Classifying Linux commits

Jesus M. Gonzalez-Barahona[1,*], Michel Maes[2], Abhishek Kumar[3], David Moreno-Lumbreras[1] and Gregorio Robles[1]

[1]Escuela de Ingeniería de Fuenlabrada, Universidad Rey Juan Carlos, Fuenlabrada, Spain
[2]Escuela Técnica Superior de Ingeniería Informática, Universidad Rey Juan Carlos, Mostoles, Spain
[3]Indian Institute of Technology, Kharagpur, India

Abstract

Background: The source code of the Linux kernel is being changed about 80,000 times a year. Each of these changes (commits) is for a reason. The implications of those reasons are huge, given the critical role of Linux for the industry. Linux is also an example of continuous development projects, very common nowadays. However, there is little research on what is changing in it, and the reasons for those changes, which is important to learn which kind of development and maintenance activities are being performed.

Research objective: To learn about Linux source code changes. In particular, what types of changes are happening, and how those changes can be classified in a way that is relevant for different stakeholders.

Methodology: We selected a random set of 499 commits to the Linux kernel. We then annotated, by three different annotators, to which kind (bug fixing, bug preventing, perfective or new feature) belongs each of them. Finally, we analyze the resulting classification.

Results: We find that about half the changes to the source code of Linux correspond to perfective commits. About one sixth, to new features. About one third, to bug suppression (including fixing and preventing). We also noticed a relatively low agreement between annotators, specially in the case of bug preventing commits, and the difficulty of classifying many commits.

Conclusions: This is a first estimation of the kinds of changes to the Linux source code, which should be refined by more detailed studies. We also found that there is room for defining with more precision the kinds of changes, and also for rethinking the classification in terms of ranges, and specific interests, instead of absolute categories.

Keywords

software engineering, software development, software maintenance, changes to source code, Linux, dataset, bugs, bug fixing, bug prevention, perfective changes, commits, new features,

1. Introduction

The Linux kernel is one of the most relevant free, open source software (FOSS) projects. It follows a continuous development process[1], with an almost continuous flux of changes to the source code, and minor releases every few weeks. Development (new functionality) and maintenance (bug fixes, refactoring, etc.) activities are performed in parallel. This is frequent in modern FOSS and non-FOSS projects, which are no longer structured as a first stage for development, up to a "put into production" moment, and then a maintenance stage, which does not add new functionality. On the contrary, continuous development projects perform activities of development of new features, and maintenance activities, in parallel, even when there are windows of time when preference is given to merging maintenance changes, and windows when new features are preferred.

BENEVOL24: The 23rd Belgium-Netherlands Software Evolution Workshop, November 21-22, Namur, Belgium
*Corresponding author.
jesus.gonzalez.barahona@urjc.es (J. M. Gonzalez-Barahona); michel.maes@urjc.es (M. Maes); abhishek16@kgpian.iitkgp.ac.in (A. Kumar); david.morenolu@urjc.es (D. Moreno-Lumbreras); gregorio.robles@urjc.es (G. Robles)
https://gsyc.urjc.es/jgb (J. M. Gonzalez-Barahona); https://dlumbrer.gitlab.io (D. Moreno-Lumbreras)
0000-0001-9682-460X (J. M. Gonzalez-Barahona); 0000-0002-8138-9702 (M. Maes); 0009-0005-8300-8327 (A. Kumar); 0000-0002-5454-7808 (D. Moreno-Lumbreras); 0000-0002-1442-6761 (G. Robles)

The history of all the changes to the Linux source code, since 2005, is recorded in its main git repository. It is known as the "Torvalds repository" or the "upstream repository", by comparison with the "stable" repository, where branches intended for production are composed from commits in the upstream repository. Currently, it has in the order of 80K commits (individual changes) per year. Understanding the motivation for those commits is fundamental to learn about the development and maintenance activities performed by Linux kernel developers. By studying them, it is possible to learn not only which changes correspond to maintenance activities, or development of new features, but also to reconsider what is maintenance and what is development of new features, in the context of the changes that Linux is experimenting.

Many studies have been performed on Linux and its development repositories. The following are some of those that are most closely related to the work we are presenting. Already in 2012, [2] used an approach based on automatically examining commit records to identify bug fixing commits. [3] focused on commits that change configuration files, to detect additions of removal of features, while [4] presented a dataset composed of threads discussing patches in mailing lists, including references to the corresponding commits. More recently [5] focuses on the automatic identification of commits fixing security issues, [6] studies a dataset of exploits of Linux, including a number of releases of it, and [7], which analyzes the communication accompanying submitted changes, which roughly corresponds to the commit comment when the patch is accepted.

