The Journal of Systems and Software 231 (2026) 112647

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

The life of software features: An exploratory case study of 189 feature requests in Marlin☆,☆☆

Aron van der Hofstad [a], Loek Cleophas [a,b], Clemens Dubslaff [a], Jacob Krüger [a,*]

[a] Eindhoven University of Technology, The Netherlands
[b] Stellenbosch University, South Africa

ARTICLE INFO

Keywords:
Software evolution
Features
Marlin

ABSTRACT

Features are a widely established notion to organize the functionalities of a software system. For instance, features are used to define variability and commonalities in product lines; feature-driven development is an agile development methodology; and social-coding platforms have explicit support for feature requests. Despite the importance of features, we are not aware of extensive research on their life cycles: how and for what reasons do developers evolve features? As a result, we lack an understanding of how features come to be, how they are evolved, or why they may be removed. To narrow this research gap, we have performed an exploratory case study on the evolution of 189 feature requests of the Marlin 3D-printer firmware. We identified the code introducing a feature and traced all commits touching that code or the feature, resulting in a collection of 1,940 unique commits spanning five years of evolution. We have manually inspected all of these commits to classify their intentions with respect to the features they change, and created process graphs of the features' life cycles based on these intentions to understand the evolution of features. Our results contribute a first overview and detailed examples of evolving features beyond code metrics, showcasing that features are primarily refactored, exhibit interdependent evolution, and are rarely removed. Serving as a starting point, these contributions can support practitioners in managing features and guide researchers in understanding feature evolution as well as in scoping future studies on this matter.

1. Introduction

Many software systems and their development are structured around *features*, which have become a widely established notion in software-engineering research and practice. For example, software product-line engineering (Apel et al., 2013; Pohl et al., 2005; Krüger et al., 2020a) or feature-driven software development (Palmer and Felsing, 2001) are entire methodologies structured around the notion of features. Similarly, developers on social-coding platforms (e.g. GitHub) often use features to organize the development of their projects (Stănciulescu et al., 2015; Krüger et al., 2019b). For instance, developers label issues as feature requests and refer to these labels in pull requests.

Interestingly, what exactly a feature is and what artifacts or properties it comprises is an ongoing debate yielding many different definitions and specifications (Apel et al., 2013; Classen et al., 2008; Berger et al., 2015). In essence, features are an abstract concept, for which most developers have an intuitive, but varying, understanding. This understanding depends on a developer's individual expertise and experiences, but typically a feature can be broadly defined as "a characteristic or end-user-visible behavior of a software system" (Apel et al., 2013).

While features have become an important notion in software engineering, their life cycles have received little attention in research. Mostly, researchers have been concerned with code metrics and configuration options (Fischer, 2021; Passos et al., 2016; Kröher et al., 2018), limiting our understanding of the *intentions* with which developers evolve features. Within this article, we build on the concept of intentions to describe the goal due to which developers change a feature (Krüger et al., 2023, 2024). The level of intentions is important to consider, since features are abstractions of the software and can include other artifacts, too. Moreover, code changes alone do typically not specify what a developer's original intentions for a change were, can implement these intentions incorrectly, and may tangle multiple intentions that do not align with the original one (Krüger et al., 2024). For instance, a developer may intend to improve the performance of a

☆ This article is part of a Special issue entitled: 'Syst.+Sw.ProductLineEng.' published in The Journal of Systems & Software.
☆☆ Editor: Prof Raffaela Mirandola.
* Corresponding author.
E-mail addresses: l.g.w.a.cleophas@tue.nl (L. Cleophas), c.dubslaff@tue.nl (C. Dubslaff), j.kruger@tue.nl (J. Krüger).

https://doi.org/10.1016/j.jss.2025.112647
Received 12 February 2025; Received in revised form 17 September 2025; Accepted 22 September 2025
Available online 25 September 2025
0164-1212/© 2025 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

feature (goal of a change), but may also refactor variable names in the process to make the code more comprehensive (tangled, unintended change)—potentially modifying multiple features to achieve different goals for each within one change, too. Metrics on feature code and changes can also be misleading (Ludwig et al., 2019). For instance, removing feature code can mean several things: it could actually be removed, be subsumed by another feature, or be moved to a different location. These intentions are not visible from code changes alone.

Consequently, focusing only on code can severely limit our understanding of how software and its features evolve. We argue that it is equally important to understand the intentions for which developers evolve features. For example, understanding when and why developers decide to deprecate a feature or rework it can help define indicators for unnecessary features or for quality problems. Investigating the life cycles of real-world features can yield insights into why features succeed, why they become obsolete, or when they may become commodity.

