Understanding Group Maintainership Model in the Linux Kernel Development

Eduardo Pinheiro[1]**,Paulo Meirelles**[1]

[1]FLOSS Software Competence Center
Institute of Mathematics and Statistics
University of São Paulo

{edupinheiro,paulormm}@ime.usp.br

Abstract. *Software development has evolved over decades, transitioning from traditional models such as the waterfall approach and the unified process to more flexible methodologies like agile methods and collaborative development strategies of Free/Libre/Open Source Software (FLOSS) projects. Alongside this trend, the global distribution of software development work has increased. This phenomenon is particularly evident in the development of FLOSS projects, where contributors from various regions worldwide collaborate asynchronously on projects. In this context, the organization of interactions among developers can significantly influence a project success or failure. An example is the Linux kernel community, which has been actively discussing the models and workload of project maintainers – a topic that has received limited attention in scientific literature. This study investigated the new maintenance methods used in the Linux kernel project. With over 30 years of development, the Linux kernel has become a benchmark for FLOSS development. We discuss how the maintainers' workload is addressed in academic literature and by practitioners in the Linux kernel community. To achieve this, we conducted a multivocal literature review to examine the evolution of maintenance models over the years.*

1. Introduction

The basic principle of the Free/Libre/Open Source Software (FLOSS) ecosystem is to promote user freedom, enabling anyone to use the software without prejudice or limitations through collaboration and an open development process. To be considered FLOSS, software must offer users the freedom to use, study, modify, and redistribute it without restrictions, ensuring that these liberties remain intact for future users. FLOSS development projects rely on publicly available source code, typically hosted in online repositories where developers and users can interact [Kon et al. 2011]. A formal and legal license for the source code, as defined by the Free Software Foundation (FSF) or the Open Source Initiative (OSI), is required.

The FLOSS ecosystem applies software engineering principles to software construction and also become a subject of study. However, there is a gap between academic studies on FLOSS and community practices, particularly a superficial understanding of the development model of the Linux kernel [Wen 2021] – the most famous and established FLOSS project. The evolution of the Linux kernel development occurred alongside the maturation of the FLOSS movement, significantly impacting its consolidation. The Linux kernel is a project with over three decades of history that maintains a robust development

ecosystem. Its characteristics have inspired various software development models – the most remembered was described by [Raymond 1999] in his essay "The Cathedral and The Bazaar." Raymond discussed his observations on the Linux kernel development and the lessons from applying some "Bazaar" practices to the *fetchmail*, his project. In summary, the goal was to disseminate the "open-source" as an apolitical and unbiased term to define the development model of "free software" projects. Thus, more companies became interested in the benefits of these practices, leading to the growth of the FLOSS ecosystem [Crowston et al. 2012] and influencing its development process [Wen 2021].

Since Raymond's essay, the Linux kernel project has grown in source code size, developer team size, and various markets [Wen 2021]. Moreover, there are different views [Fogel 2005, Rigby et al. 2014, Lindberg et al. 2014, Shaikh and Henfridsson 2017, Tan and Zhou 2022] and an apparent shallow understanding of the FLOSS development and the Linux kernel model. We should recognize that the Linux kernel is an example of an efficient contribution model involving a high level of collaboration among diverse contributors from around the world – for instance, nearly 2,000 developers contributed to version 6.3[1]. The development cycle involves substantial changes; based on email lists, the Linux kernel contribution model is theoretically accessible. However, the model is not well-documented, and many practices can only be learned from more experienced contributors or maintainers. Additionally, the workflow includes many procedures that could be automated or simplified to make the contribution process more harmonious and agile. In short, the efficiency and simplicity of the collaboration model are crucial for the project success. A complex or ineffective collaboration model can hinder the project progress and scalability, making it difficult for new contributors to join and effectively participate and reducing the rise of new maintainers.