Even when these studies deal with topics such as identifying which commits correspond to bug fixing and other maintenance-related tasks, or to new features and other development-related tasks, we are not aware of any one comprehensively studying what types of changes are committed into the Linux kernel, with the aim of understanding which kinds of development and maintenance activities are being performed on it. What is more important, we are not aware of studies targeted at understanding with some detail how modern processes of continuous development, in complex and critical systems such as Linux, are structured, and how they mix development of new functionality and maintenance of the code base.

Therefore, we decided to perform a nuanced, extensive "manual" analysis of code changes. This consisted in the analysis of 499 commits by three different annotators. The annotation classified each commit, with the aim learning about the development or maintenance tasks that those commits reflected. In the process, we found that in some cases the nature of a commit is "in the eyes of the annotator", which lead us to reflect about the differences between different types of maintenance and development commits, and the blurry limits between them.

The main contributions of this study are:

- A detailed dataset consisting of 499 commits of the Linux kernel, classified according to the kind of activity that lead to the code change.
- A characterization of the development and maintenance tasks performed in Linux, based on that classification.
- A reflection on the nature of bug fixing, bug preventing, perfective and new functionality changes to the source code in a continuous development project.

2. Method

Commits to the Linux kernel follow a well-established process, which includes discussion and peer review, before they are merged into the code base. This process is based on discussion threads in public mailing lists, where proponents, reviewers, and anyone else with interest interact. It usually has several stages, since the contribution structure to Linux is hierarchical. A proposed patch may be discussed first at the module level, then at the subsystem level, and maybe later at the system level. In each stage, the proposed change may be rejected, accepted as such, or accepted with some modification. Finally, at some point, accepted changes are merged in the Torvalds ("upstream") repository.

At certain points in time, usually every 9-10 weeks, a new release (a mainline kernel, in Linux *parlance*) is tagged in that repository, meaning that a new point release branch (for example, 6.2) will be

opened in another repository, the "stable" repository, intended to be used for production code[1]. Since we were interested in all the changes to the Linux source code, we decided to use the Torvalds repository as our main data source. The specific method we followed in our study, based on this data source, is described below:

Random sample. We selected a random sample of 499 commits from all the commits in the Torvalds repository of the Linux kernel with committer date in 2022.

Manual annotation. We developed annotation guidelines, an annotation process, and a web-based tool to let annotators read commit messages, and if needed the full commit (including changes to the source code) and the thread discussing the commit in the corresponding mailing list. Three different annotators classified all the commits.

Analysis. We compared the results of the three annotations, and performed a statistical analysis of them.

2.1. Random sample

For the random sample, we started with a dataset which includes metadata for all commits in the Torvalds Linux repository, up to 2023 [8], obtained using the Perceval tool [9]. From that dataset, we filtered all commits with a commit date in 2022, totaling 85,820 commits. We then selected a random sample of 499 commits from it. For each commit, the sample includes the commit hash (unique identifier), date, and message (among other data not used for this research).

2.2. Manual annotation

The annotation consisted, for each commit in the random sample, of a careful manual analysis of the commit, with the aim of classifying it. If the commit was a merge commit, we ignored it for the annotation. To maximize the consistency of the annotation criteria, we used a detailed definition of each category (see Appendix 2.3). We insisted that annotators had to classify according to the major "intention" of the commit, to clarify how to interpret the classification. We also wrote annotation guidelines, which all annotators read carefully before starting their annotations. These categories, and the definitions we used, were produced after a refinement process, based on several previous test-runs of the annotation.

The annotation was performed by three of the authors of this paper, independently. All of them had formal training in computer engineering, and have developed software themselves. Two of them were, at the time of performing the annotation, PhD in software-related areas, and the third one was in his third year towards the same goal.

To facilitate the annotation, we developed a tool to assist annotators in their task (see Figure 1). It allowed annotators to select a commit hash from the sample to annotate, visualize its commit message, and, if needed, browse the mailing list thread where it has been discussed, and the full commit info, including changes to the source code. Using this tool, annotators classify each commit using a Likert scale of 5 values from several points of view. The most relevant for this study are:

- Is this a bug fixing commit (BFC)?
- Is this a bug preventing commit (BPC)?
- Is this a perfective commit (PRC)?
- Is this a new feature commit (NFC)?

