the most recent year, we selected ten subsystems which are considered suitable (by the features) to apply the new model but currently are not using it, and sent emails to their maintainers asking for their opinions about it. In particular, we ranked all of the subsystems based on the ranking of their busiest maintainers (Eq. 1) and the ranking of code ownership (Eq. 12). We also considered contributors' affiliations. Because the following reasons, we did not consider the remaining two features. 1) "stable core team" is similar to "sharing affiliations" (they both suggest "having candidate committers"), and even without this feature, subsystems may try this model. 2) "supporting CI" is hard to evaluate unless the subsystem explicitly explains it somewhere. In the emails, we first introduced the multiple-committer model (with quantified effect on i915), and asked the maintainers whether their subsystems were willing to try it. We received four responses, three expressed interests in the model. The maintainer of "RADEON and AMDGPU DRM DRIVERS" clearly indicated that they were currently attempting to use the new model, suggesting that our metrics are effective for locating suitable subsystems. The maintainer of the "MELLANOX MLX5 IB driver" said, "*We had tried this model for a few months, but finally gave up because the patch quality was difficult to guarantee*", which demonstrates the importance of sufficient testing and strict review facilitate adoption of the new model. Two maintainers of "INFINIBAND SUBSYSTEM" showed interest in this model (the busiest maintainer forwarded the email to another maintainer), stating "*It's possible we could do something like that*", but they have concerns regarding the candidates. Because their contributors belong to mainly two competitors, Mellanox and Intel, they worried that, if the committers came from one company, it could be harmful to the other. They also worried that it was difficult to find enough reviewers. Their concerns exactly prove the effectiveness of the features (sharing affiliations and stable core team) we recommended. Although we thought that "INTEL ETHERNET DRIVERS" may be suitable to adopt the model, its maintainer did not believe himself to be overloaded at present. Researchers find that innovation resistance is determined by habit toward an existing practice and perceived risks associated with the innovation [36]. After all, the risks of applying this model requires further exploration and the maintainers are accustomed to the current workflow, so they may be willing to circumvent this risk by virtue of their prominence or dedication.

We designed a five-point Likert scale to request the maintainer of the i915 subsystem to evaluate the importance of the practices obtained in RQ3. He said, "*I think you nailed very well. All items there are very important (5 points)*," which confirmed our results.

V. Limitations

We identified the review workload of the maintainers by looking at the "signed-off-by" field in the commit messages, as well as the MAINTAINERS file, which enables us to confirm that the individual named by the signed-off-by field is indeed a maintainer. Although this approach is better than simply using the "committer" or "signed-off-by" fields, some problems are difficult to avoid, e.g., individuals who do not wish their names to appear in the MAINTAINERS file are excluded. Because "signed-off-by" does not contain the information of time, we calculated the review latency by the difference in time between the "commit date" and the "author date". 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. For some metrics, there may be problems in some cases. For example, "overwork" may not precisely reflect the additional workload of contributors who prefer to distribute the work over a period of more than eight hours. However, we focus on the change of overwork over time. Considering that the working habits of the contributors are relatively stable, the metrics may, to some extent, reflect changes in work pressure. 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 was excluded [37]. Reviewing and rejecting these patches also consumes maintainers' time. However, according to Zhou et al. [15], "It is easier for the maintainer to not accept the code at all. To accept it, it takes time to review it, apply it, send it on up the development chain, handle problems of it, accept responsibility for the patch, 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 internal validity, we interviewed Linux kernel maintainers and inspected various online resources. For example, we attained a reasonable understanding of the main burden on maintainers by reading relevant online documentation and by communicating with the maintainers. By exploring the use cases for the multiple-committer model, we also confirmed our findings based on the feedback from the maintainers of the four subsystems we selected.