Every FLOSS community has its communication and collaboration methods, which can vary within a community based on subparts of the project. The Linux kernel has multiple subsystems, each with its own dynamics. The interaction between members can shape the project progression. The Linux kernel maintainership model dictates how maintainers manage subsystem contributions. This model can vary between subsystems, with some maintainers adopting different strategies to manage contributions. The community observes recent strategies as potential ways to mitigate the maintainership challenges many subsystems face, which we can encapsulate by the phrase: "Too many lords, not enough stewards" [Edge 2018].

In this context, our research investigated how these challenges have affected the Linux community. We identified two different ways maintainers can organize their subsystems. First, the classical model typically has one or two maintainers who handle all the reviewing work. Second, the group maintainership model, categorized into three approaches: hands-off, delegation, and multiple-committer [Corbet 2016]. These new strategies are used in some Linux kernel subsystems and are observed as alternatives to sustain the maintainership process.

In this work, we characterized the group maintainership model and discussed how it is applied to Linux kernel subsystems. We analyzed the problems that lead project members to seek new maintainership approaches. We investigated how these new models

[1] `https://lwn.net/Articles/929582/`

differ from the existing ones and if there is a need for specific conditions to make them work. To conduct this study, we established the following questions to guide our research:

- **RQ1**: What are the characteristics of the subsystems communities implementing the group maintainership model? What differentiates them from other subsystems?
- **RQ2**: Which aspects of the group maintainership model differ from the traditional process of reviewing and accepting patches in the Linux kernel?

Our methodology involves a Multivocal Literature Review (MLR) integrating perspectives from a Systematic Mapping Study (SMS) and a Grey Literature Review (GLR). An MLR is a research method that incorporates a variety of sources beyond traditional academic publications. This approach includes "grey literature" such as developer blogs, forum discussions, and white papers. By integrating these diverse perspectives with academic research, an MLR provides a more comprehensive picture of the topic, reflecting state-of-the-art and state-of-practice insights [Garousi et al. 2019]. Our approach seeks an understanding of the Linux kernel contribution process, particularly emphasizing the role of maintainers.

2. The Linux kernel Project

The Linux kernel project is an extensive initiative comprising nearly 25 million lines of code and involving almost 2,000 developers from over 250 companies or independently working. Its development process diverges significantly from traditional proprietary methods, embracing a community-driven framework, a patch-oriented development process, and a time-based release model.

The kernel codebase is structured into subsystems such as the process scheduler, memory management, device driver infrastructure, networking, and filesystems. "A subsystem is a representation for a high-level portion of the kernel as a whole" [Corbet et al. 2005]. It can also be referenced as an abstraction, which refers to a specific part of the kernel that deals with a particular kernel function.

Usually, each subsystem has designated maintainers. A maintainer manages and accepts contributions into a specific subsystem code repository. Contributions are formatted and sent to the Linux kernel project as a patch. A patch is a text document describing the differences between two versions of a source tree. These contributions (or patches) are grouped according to a particular need or interest. This collection of patches is arranged in a tree. Moreover, each tree generally references a subsystem, which works as a place for a specific development.

Throughout its over three decades, Linux creator Linus Torvalds has repositioned the project. For instance, in 2007, Torvalds positioned itself against the adoption of GPLv3 [Torvalds 2007], created by the Free Software Foundation. In 2016, he stated that he was never concerned with following FLOSS methodology or policies [McManus 2016]. Linus just opened the code for feedback and created methods and tools (such as Git) that made it possible to shape development in the most comfortable possible way (for him). Finally, in a letter sent to the kernel project mailing list in 2018, Torvalds licenses himself briefly from coordinating Linux due to strong disagreements with the community regarding his way of conducting the project [Torvalds 2018]. Greg

Kroah-Hartman took over Torvalds' coordination responsibilities, and after a month, Torvalds returned to Linux development [Statt 2018]. At that time, the controversial code of conflicts was replaced by a new code of conduct, symbolizing a welcoming and inclusive community that cares about its developers.

In the contribution process and coordination, maintainers became fundamental, overseeing every development aspect, from code review and bug fixing to release management and community engagement. Maintainers often act as the gatekeepers of the project, deciding which contributions to accept or reject and ensuring that the project remains healthy and sustainable. They then became not only the most knowledgeable members of the community in the codebase but also mediators, mentors, and community leaders. They facilitated contributor collaboration, resolved conflicts, and upheld the project standards and values.