As a step in this direction, we report an exploratory case study on the life cycles of 189 pull requests labeled as feature requests in the Marlin 3D-printer firmware.[1] To elicit the features' life cycles, we tracked the code changes related to them, covering 2956 commits (1940 unique commits). We manually reviewed each feature request, the respective code, the commits touching that code and feature, as well as all connected developer discussions and messages to understand why the features were changed (the intended goal). Additionally, we analyzed 180 pull requests not labeled as features to determine whether these would meet the definition of a feature by Apel et al. (2013) to reason on the reliability of feature labels. Our findings can support developers in making informed decisions while developing and maintaining software features. For example, we advise developers to establish and document a change-labeling strategy, agree on what a feature comprises, and untangle changes to facilitate collaborative development and maintenance. In turn, each developer has a shared understanding and access to relevant information when working on a feature. Researchers can use our study as a starting point for more in-depth analyses of feature life cycles. We publish our data and supplementary materials in a persistent open-access repository.[2] Our dataset allows researchers to replicate our work, and is a helpful artifact for developing automation for identifying (intentions of) feature changes, untangling changes, and studying the underlying causes for such changes. Thus, based on our findings and dataset, future work can investigate specific feature and change properties in greater detail, which also enables expanding to multiple systems.

2. Case-study design

First, we describe the methodology of our exploratory case study, which we used to obtain deeper insights into a single case (Yin, 2018). We decided for a case study to provide a starting point for further investigations on the intention level of feature evolution, which has not been researched so far. Inherently, case studies do not aim for and are limited regarding generalizability, replicability, and internal validity (Ralph et al., 2021). Instead, the focus of case studies, including ours, is on transferability, meaning that we aimed to obtain insights that can be transferred to other cases in principle and that can inform future practice or research on feature evolution.

2.1. Research questions

Our goal was to shed light onto the life cycles of software features, focusing on the intentions behind their evolution in a substantial real-world system (Marlin, cf. Section 2.2). For this purpose, we defined three research questions (RQs):

RQ$_1$ *What feature labeling practices does Marlin employ?*
Open-source communities use labels for issues and pull requests. We built on such labels as a starting point for investigating how features evolve. For this purpose, we first had to obtain a detailed understanding of how Marlin developers label their issues and pull requests (**RQ$_{1.1}$**). This was necessary because we noticed in an exploratory investigation that the community used a huge variety of labels, not all of which were intuitive. Due to the goal of our case study, we paid particular attention to the label *PR: New Feature* (**RQ$_{1.2}$**), which we aimed to use as starting point for our remaining research questions. For this label in particular, we investigated to what extent its respective issues or pull requests had potentially tangled changes, for instance, asking for a feature and bug fix or refactoring. So, by understanding the labeling practices and change tangling, we informed our study design and contribute insights into how Marlin developers manage their repository and features.

RQ$_2$ *How do Marlin's software features evolve?*
In open-source projects like Marlin, new features are often proposed and developed via issues, fork-based development, and pull requests (Gousios et al., 2014; Jiang et al., 2017; Stănciulescu et al., 2015; Krüger et al., 2019b; Zhou et al., 2020). Pull requests labeled as feature requests represent a starting point for studying feature evolution: They define a clear point in time when a new feature is added to a system, and the label implies agreement by the community that it is, in their understanding, a feature. By tracing later changes to the introduced features, we aimed to understand for what reasons these features are changed afterwards and to thereby elicit their life cycles.

RQ$_3$ *To what extent do (labeled) pull requests in Marlin align to the definition of features by Apel et al. (2013)?*
Our research started with pull requests that the developers of Marlin labeled as feature requests. Consequently, an important question for contextualizing our findings is to what extent these labels are accurate or are missing for other pull requests. For this purpose, we compared pull requests that are labeled as feature requests to those not labeled as feature requests. This way, we aimed to understand whether the Marlin developers' notion of features aligns to the (broad) one by Apel et al. (2013) that we use.

By addressing these research questions, we contribute to a better understanding of the notion of features and their life cycles to guide future research and practitioners.

2.2. Case selection and site description

Analyzing developers' change intentions (Krüger et al., 2024, 2023) regarding individual software features throughout their entire life cycle requires an in-depth investigation of various software artifacts. For this reason, we decided to conduct a manual, exploratory case study involving one substantial subject system. To select that system, we compiled a list of the 100 most starred repositories according to the Gitstar Ranking.[3] Then, we removed repositories not representing a software system but another type of artifact (e.g., lists, tutorials) and those that have no explicit label for feature requests.