Threats to external validity mainly from two phenomena. 1) Studying only a single subsystem, the i915 subsystem, with regard to the multiple-committer model limits external validity. On the one hand, to our knowledge, there are only two subsystems, the i915 subsystem and drm-misc subsystem in the Linux kernel, that currently use this model. The i915 subsystem was the first to develop and utilize the model. However, in the drm-misc subsystem, the number of new maintainers is larger than the number of committers. Therefore, the effect of applying the multiple-committer model is less obvious. On the other hand, it is exactly because there are only a handful of subsystems attempting to use the new workflow that studying it is necessary and timely. 2) The uniqueness of the Linux kernel limits external validity. As a prominent FLOSS project, the Linux kernel has many unique practices (as extensively studied by many researches) that have been referenced and used by many other projects [38]. In our study, we found that FLOSS projects that emulated the traditional kernel review process also encountered the problem of pressure on maintainers, e.g., the dpdk project.[3] The implication of delegating commit rights is applicable to all FLOSS projects that share the similar process. Therefore, our framework for quantifying the maintainers' review burden and our analysis of the multiple-committer model may benefit both the subsystems of the Linux kernel and other projects encountering the similar problems.

VI. Related Work

Software is developed in a dynamic context where team structure, requirements, and processes evolve together with the product [39]. How to maintain and sustain FLOSS projects is an urgent and timely issue for a number of communities. Many studies have focused on evolutionary aspects of open source development in answer to long-term sustainability and viability concerns of community-based software projects [40]. Examples of such researches include collecting experiences and building theories in terms of planning, process improvement, community involvement and software maintenance [41, 42, 43]. One of well-established theories of software evolution is Lehmans law [44], which conducted studies in the context of FLOSS to assess evolutionary and quality characteristics, e.g., survivability, growth potential. It has been widely observed by others [45, 46, 47].

The successful evolution of FLOSS projects depends on the co-evolution of the community (both developers and users) and the software [48]. Related studies in this track identified that the number of contributors grows with the product size [49]. Wang et al. also confirmed this phenomenon [50], with evidence that the increase of introduced packages and reported bugs are highly coherent with the increase of contributors and active users, respectively. Capiluppi et al. found that the increase in documentation and modularization levels are obliged to rising number of developers and their contribution in a project [51]. However, the studies about the adaption of the existing workflow are not common. Zhou et al. [15] studied the scalability of the Linux kernel maintainers and found that assigning multiple maintainers to a file yields only a power of 1/2 increase in productivity. Yet they do not provide solutions to adjust the current workflow to adapt the project. In this study, we try to bridge this gap by investigating a new workflow – multiple-committer model, which can bring insights for the co-evolution of the community (workflow) and the product, therefore help the FLOSS ecosystem survive the constantly changing environment and sustain.

VII. Conclusions

Increasing numbers of contributors to the Linux kernel have concerns regarding whether the development process (workflow) can handle the current mass of patches and even more patches in the future. While the workflow gives maintainers many rights, it also places significant pressure on them, which bears the risk of a single point of failure. We investigated the effect of a new model, the multiple-committer model, which is currently used by the i915 subsystem to alleviate the burden on its maintainers. We find that this model can reduce the proportion of the review workload of the busiest maintainer, reduce the review latency and maintainers' overwork, and simplify the review network. The review process is strictly enforced to guarantee the quality of patches. Subsystems with usage demands, mechanisms for ensuring patch quality, and candidate committers are suitable for the model. Sufficient precommit testing, a strict review process, necessary documentation, and the use of automatic tools are important practices to follow when applying the new model. Our proposed framework to quantify maintainers' review workload may lead to a better understanding of the factors that impede rapidly growing projects, ultimately making them more sustainable and reduce the risk of project failures. Knowledge of the use cases for the multiple-committer model and the key factors that make it effective can help other subsystems extract valuable information and optimize their workflow to achieve a more efficient review process and sustain themselves in a constantly changing, complicated environment.

REFERENCES

[1] D. Harhoff, J. Henkel, and E. Von Hippel, "Profiting from voluntary information spillovers: how users benefit by freely revealing their innovations," *Research policy*, vol. 32, no. 10, pp. 1753–1769, 2003.