However, maintainers face numerous challenges, including maintaining project sustainability, managing community dynamics, and addressing security vulnerabilities. The sheer volume of contributions and patches can overwhelm maintainers, leading to burnout and fatigue. Additionally, maintainers must navigate legal and licensing issues, enforce code quality standards, and balance priorities and interests within their communities as companies and individuals with different goals and motivations contribute to the projects.

In this scenario, we studied the maintainership process in the Linux kernel community. We focused on the group maintainership model and investigated its application to Linux kernel subsystems [Corbet 2016]. Specifically, we examined how these new models differ from the existing ones and whether specific conditions are required for effective implementation.

3. Systematic Mapping Study

We conducted a Systematic Mapping Study (SMS) to understand better how much information the academic community has about the maintainership models used in the Linux kernel. We choose the major software engineering research publishers, searching their digital collections: ACM, Scopus, IEEE, and Springer.

From a resultant string [**Linux AND (kernel OR community OR subsystem) AND (maintainers OR maintainership)**], our search string returned 91 results, including all databases, and through a selection based on whether the title and abstract contained keywords such as maintainership, contribution process, maintainers, we ended up with 33 selected papers. After reading the abstract and verifying if the papers were related to our research by looking for key phrases that indicated that they were related to the contribution process or the maintainership model, we reduced our selection to 20 works[2]. Lastly, speed-reading through the text led us to reach 16 relevant papers, as some of the papers only marginally related to the topics researched and did not contribute relevant information. Then, after reading these papers, we also found some related work through a snowballing process; while these papers were not directly related to the topic, they helped give a better context to our research, so we ended up with 33 comprehensively read papers.

In our findings in the literature, we identified two models coexisting in the Linux

[2]Our replication package is available at `https://zenodo.org/records/13345129`

kernel community: the traditional model (identified as the single maintainer model) and the group maintainership model. The single maintainer model is the most common model in the Linux kernel community. Its characteristics are the hierarchical structure, where the maintainer is the single point of contact for the subsystem and is responsible for reviewing and accepting the patches and managing upstream patches. On the other hand, the group maintainership model is more decentralized, characterized by the maintainer having a more administrative role, and the review and acceptance of patches are done by a group of regular contributors with the right to commit to the subsystem. This model has been shown to reduce the maintainers' workload significantly.

From our mapping study, we understood how far academia has researched and what conclusions it has reached concerning contribution management in the Linux kernel. When going through the research related to the Linux kernel and FLOSS projects in general, a trend can be identified, and the most common topics that researchers focus on are the newcomers and the contributors. At the same time, maintainers are neglected by the academic eyes [Bettenburg and Hassan 2012].

As [Capiluppi 2003] pointed out, many FLOSS projects struggle with contribution management. The burden on maintainers increases as the project grows. For instance, the Linux kernel has grown from 10,239 lines of code in 1991 (version 0.01) to over 27.8 million lines of code in 2020 (version 5.5) [Tan and Zhou 2022], highlighting the importance and need to find solutions to successfully scale up the contribution process with the growth of the project.

3.1. Multiple-Committer Model

There is a demand to adapt the original workflow to the growing workload and to sustain the community. Exploring the repository of the Linux kernel, [Tan et al. 2020] has identified a new model for the contribution process, the multiple-committer model (MCM), where a **group of regular contributors for the subsystem is given commit rights**. The authors identified that the multiple committer model significantly reduced the maintainers' burden in the i915 (Intel DRM – Direct Rendering Manager subsystem – Drivers) module, which was first implemented in 2015. They pointed out that the reviewing process pressure, latency, and complexity are reduced when more committers are accepted. With that, we have a lower risk of the **maintainer being the single point of failure**.

A subsystem that is contemplating the adoption of the MCM should, according to [Tan et al. 2020], have the following prerequisites to apply the multiple committer model successfully:

- Sufficient pre-commit testing and continuous integration (CI) tools can be quite useful to achieve this;
- A strict review process that the committers can perform;
- The use of tools to simplify work and reduce errors, automated tests can benefit the review process.