Feature labels are important to improve the quality of feature location, since mere manual or automated feature location both have inherent limitations (Wilde et al., 2003; Wang et al., 2011; Krüger et al., 2019a; Razzaq et al., 2018; Rubin and Chechik, 2013). Specifically, as for the notion of what a feature is, the code locations belonging to a specific feature are also subjective. If we would manually recover (mandatory) feature locations, this would very likely not match the

[1] https://github.com/MarlinFirmware/Marlin.
[2] https://doi.org/10.5281/zenodo.17121913.
[3] https://gitstar-ranking.com/.

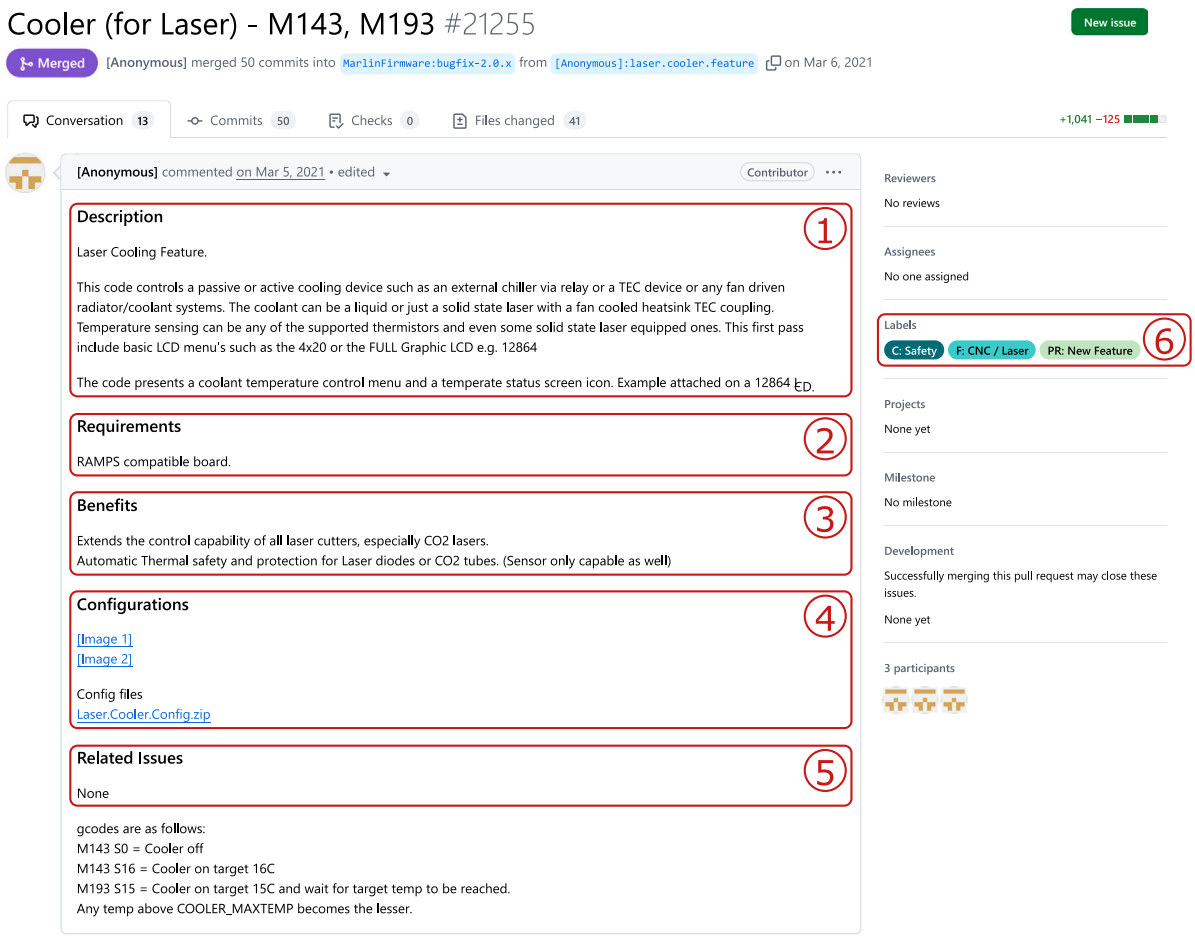

Fig. 1. Anonymized pull request 21255 of the Marlin system (https://github.com/MarlinFirmware/Marlin/pull/21255).

original developers' understanding of features and feature locations. For the same reason, existing techniques for automated feature location are also not reliable. To mitigate such problems, we decided to start from pull requests labeled as feature requests, building on the reasonable assumption that these represent features the developers agreed on. The pull requests also link to the respective feature code.