[2] J. West and S. Gallagher, "Challenges of open innovation: the paradox of firm investment in open-source software," *R&d Management*, vol. 36, no. 3, pp. 319–331, 2006.

[3] A. Mockus, R. T. Fielding, and J. Herbsleb, "A case study of open source software development: The apache server," in *International Conference on Software Engineering*, 2000, pp. 263–272.

[4] L. Torvalds and D. Diamond, "Just for fun: The story of an accidental revolutionary," *Harperbusiness*, vol. 238, no. 6-7, p. S87, 2001.

[5] N. Ducheneaut, "Socialization in an open source software community: A socio-technical analysis," *Computer Supported Cooperative Work*, vol. 14, no. 4, pp. 323–368, 2005.

[6] G. V. Krogh and E. V. Hippel, "Special issue on open source software development," *Research Policy*, vol. 32, no. 7, pp. 1149–1157, 2003.

[7] A. Bonaccorsi and C. Rossi, "Why open source software can succeed," *Research Policy*, vol. 32, no. 7, pp. 1243–1258, 2002.

[8] M. Fink, *Business and Economics of Linux and Open Source*. Prentice Hall PTR,, 2003.

[9] M. Kerrisk, "Ks2012: Improving the maintainer model," https://lwn.net/Articles/514893/, accessed August, 2018.

[10] R. E. Cole, "From a firm-based to a community-based model of knowledge creation: The case of the linux kernel development," *Organization Science*, vol. 14, no. 6, pp. 633–649, 2003.

[11] G. Kroah-Hartman, "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, accessed August, 2018.

[12] S. Sarah, "Linux kernel introduction," https://www.slideshare.net/saharabeara/linux-kernel-introduction, accessed August, 2018.

[13] D. Vetter, "Vetter: Linux kernel maintainer statistics," https://lwn.net/Articles/752563/, accessed August, 2018.

[14] J. Corbet, "Group maintainership models," https://lwn.net/Articles/705228/, accessed August, 2018.

[15] M. Zhou, Q. Chen, A. Mockus, and F. Wu, "On the scalability of linux kernel maintainers' work," in *Joint Meeting*, 2017, pp. 27–37.

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

[17] D. Vetter, "Maintainers don't scale," https://blog.ffwll.ch/2017/01/maintainers-dont-scale.html, accessed August, 2018.

[18] ——, "Linux kernel maintainer statistics," https://blog.ffwll.ch/2018/04/maintainer-statistics.html, accessed August, 2018.

[19] G. Kroah-Hartman, *Linux Kernel in a Nutshell: A Desktop Quick Reference*. " O'Reilly Media, Inc.", 2006.

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

[21] T. S. Rappaport, "Wireless communications–principles and practice, (the book end)," *Microwave Journal*, vol. 45, no. 12, pp. 128–129, 2002.

[22] C. E. Shannon, W. Weaver, B. Hajek, and R. E. Blahut, "The mathematical theory of communication," *Physics Today*, vol. 3, no. 9, pp. 31–32, 1950.

[23] J. Corbet, "On linux kernel maintainer scalability," https://lwn.net/Articles/703005/, accessed August, 2018.

[24] R. Mishra and A. Sureka, "Mining peer code review system for computing effort and contribution metrics for patch reviewers," in *Mining Unstructured Data*, 2014, pp. 11–15.

[25] L. Macleod, M. Greiler, M. A. Storey, C. Bird, and J. Czerwonka, "Code reviewing in the trenches: Understanding challenges and best practices," *IEEE Software*, vol. PP, no. 99, pp. 1–1, 2017.

[26] J. Zhu, M. Zhou, and A. Mockus, "Effectiveness of code contribution: from patch-based to pull-request-based tools," in *ACM Sigsoft International Symposium on Foundations of Software Engineering*, 2016, pp. 871–882.

[27] P. C. Rigby, D. M. German, L. Cowen, and M. A. Storey, "Peer review on open-source software projects:parameters, statistical models, and theory," *Acm Transactions on Software Engineering & Methodology*, vol. 23, no. 4, pp. 1–33, 2014.