Implementing the Multiple-Committer Model requires certain safeguards to ensure that expanding the pool of people who can apply patches does not compromise the quality of the project. Overloaded subsystems with trustworthy candidate committers are particularly suited for adopting this model. To effectively apply the MCM, several best practices have been identified in the academic literature [Tan et al. 2020].

First, it is crucial to train potential committers as early as possible and to encourage developers to make code review a routine part of their daily work. Setting strict but flexible requirements for the qualification of committers is also important. Communities can encourage developers to review code by having reviewers sign their names in commit messages, which records their contribution and tracks responsibility. Code review should be a key factor in evaluating individual contributions, and in projects with high technical thresholds, maintainers should establish procedures to identify and train potential committers early to avoid a shortage of capable individuals.

Trust between maintainers and committers is essential for the MCM to function effectively. Committers must earn the maintainers' trust by consistently performing quality work. Additionally, sufficient pre-commit testing is fundamental to maintaining the quality of accepted patches and minimizing the risk of introducing bugs or errors.

A clear and strict review process should be in place, ensuring that committers adhere to it and do not show favoritism towards patches from friends or coworkers. Utilizing tools that simplify the work and reduce errors is another best practice. For example, the i915 module (DRM subsystem) uses "dim" [3] to provide functionalities that make the workflow more straightforward for maintainers and committers. Finally, all discussions, decisions, and rules should be open and transparent, ensuring the process remains fair and accountable.

In short, this emerging model can potentially mitigate the previously mentioned problems. As seen in its implementation on the i915, it can reduce the maintainers' workload. At the same time, it reduces latency for patch response and increases the capacity to incorporate code. Still, to work, it needs to fulfill a series of pre-requirements that will provide some insurance so that the code quality will not drop due to less experienced people being able to apply patches.

4. Grey Literature Review

The selection of grey literature proved to be more challenging, primarily because we could not rely on robust search engines like those available in research publisher databases. Unlike our SMS, this process resulted in more extensive search results. However, by following the guidelines from [Wen et al. 2020], we established a robust method for identifying relevant material. We began by searching through the most pertinent repositories related to Linux Kernel development, such as Linux Weekly News (`lwn.net`) and the Linux Foundation (`linuxfoundation.org`). Additionally, we consulted blogs maintained by key persons like Simma Vetter (`blog.ffwll.ch`) and Greg Kroah-Hartman (`kroah.com/linux/`).

Since some sources lacked direct search engines, we initially relied on custom Google searches, focusing on results strictly from these sites. Subsequently, we broadened our search to include results from other sites, but only those containing the exact keywords we were targeting. This approach yielded a substantial number of findings. However, because the grey literature texts were generally shorter than those in the systematic mapping study, we could quickly filter them. We skimmed titles and briefly reviewed the context in the initial step, eliminating most irrelevant results. During the final step of

[3] `https://drm.pages.freedesktop.org/maintainer-tools/dim.html`

reading the full texts, we concluded our filtering process, identifying 19 relevant results[4].

As with the SMS, both single maintainer and group maintainership models are discussed in the grey literature, and the characteristics of these models are somewhat equal to the ones we found in the SMS. However, here we were able to identify that group maintainership models can have different approaches, such as the hands-off model, the delegation model, and the multiple-committer model, thus providing a more detailed view of how these models are implemented in the community, which was not available in the academic literature.

Both maintainers and developers use informal ways to publish text related to problems they encounter in their day-to-day experience in the kernel community. Therefore, the data in these media was crucial to highlight the state of the Kernel Linux community in a way that was not available in the academic format. [Tan et al. 2020] captured some of the discussions, but we could also extend our knowledge of the challenges that maintainers are dealing with more directly.

4.1. Challenges within the Community

Many members of the Linux kernel community have reported various issues that negatively impact project development and collaboration. A significant challenge is the presence of toxic individuals within the community. There is a widespread perception that too many members exhibit toxic behavior toward others. For instance, one community member observed that "pretty much every subsystem has its local toxic person." These individuals often possess specialized knowledge, making them irreplaceable and granting them undue influence over the rest of the community [Edge 2018].