Using the tool, annotators read carefully the message in each commit, and if they considered it necessary, they also checked the corresponding review threads in the Linux mailing lists, and the changes to the source code. When they considered they were ready, they filled in their annotation, by

[1]Active kernel releases: https://www.kernel.org/category/releases.html

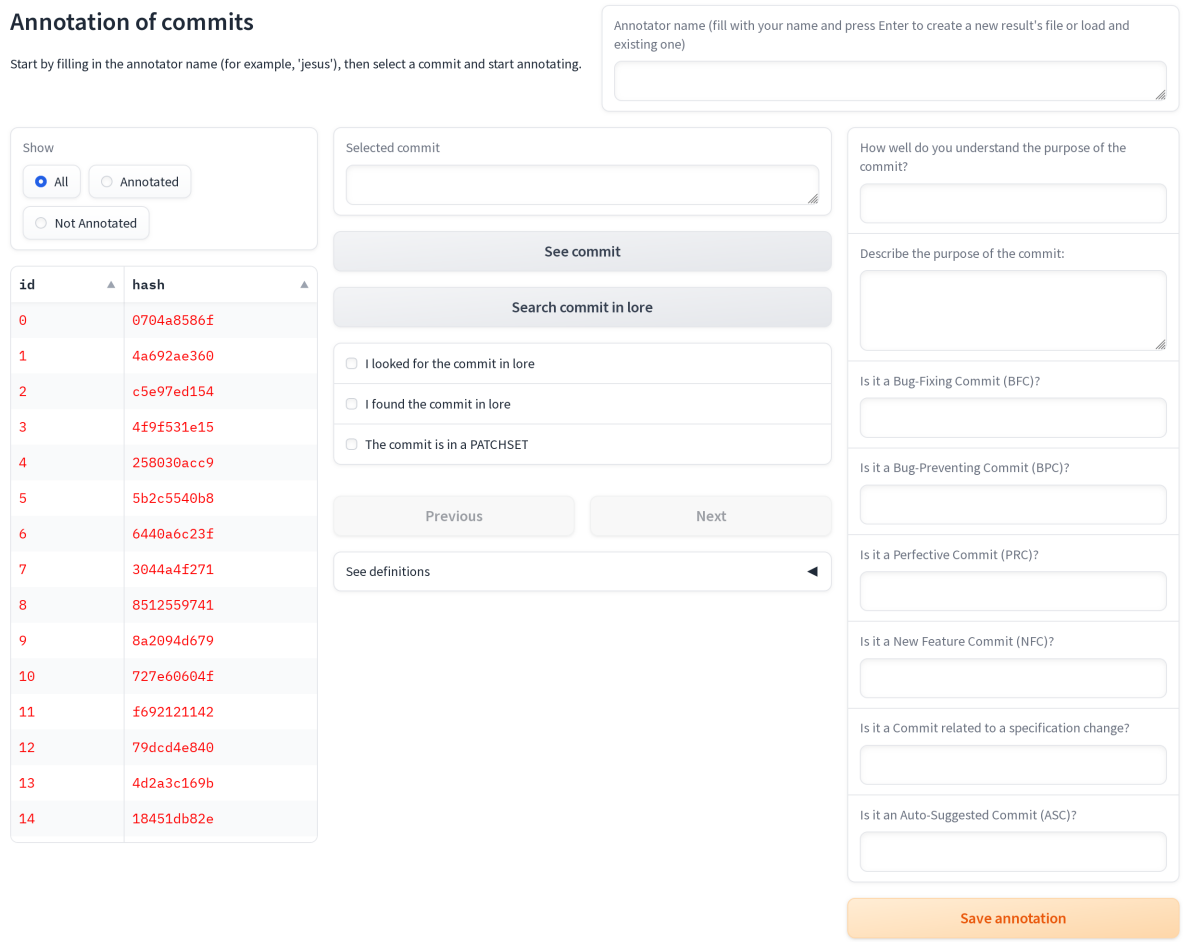

Figure 1: Screenshot of the tool used by annotators.

Agreement	all	A,B	A,C	B,C
BFC	0.790	0.755	0.810	0.807
BPC	0.456	0.231	0.464	0.638
PRC	0.736	0.629	0.738	0.837
NFC	0.787	0.692	0.826	0.847
BSC	0.658	0.505	0.703	0.754

Table 1
Krippendorff's alpha values for estimating agreement between annotators for each type of commit (including BSC as the merge of BFC and BPC). The column "all" shows values for agreement between all annotators. The other columns show values for pair of annotators (A, B, and C), irrespectively of the value selected by the third annotator

selecting a value in a menu with the options of the Likert scale for each of the questions above (among others). The annotation was then saved to a CSV file.