[28] S. Mcintosh, B. Adams, 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 *Working Conference on Mining Software Repositories*, 2014, pp. 192–201.

[29] M. L. Guimarães and A. R. Silva, "Improving early detection of software merge conflicts," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 342–352.

[30] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: examining the effects of ownership on software quality," in *ACM Sigsoft Symposium and the European Conference on Foundations of Software Engineering*, 2011, pp. 4–14.

[31] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Early detection of collaboration conflicts and risks," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1358–1375, 2013.

[32] W. Scacchi, "Free/open source software development," in *ESEC-FSE*, 2007, pp. 459–468.

[33] M. Zhou and A. Mockus, "What make long term contributors: willingness and opportunity in oss community," in *International Conference on Software Engineering*, 2012, pp. 518–528.

[34] D. Vetter, "Testing requirements for drm/i915 features and patches," https://blog.ffwll.ch/2013/11/testing-requirements-for-drmi915.html, accessed August, 2018.

[35] R. Jennifer, "Art of open source documentation," https://medium.com/capital-one-developers/art-of-open-source-documentation-5b8b3f5b0ab, accessed August, 2018.

[36] J. N. Sheth and W. H. Stellner, "Psychology of innovation resistance : the less developed concept (ldc) in diffusion research / bebr no. 622," vol. 4, 1981.

[37] B. Adams, B. Adams, and A. E. Hassan, *Continuously mining distributed version control systems: an empirical study of how Linux uses Git*. Kluwer Academic Publishers, 2016.

[38] T. Mens, M. Claes, P. Grosjean, and A. Serebrenik, *Studying Evolving Software Ecosystems based on Ecological Models*. Springer Berlin Heidelberg, 2014.

[39] K. Ngamkajornwiwat, D. Zhang, L. Zhou, R. Nolker *et al.*, "An exploratory study on the evolution of oss developer communities," in *hicss*. IEEE, 2008, p. 305.

[40] M. M. Syeed, I. Hammouda, and T. Systa, "Evolution of open source software projects: A systematic literature review," *Journal of Software*, vol. 8, no. 11, pp. 2815–2830, 2013.

[41] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, and I. Herraiz, "Evolution and growth in large libre software projects," pp. 165–174, 2005.

[42] C. K. Roy and J. R. Cordy, "Evaluating the evolution of small scale open source software systems," *Special Issue: Advances in Computer Science and Engineering*, vol. 123, 2006.

[43] Q. Hong, S. Kim, S. C. Cheung, and C. Bird, "Understanding a developer social network and its evolution," in *IEEE International Conference on Software Maintenance*, 2011, pp. 323–332.

[44] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution - the nineties view," in *Software Metrics Symposium, 1997. Proceedings., Fourth International*, 1997, pp. 20–32.

[45] A. Meneely and L. Williams, "Secure open source collaboration:an empirical study of linus' law," in *ACM Conference on Computer and Communications Security*, 2009, pp. 453–462.

[46] M. Wermelinger, Y. Yu, and A. Lozano, "Design principles in architectural evolution: A case study," in *IEEE International Conference on Software Maintenance*, 2008, pp. 396–405.

[47] L. Yu and A. Mishra, "An empirical study of lehman's law on software quality evolution." *Int. J. Software and Informatics*, vol. 7, no. 3, pp. 469–481, 2013.

[48] W. Scacchi, "Understanding open source software evolution: Applying, breaking, and rethinking the laws of software evolution," 2006.

[49] A. Capiluppi, "Models for the evolution of os projects," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, 2003, pp. 65–74.

[50] Y. Wang, D. Guo, and H. Shi, "Measuring the evolution of open source software systems with their communities," *Acm Sigsoft Software Engineering Notes*, vol. 32, no. 6, p. 7, 2007.

[51] A. Capiluppi, P. Lago, and M. Morisio, "Characteristics of open source projects," in *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*. IEEE, 2003, pp. 317–327.