Another critical problem is the lack of documentation, which poses a significant barrier for newcomers who struggle to contribute to the project due to the steep learning curve. This issue also threatens project continuity, as knowledge is not adequately shared among community members [Edge 2018]. Closely related to this is the problem of knowledge propagation. Tools, tests, and vital information are often kept by maintainers rather than shared, making these maintainers unreplaceable and further concentrating power within the project [Edge 2018].

The difficulty in enacting change is another issue, as even those who believe changes are necessary are often afraid to take action for fear of public shaming if something goes wrong. This fear leads to maintainers' more controlling approach to development, making it harder to implement new maintainership models [Edge 2018]. Furthermore, the succession process within the community lacks clarity. There is generally no succession plan for relinquishing the maintainer role, making finding new maintainers challenging. Some former maintainers suggest creating a vacuum might be an effective exit strategy, as it could force experienced developers to take on the role. However, this challenging process has resulted in maintainers struggling to keep up with the overall growth of the kernel. If current trends continue, many Linux kernel maintainers may become bottlenecks within their subsystems [Edge 2016, Vetter 2018].

The lack of proper review is another concern, with patches not being thoroughly reviewed before acceptance [Corbet 2013]. Additionally, there is an abuse of power,

[4]Our replication package is available at `https://zenodo.org/records/13345129`

where those in positions of authority do not hold themselves to the same standards as they do others. For example, self-commits by maintainers, where the patch author and committer are the same person, often go unreviewed [Vetter 2018]. Another worrying issue is the existence of maintainerless subsystems. There are 367 subsystems without a maintainer or where the maintainer has never been active in the Git history. Many of these subsystems may point to obsolete code that could be removed, such as drivers for devices no longer in use [Corbet 2021].

The workload of current maintainers in the Linux kernel community is becoming increasingly overwhelming, as they are responsible for reviewing and applying numerous patches, leaving them with little time to contribute their own. Greg Kroah-Hartman highlighted recurring issues with submitted patches that unnecessarily increase maintainers' workload, including incomplete patch series, patches arriving out of order, and those with email signatures marked as confidential, which are not acceptable in the open nature of kernel development. Other issues involve patches that do not adhere to the project code style, are created in the wrong directory, do not compile, or are sent to the wrong maintainer despite the availability of tools to help avoid such errors – overly large patches also present challenges, as incremental changes are more accessible to review.

Kroah-Hartman reported receiving 487 patches in just two weeks during what he described as a "calm two weeks," illustrating the intense workload maintainers face. This repetitive and sometimes frustrating work can lead to what Tim Bird calls the "Maintainer's Paradox," where maintainers are excited about new contributions but feel overwhelmed by the volume of patches requiring careful review and feedback. Trond Myklebust, the NFS client subsystem maintainer, described the multifaceted role of a maintainer as encompassing five key responsibilities: software architect, developer, patch reviewer, patch committer, and software maintainer. Wolfram Sang, the I2C subsystem maintainer, expressed concern that the number of maintainers is not keeping pace with the increasing volume of patches. Although no immediate collapse has been predicted, Sang's data shows that the problem is already present, with increasing latency in subsystems and the potential for accepting questionable patches. He forecasts that these issues will worsen over time. During the 3.0 to 3.10 release period, the number of patch authors rose by about 200, while the number of reviewers remained largely static, highlighting the growing gap between authors and reviewers, underscoring the scaling problem that maintainers face.

4.2. Group Maintainership Models

Some subsystems implement new models to handle the contribution process and mitigate several problems related to the kernel maintainership process. These models are intended to help maintainers not burn out, providing a better way to balance the workload. They can prevent issues with personal availability and are also a great way to develop new maintainers [Corbet 2016]. Table 1 shows the subsystems and the models they have applied.