After the annotation, we studied the agreement between annotators, using the Krippendorff's alpha coefficient [10]. We used this measure for agreement because we needed a metric capable of working with more than two annotators, and with a discrete, but ordered, set of possible values for each annotation. The values for agreement are described in Table 1. That table shows how the agreement between the three annotators is between 0.67 and 0.8, except for BPC. This would mean that the agreement for BFC, PRC, and NFC is good enough to draw tentative conclusions, but not for BPC.

To mitigate this problem, which could be due to a difficulty in telling BPCs apart from BFCs, we created a new category, BSC (bug suppression commits), to merge both of them. See details below for more information about how that category was created during the analysis. In Table 1 we also show

the agreement for this category, which is very close to 0.67, and therefore, very close to be good enough to draw tentative conclusions.

2.3. Definition of categories

For the annotation, we used some definitions which all annotators had to read carefully before starting annotation. For the matters discussed in this paper, the most relevant definitions are:

Purpose of the commit. The purpose of the commit is the intention with which the code change (commit) was done. It will usually be a short text, which to some extent may be a summary of the commit message. But in the end, it is what the annotator understands as the purpose or intention of the code change.

The annotator will not always fully understand that purpose, because of technical complexities, unfamiliarity with the details mentioned, deficiencies in the commit message and associated information, or anything else.

Definition of failure, bug and fault. Failures are defined as erroneous behaviour of the software system, when the system deviates from its expected behavior, resulting in an inability to perform its intended functions or deliver the expected outputs. If the failure is caused by a software fault (a bug), it can be fixed only by some change to the software. Therefore, a commit that doesn't change the source code (including its default configuration or other assets that may influence in the behaviour of the system) cannot fix or prevent a bug.

Therefore, we will use "bug" and "fault" as synonyms, which will cause failures. If no failure was found, no bug was found either, so any change could maybe be preventive of bugs, but for sure it will not fix a bug. Also, if a commit does not change the behaviour of the system, it cannot fix a bug (because behaviour before and after the commit is the same). Therefore, changes which only touch comments, or which are only refactoring, cannot fix a bug.

We will also consider changes to the default configuration as changes that could change the behaviour of the system, and therefore could introduce and fix bugs.

Bug-fixing commit (BFC). Any commit that fixes a bug present in the source code, defined as a software fault that manifests as a failure when running in a certain way the version of the system previous to the commit.

Bug-Preventing Commit (BPC). Any commit that prevents a bug that could cause a failure in the future. This type of commit does not fix known bugs, but possible, still undiscovered bugs, that could happen in the future. For example, they could improve the values returned by a function, in a way that is likely to prevent failures in code calling that function.

Perfective Commit (PRC). Any commit that improves the quality of the code (understandability, composability, performance, etc). This includes refactoring, optimizations, code style improvements, adding comments, etc. The key aspect is that perfective commits do not fix bugs or add new features.

New Feature Commit (NFC). Any commit that adds new functionality or capabilities to the codebase. This includes adding support for new hardware, implementing new APIs, exposing new configuration options, etc. The key aspect is that new feature commits add new code to enable new behaviors not previously possible.

2.4. Analysis

We estimated the type (BFC, BPC, PRC, or NFC) of each commit using the values produced by the annotators. To estimate a single value for each commit, we computed the mean of the values produced

Kind	Number of commits
PRC	249
BFC	104
NFC	79
BPC	67

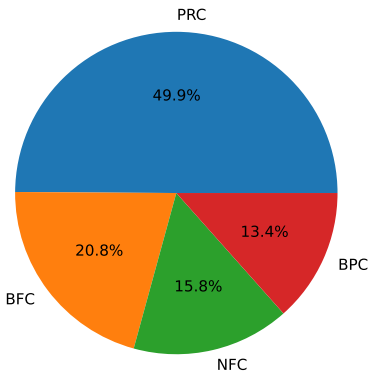

Figure 2: Results of the annotation: number of commits per kind of commit, and chart with the corresponding fractions.

by the annotators for each type of commit, and used the largest of those means to estimate the type. Then, we considered the sample to be representative of the whole set of commits committed during 2022. Therefore, we estimated the number of commits of each type during 2022 by considering their fraction was like the one in the annotated sample.