However, introducing changes is quite challenging in the kernel community, primarily because of its size. Changing the maintainer's culture is proving to be a challenge that will be around in the kernel for a long time and maybe define if the project will keep growing or fail in the future [Corbet 2018]. We have identified a few approaches for these group maintainership models deployed in some subsystems. For instance, X86 Platform Drivers, ARM/ARM64 SoC, and the graphics subsystem (DRM) [Vetter 2017].

Subsystem	Maintainership model	Module
DRM	multiple committer	i915 graphics driver
ARM and ARM64 SoC	hands-off	SoC sub-architectures
Power Management	delegation	not specified
Media	multiple committer	not specified
X86 Platform Drivers	multiple committer	not specified

Table 1. Maintainership models by subsystem

Darren Hart, an X86 Platform Drivers subsystem maintainer, identified the **Hands-off** model, where maintainers manage a single repository using an IRC channel to "lock" the repository (to avoid concurrency problems). When changes are ready to be applied, they also keep a log of the changes made so the other maintainers can always see what was done. It is currently being used on the *arm-soc tree* [Corbet 2016].

Darren also explained the **Delegation** model, which is mainly deployed in subsystems that use patchwork [5] (patch management subsystem). The patchwork can delegate the handling of each patch to a specific maintainer, automatically working as the load balance for these maintainers. Some subsystems that are using patchwork are the Media and the Power-Management subsystems [Corbet 2016].

Vetter also explains that the **Multiple-committer model** differs from the models above as it introduces the concept of the committer, a developer with writing rights to the repository. They are not maintainers as they do not perform some of the "maintainers" tasks, such as externally communicating with other subsystems and taking the blame for mistakes. Still, they help review and accept the patch process. It is used in the i915 graphics driver (DRM subsystem) [Corbet 2016].

Vetter, through a series of scripts gathering data from the Linux kernel releases, shows that the Graphics subsystem, which has been using group maintainership models for some time now, does have a higher percentage of maintainers self-commits, 40% against below 30% found in the overall kernel [Vetter 2018]. However, it has a much higher rate of reviewed self-commits, getting to over 80% of patches being reviewed against 40% on the rest of the Linux kernel, suggesting that the model could incorporate more committers without increasing the number of unreviewed patches. This strategy also directly helps the process of training new maintainers, as more people participate in the process.

5. Concluding Remarks

Our multivocal literature review explored academic literature and materials published by Linux kernel practitioners, focusing on the growing problem of overworked maintainers and the inadequate scaling of the maintainership workforce to meet demand. We identified new approaches to contribution management in the Linux kernel, particularly the group maintainership model.

Our findings addressed RQ1 (*What are the characteristics of the subsystems communities implementing the group maintainership model? What differentiates them from*

[5] http://jk.ozlabs.org/projects/patchwork/

other subsystems?) by identifying key characteristics of subsystems that have successfully implemented group maintainership models, such as having recurrent contributors, sufficient pre-commit testing, and a well-defined contribution process. In response to RQ2 (*Which aspects of the group maintainership model differ from the traditional process of reviewing and accepting patches in the Linux kernel?*), we highlighted the new contribution processes and their differences from the single maintainer model.

The systematic mapping study revealed that the literature recognizes the maintainership workload issue. We identified a new maintainership model – the multiple-committer model – that aims to address this problem and support the project growth, suggesting that further research is necessary to understand its potential fully. Our grey literature review provided an understanding of the ongoing concern over the maintainers' workload. We also identified new maintainership approaches and the subsystems adopting these strategies.

Throughout our research, the DRM subsystem emerged as a prominent example of successfully implementing the group maintainership model within the Linux kernel. Our SMS and GLR provided evidence that the DRM subsystem had advanced in adopting a multiple-committer model. This model allows maintainers to take breaks from their duties without stalling development, reducing their workload and increasing the number of contributions to the driver. The Media subsystem is also in the process of adopting the multiple-committer model. Although this transition is ongoing, some of our findings about the requirements for this model have proven accurate, as the Media subsystem maintainers are waiting for a robust pre-commit testing infrastructure to be in place before granting regular contributors commit access.