We also created a new category, result of merging BFC and BPC. We called it "bug suppression commits" (BSC) because these both types have a similar intention: to reduce bugs in the source code. For creating it, we use, for each commit and each annotator, the maximum value of BFC and BPC. We then repeated the same analysis for the three resulting types (BSC, PRC and NFC), since the level of agreement for them make results more plausible.

3. Results

The results of the annotation process are presented in the table (absolute numbers) and chart (fractions) in Figure 2. About half the commits were classified as perfective commits (PRC), about one fifth as bug fixing commits (BFC), and the rest as bug preventing commits (BPC) and new feature commits.

We also grouped the annotations as BFC and BPC into a single "bug suppression commit" (BSC) category. This resulted in BSCs being about one third of all commits, which represents the fraction of changes to the source code that are targeted at dealing with bugs. Notice that, when merging BFC and BPC in the BSC category, overall numbers for categories may change, as it happens with PRC in Figure 2 (247) and Figure 3 (248). This is because the classification of a single commit changes from PRC to BSC: when considering only BSC, PRC and NFC, the mean for BSC for the three annotators is larger than the mean for PRC.

Extrapolating the fractions found in the annotation, we can obtain an estimation for the number of commits of each type for all commits committed during 2022, as shown in Table 4. We can see how more than 42,000 changes to the source code were perfective, while more than 29,000 were changes for fixing or preventing bugs. All of them (more than 71,000 commits) could be considered as maintenance commits, since they do not change functionality (except for dealing with bugs). Only about 13,000 commits were for adding new functionality, which could be considered development activity. Therefore, we can estimate that about 84.2% of all commits correspond to maintenance activity, while about 15.8% of all commits correspond to development activity.

Kind	Number of commits
PRC	248
BSC	172
NFC	79

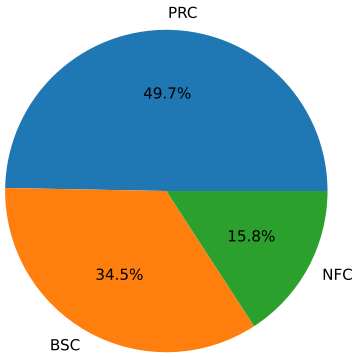

Figure 3: Results of the annotation, merging BFC and BPC in a single type (BSC): number of commits per type of commit and chart with the corresponding fractions.

Kind	Number of commits
PRC	42,824
BFC	17,886
NFC	13,587
BPC	11,523

Kind	Number of commits
PRC	42,652
BSC	29,581
NFC	13,587

Figure 4: Estimation of the total number of commits based on the fractions found as result of the annotation.

4. Discussion

Even when our study shows some preliminary results about the type of changes to the source code in the Linux kernel, we found the process to conduct the study as even more interesting than the results themselves.

4.1. Discussion: context

Linux kernel development and maintenance is complex [11], with activities structured in a hierarchy of hundreds of subprojects, using a diverse and relatively peculiar set of development support tools, compared to the usual uses in FOSS projects. For example, most of the code review is done via mailing lists, and bug reporting is spread among a Bugzilla instance, mailing lists, issues in some GitHub repositories, and other means.

Despite these peculiarities, and likely because of its importance, it has been widely studied. In the specific case of changes to the source code, there are some early papers about its evolution from the early 2000s [12, 13], which are focused on the overall evolution, and not in characterizing individual changes to the source code. More directly related to changes to the source code, the most relevant studies are focused on how changes fix or introduce bugs. [14] empirically studied bugs in OpenBSD and Linux, a study which was replicated 10 years later [15]. Both of them use the results of applying static analysis tools to the source code to detect bugs in it. Using bug reports as the main data source, [16] characterized bugs reported and fixed in three FOSS projects, including 829 in the Linux kernel, and [17] analyzed in depth 5,741 bugs reported in the Linux Kernel.

However, we are not aware of comprehensive studies classifying all changes to the source code of Linux reflect. With the industrial importance of Linux, and being it also a key example of continuous development (when maintenance and development activities are performed continuously in the software), we considered it was important to shed some light on this area. For that, we decided to start from Linux commits in the Torvalds repository, because all changes to the kernels used in production

go through it. Since we couldn't find any automated technique proved to be useful to classify Linux commits, we also decided to start with a detailed manual classification. This allows us to both having a ground truth for future studies, and to learn about the nuances of commits in Linux, and their classification.

4.2. Discussion: study

While doing our nuanced study of Linux commits, we found many interesting details about source code changes. Two of them, which we think deserve further research, are:

Classification is in the eyes of the classifier. When writing some change to the source code, the intention is always, to some extent, to improve the code. But the exact way in which the code is perceived to be improved depends on the context of the observer. We observed this effect in some test-runs of our annotation that we performed before the one described in this paper. When explaining the reasons for each classification, each annotator had different, but in general valid, views. For example, what one annotator perceived as a perfective commit, just making the code more readable, was perceived by other as preventing future bugs, because being the code easier to read and understand would likely have that effect. An extreme case would be two different persons, one interested in counting all cases where any kind of problem with the code was fixed, to estimate the number of problems (flaws) present in the code before the fixes. The other one would be interested in finding all cases where potential future problems are prevented. Confronted to a commit solving a memory leak in some circumstances that have never been described in real use cases, the first one would likely classify it as a bug fix (a certain problem that could be happening is fixed), while the second one would likely classify it as a bug prevention (a potential problem, which has never been reported, is fixed).

Continuous range of categorization. Instead of separate categories, we may have a continuous range of them. There are two main reasons for this: (1) even in the case of small commits, a single one may, for example, fix a bug but at the same time refactor some code; and (2) the frontiers between categories are blurry, such as the difference between fixing a bug and preventing future bugs described above. With respect to (1), even when commits in the kernel tend to be relatively small, because it is encouraged to devote each commit to a single task, the case happens in a number of commits. For example, we found cases of adding some new functionality while refactoring code, or preventing future bugs while adding some functionality. With respect to (2), it happens between all categories. For example, there are perfective changes that, maybe because they improve the code, also prevent future bugs. There are bugs that are fixed by adding some new functionality.

Even with precise definitions for all categories, we think these two are important reasons for the relatively high disagreement between annotators that we observed.

Also, we found some other issues, which may be more particular to the development practices of the Linux kernel, but we found important to produce an accurate annotation:

Patch sets. Changes usually do not happen in isolation, but in patch sets, also named "patch series", that are reviewed together. A patch set usually includes a number of individual changes that are related somehow. For example, they together may fix a bug, some of them refactoring code to easily produce the fix, and only one actually fixing the bug. Or all of them together may add some new functionality. This means that commits have to be considered in the context of their patch set. For example, we may have a commit that "fixes" a call to a function that has two parameters, but should have three of them. Isolated, this seems to be a clear bug fix. But when it comes in a patch set with other commit adding that third parameter to the called function, it becomes a part of some new functionality.

Changes in requirements. In projects such as the Linux kernel, there are no clear specifications or requirements, and usually the consensus among developers is considered as "the" specification. Therefore, it is difficult to track how they evolve. But knowing the specifications is fundamental to decide if certain behavior is correct or not, and therefore, if we are in front of a bug or not. Consider for example a change adding read/write support for a device that had only read support. It could happen that the previous specification (the consensus among developers, in this case) was that the kernel only supported that device in read mode, and now they are adding new functionality, because specification changed, and they want to support it now in read/write mode. But it could also happen that the "specification" assumed the device should be supported in read/write, but it was not, and therefore the commit is fixing a bug.

All of these issues make it difficult to strictly classify changes to the source code. Which means that even answering simple questions such as "how many bugs were fixed in Linux during 2022?" or "how many changes were done to the source code to add new functionality?" doesn't have a single answer if a lot more of detail is not considered. It is also worth noticing that, if we want to produce datasets with BFCs to use them to detect bugs or to research how bugs are fixed, we need to have all these problems in detail. We need to consider commits in context, to account for changes in informal specifications, and of course to deal with difficulties in determining if a given commit is really "fixing a bug" or not.

4.3. Discussion: results

Despite all the problems mentioned in the previous section, we managed to get a level of agreement between annotators which seems good enough to make our results sound, even when it is clear that they may have a large error interval.

The first interesting result is that about half the commits are perfective commits, which by our definition, are changes that improve the source code in some way, but have no direct impact on functionality (except for, maybe, performance). From our point of view, this is a very interesting result, showing how Linux developers are not only interested in ensuring a correct behavior (fixing or preventing bugs), or in adding new functionality, but also on improving the overall quality of the code. In fact, given the numbers we found, we could say that they are mostly interested in improving the quality of their code, since they are devoting half their code changes to it. On the other hand, this is a result that maybe could be expected for a project as mature, large, and relatively stable as the Linux kernel, in which perfective actions make a lot of sense.