Regarding the potential limitations and threats to the validity of our work, we may have missed relevant papers in our systematic mapping study, and searching for information in grey literature, such as other blog posts, forums, and mailing lists, posed challenges in identifying the most pertinent details. While we employed a methodical approach, some relevant information might have been overlooked.

Finally, the Linux kernel community constantly evolves; since workload is a recurrent problem, we understand that the maintainership model will continue to mature. Group maintainership seems to be a reasonable solution for the workload problem, but it is not a "silver bullet". In an extended version of this study, we can discuss how the group maintainership model improves the maintainership process, helping to reduce the maintainers' workload. We already have evidence that adopting the multiple-committer model reduces the workload of maintainers in subsystems, based on evidence from our multivocal literature review; moreover, we have data from community members (that we did explore in this paper because of the limit of pages). Also, more research is needed as more subsystems adopt group maintainership models and new challenges arise. We also recommend conducting more research to understand the challenges maintainers face and how the group maintainership approach will evolve.

Acknowledgments

This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq proc. 465446/2014-0, CAPES – Finance Code 001, and FAPESP procs. 14/50937-1 and 15/24485-9. We also thank the support of FAPESP proc. 2023/00811-0.

References

Bettenburg, N. and Hassan, A. E. (2012). Studying the impact of social interactions on software quality. *Empirical Software Engineering*, 18(2):375–431.

Capiluppi, A. (2003). Models for the evolution of os projects. pages 65–74.

Corbet, J. (2013). On saying "no".

Corbet, J. (2016). Group maintainership models.

Corbet, J. (2018). The code of conduct at the maintainers summit.

Corbet, J. (2021). Maintainers truth and fiction.

Corbet, J., Kroah-Hartman, G., and Rubini, A. (2005). *Linux device drivers*. O'Reilly.

Crowston, K., Wei, K., Howison, J., and Wiggins, A. (2012). Free/libre open-source software development. *ACM Computing Surveys*, 44(2):1–35.

Edge, J. (2016). On moving on from being a maintainer.

Edge, J. (2018). Too many lords, not enough stewards.

Fogel, K. (2005). *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, Inc.

Garousi, V., Felderer, M., and Mäntylä, M. V. (2019). Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106:101–121.

Kon, F., Meirelles, P., Lago, N., Terceiro, A., Chavez, C., and Mendonca, M. (2011). Free and open source software development and research: Opportunities for software engineering. In *2011 25th Brazilian Symposium on Software Engineering*, pages 82–91.

Lindberg, A., Xiao, X., and Lyytinen, K. (2014). Theorizing modes of open source software development. In *2014 47th Hawaii International Conference on System Sciences*, pages 4568–4577.

McManus, E. (2016). The quotable Linus Torvalds, live onstage at TED.

Raymond, E. (1999). The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49.

Rigby, P. C., German, D. M., Cowen, L., and Storey, M.-A. (2014). Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Trans. Softw. Eng. Methodol.*, 23(4).

Shaikh, M. and Henfridsson, O. (2017). Governing open source software through coordination processes. *Inf. Organ.*, 27(2):116–135.

Statt, N. (2018). Linus torvalds returns to linux development with new code of conduct in place.

Tan, X. and Zhou, M. (2022). Scaling open source software communities: Challenges and practices of decentralization. *IEEE Software*, 39(1):70–75.

Tan, X., Zhou, M., and Fitzgerald, B. (2020). Scaling open source communities: An empirical study of the linux kernel. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1222–1234, New York, NY, USA. Association for Computing Machinery.

Torvalds, L. (2007). Re: Dual-licensing linux kernel with GPL v2 and GPL v3.

Torvalds, L. (2018). Linux 4.19-rc4 released, an apology, and a maintainership note.

Vetter, D. (2017). Maintainers don't scale.

Vetter, D. (2018). Linux kernel maintainer statistics.

Wen, M., Leite, L., Kon, F., and Meirelles, P. (2020). Understanding floss through community publications: strategies for grey literature review. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '20, page 89–92, New York, NY, USA. Association for Computing Machinery.

Wen, M. S. R. (2021). *What happens when the bazaar grows: a comprehensive study on the contemporary Linux kernel development model*. PhD thesis.