It is also noteworthy how in such a mature software system like the Linux kernel, new functionality is still added: close to one of every six changes are for adding new features. If we have into account that some of the perfective commits are also done to more easily support new functionality, the proportion would be higher. And all this activity is done while the project is fixing and preventing bugs. The fact that this mixture of activities is working, producing a kernel suitable for production, shows a very high level of coordination between both types of activities.

With respect to the soundness of our results, it is clear that the classification of a commit as a BPC is the weakest one: the disagreement between annotators is high. This is the main reason we produced a second classification merging BFC and BPC into the single "bug suppression commit" category. On the one hand, this merge makes the distinction between "fixing" and "preventing", which is sometimes difficult to assess, unnecessary. On the other hand, we could also argue that if developers considered it important to prevent some kind of bugs, maybe they were already manifesting, even if they were not reported. Following this merge, we can say that about one third of the activity (measured by number of changes) is devoted to suppressing reported or not-reported bugs. And we can say that with relative soundness, since in this case the agreement between annotators is much higher. Unfortunately, this merge also blurs the distinction between fixing and preventing, which could be important if we want to consider the position of Linux developers to improve the reliability of their kernel.

All in all, our results show quantitatively how a relevant continuous development project is working, keeping a balance between dealing with bugs, improving the quality of the code, and adding new

functionality.

4.4. Discussion: threats to validity

The validity of the results presented in this paper are subject to several threats:

Internal validity. The most important threat to validity in this area is the low level of agreement measured in the case of BPC, and the overall only "acceptable" level of agreement. We think that this mainly means that the error interval of our results may be large, but still is acceptable as a first measure. Another threat is that all annotators have an academic background, which may bias the results: having annotators with more diverse backgrounds would certainly be beneficial.

External validity. Our sample of commits is relatively limited (499 out of 85,820 for the whole year 2022. Maybe it is not representative enough of what happened that year. The generalization of results to longer periods could be wrong, because maybe 2022 was different from other years for any reason. For example, that year had a precise combination of releases, which could affect the commits in the sample. The generalization to other continuous development projects beyond Linux is of course out of scope, and we are not claiming it, because every project could behave in very different ways.

Construct validity. Our definitions, and in general our annotation guidelines could be biased, not appropriate, or misleading. However, we were careful in considering previous literature, and our experience of several previous test-runs, that we used to refine them. The specific tool that we used for the annotation could also be the cause of bias, although we have been careful in designing it in way as much neutral as possible. Finally, the specific classification we intended, in four categories, could also be the cause of bias, specially in the context of the continuity of the categorization that we have observed.

Other threats. The Linux kernel is a large system. That means that different parts of it could behave very differently. Since we're aggregating changes to all those parts, maybe the "mean" effect we are observing is masking very different behaviors in different parts, and that could bias results. For example, a large part of the changes is related to support for specific devices, and that kind of code could behave very different from more "core" parts, such as the memory or the tasking subsystems.

5. Conclusions

Since many years ago, it is common that software development projects follow a "continuous development" model, in which maintenance and development of new features are intermixed. The Linux kernel is a prominent case of this kind of development. In this paper, we have presented a preliminary study on the intention of code changes to it, finding the approximate proportion of four types of it. The study shows the importance that improvement to the code base has in that project, and how new features are a relevant, but relatively small, part of the activity. To get those results, we have analyzed in depth a set of almost 500 code changes.

In the process of this annotation, we have also found some interesting details, which could lead to future research lines. In particular, we found that the boundaries between the types of commits are blurry, and that instead of concrete categories, we have more like a continuum of shades, with commits being not exactly in a single category, but having aspects of two or more of them. We also found that the peculiarities of the development model make it difficult to produce good datasets from which sound conclusions can be obtained. However, it is of great importance to build those datasets, since gaining understanding of how Linux works is fundamental in the context of the current IT industry.

Based on the lessons learned in this study, we will work on improving the annotation of Linux commits, and on doing more detailed analysis of the intention of developers when they change the

source, with the aim of explaining those more than 80,000 changes that the code of the Linux kernel is having every year. We also think that, in general, more studies are needed to better understand the different reasons for changes in continuous development projects, and what we consider as a bug, a bug fixing change and a bug preventing change, which could lead to revisiting the idea of what is a bug, in a project with changing requirements and interlaced activities involving development and maintenance. We think these studies are fundamental to get to a theory of how software quality is managed over time in this kind of projects.

Author roles (CRediT statement)

Jesus M. Gonzalez-Barahona: conceptualization, methodology, software, validation, analysis, writing, visualization, supervision; Michel Maes: software, investigation, data curation, writing; Abhishek Kumar: investigation, data curation; David Moreno-Lumbreras: investigation, data curation; Gregorio Robles: conceptualization, methodology, analysis, writing.

Reproducibility and data availability

The study presented in this paper is based on data from [8]. The specific data presented in this paper (annotations and final results), along with the software for producing results, are available in a reproduction package[2].

Reproducibility self-assessment, according to [18] (details in the reproduction package):
Raw: R, Extraction: U+*, Parameters: U, Processed: U+*, Analysis: U+*, Results: U+*

Acknowledgments

Thanks to the developers of the Linux kernel, who by publishing so much information about their development and maintenance processes make our research possible. This study has been funded in part by Spanish Government under the Dependentium project, grant number PID2022-139551NB-I00.

References

[1] B. Fitzgerald, K.-J. Stol, Continuous software engineering: A roadmap and agenda, Journal of Systems and Software 123 (2017) 176–189.
[2] Y. Tian, J. Lawall, D. Lo, Identifying Linux bug fixing patches, in: 34th International Conference on Software Engineering (ICSE 2012), IEEE, 2012, pp. 386–396.
[3] L. Passos, K. Czarnecki, A dataset of feature additions and feature removals from the Linux kernel, in: Proceedings of the 11th working conference on mining software repositories, 2014, pp. 376–379.
[4] Y. Xu, M. Zhou, A multi-level dataset of Linux kernel patchwork, in: Proceedings of the 15th International Conference on Mining Software Repositories, 2018, pp. 54–57.
[5] S. Page, Analysing Linux kernel commits, Blog post, 2023. https://sam4k.com/analysing-linux-kernel-commits/.
[6] G. Duan, Y. Fu, M. Cai, H. Chen, J. Sun, DongTing: A large-scale dataset for anomaly detection of the Linux kernel, Journal of Systems and Software 203 (2023) 111745.
[7] X. Tan, M. Zhou, How to communicate when submitting patches: An empirical study of the linux kernel, Proceedings of the ACM on Human-Computer Interaction 3 (2019) 1–26.
[8] M. Maes Bermejo, J. M. Gonzalez-Barahona, M. Gallego, G. Robles, A dataset of Linux kernel commits, 2024. URL: https://doi.org/10.5281/zenodo.10654193. doi:10.5281/zenodo.10654193.

[2]Reproduction package: https://zenodo.org/doi/10.5281/zenodo.13770273

[9] S. Dueñas, V. Cosentino, G. Robles, J. M. Gonzalez-Barahona, Perceval: software project data at your will, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ACM, 2018, pp. 1–4.

[10] K. Krippendorff, Estimating the reliability, systematic error and random error of interval data, Educational and psychological measurement 30 (1970) 61–70.

[11] M. S. R. Wen, What happens when the bazaar grows: a comprehensive study on the contemporary Linux kernel development model, Ph.D. thesis, Universidade de São Paulo, 2021. URL: https://doi.org/10.11606/D.45.2021.tde-07092021-041136. doi:10.11606/D.45.2021.tde-07092021-041136.

[12] M. W. Godfrey, Q. Tu, Evolution in open source software: A case study, in: Proceedings 2000 International Conference on Software Maintenance, IEEE, 2000, pp. 131–142.

[13] C. Izurieta, J. Bieman, The evolution of FreeBSD and Linux, in: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, 2006, pp. 204–211.

[14] A. Chou, J. Yang, B. Chelf, S. Hallem, D. Engler, An empirical study of operating systems errors, in: Proceedings of the eighteenth ACM symposium on Operating systems principles, 2001, pp. 73–88.

[15] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, G. Muller, Faults in Linux: Ten years later, in: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, 2011, pp. 305–318.

[16] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, C. Zhai, Bug characteristics in open source software, Empirical software engineering 19 (2014) 1665–1705.

[17] G. Xiao, Z. Zheng, B. Yin, K. S. Trivedi, X. Du, K.-Y. Cai, An empirical study of fault triggers in the Linux operating system: An evolutionary perspective, IEEE Transactions on Reliability 68 (2019) 1356–1383.

[18] J. M. Gonzalez-Barahona, G. Robles, On the reproducibility of empirical software engineering studies based on data retrieved from development repositories, Empir. Softw. Eng. 17 (2012) 75–89. URL: https://doi.org/10.1007/s10664-011-9181-9. doi:10.1007/s10664-011-9181-9.