From the remaining systems, we picked the Marlin 3D-printer firmware.[1] Marlin is primarily implemented in C and C++, using C preprocessor directives to allow users to configure the software to their own hardware. It is a substantial system that exists since 2011 and involves more than 369,000 lines of C and C++ code, 20,000 commits, 70 releases, 1100 contributors, 14,000 issues, and 12,600 pull requests. Since Marlin provides a clear labeling system for pull requests, we could identify a reliable set of features the developers agreed upon and trace their source code—also for mandatory features. Moreover, Marlin is open-source software that has a broad community of contributors, which promises to yield a diverse set of features. Lastly, we (Krüger et al., 2018, 2019b) and other researchers (Stănciulescu et al., 2015; Viegener, 2021) have extensively studied Marlin in terms of its features and its variability. Therefore, we had ample material for Marlin available to design our case study and to inform our data analysis. For these reasons, we considered Marlin a feasible subject system for our exploratory case study. We then continued with extracting all pull requests and issues from Marlin via the GitHub API to enable our following analyses. Note that while we knew that Marlin would be a feasible subject system from our previous work, we followed a systematic selection procedure to identify whether another subject system would be even more suited.

2.3. Feature-request analysis (RQ_1, RQ_2)

In the remainder of this article, we distinguish between pull-request *labels* and commit **tags**, using these highlights to indicate the respective type. As explained before, the Marlin developers define a pull-request label to categorize a pull request, as we exemplify in Fig. 1 (6). Commit tags are codes we assigned to individual commits to specify their underlying intention (11 tags, explained shortly).

Domain Analysis and Labeling Practices. At first, leading us to RQ_1, we performed an extensive domain analysis of Marlin. For this purpose, we recapped the related work (cf. Section 6) and investigated Marlin's development practices. Specifically, the first author inspected Marlin's project website, code, issues, pull requests, commits, developer discussions, and documentation to understand the processes involved. We noticed that Marlin used very different labels, which is why we collected these through the GitHub API and analyzed their purpose. To this end, we first studied the Marlin documentation to understand the labels. When we noticed that some were not explained in the documentation, we investigated examples of these labels and reasoned on their purposes ourselves.

Identifying Feature Requests. To identify features labeled by the Marlin developers or users, we iterated through the pull requests of Marlin, starting with the most recent ones. We considered all pull requests as relevant for RQ_2 that were merged and were labeled as a feature request through the label *PR: New Feature*. We used the GitHub web UI to further inspect these pull requests, as Marlin has a standard

template that we could analyze more easily. As an example, we use pull request 21255,[4] which we depict in anonymized form in Fig. 1.

First, we read the description (1) and comments (discussions below a pull request) to understand what the feature is about and how it is supposed to work. Second, we analyzed the requirements that must be met (2), for instance that a machine supports RAMPS boards. Third, the benefits (3) helped us to understand why a feature should be introduced. Fourth, the configuration entry (4) provides images or configuration files to explain a feature's constraints. Fifth, related items (5) list issues that are solved by the pull request, which then served as an additional information source. Lastly, on the right side, a list of labels is present (6), which provided additional context and served as a manual cross-check of our automated crawling on whether we inspected a feature request. Then, to create a concise dataset, we extracted when the pull request was created and merged, its ID, how many comments it had, which files were changed, its title, its labels, and its description.

Tracing Commits. Locating feature code and changes to a feature is challenging, since features can be scattered and tangled while commits and pull requests may perform multiple intentions (e.g., a tangled bug fix or refactoring) (Krüger et al., 2024, 2023; Kirinuki et al., 2014; Dias et al., 2015; Queiroz et al., 2017; Ludwig et al., 2019; Liebig et al., 2010). To resolve this problem, we inspected each identified feature request to locate the respective feature's code from its commits. Then, we used scripts and Git commands to download the change history of every modified file, collecting any commit potentially related to the feature's code. We manually inspected these files to identify whether a commit actually modified the feature code under inspection. Since this could involve many commits as well as files, and file renames are (inherently) not perfectly captured by Git, we also collected unique keywords in each feature's pull-request code to search for in the commits. To handle file renaming, we utilized that the respective Git queries return an error code for files that they do not find (i.e., that have been renamed and thus removed). In these cases, we queried the added files in that commit (i.e., the new names) and manually inspected whether any of these represented the old files. For each change, we documented the date of the commit and its title in a document for the respective pull request.

Inspecting Commits. While there are many techniques that attempt to automatically classify the intentions of commits (Krüger et al., 2024; Mauczka et al., 2012; Levin and Yehudai, 2017), these are often focused on certain pieces of commit information and specific categories; such as the maintenance activities proposed by Swanson (1976). This causes problems for our study, because these techniques cannot consider the context between commits, pull requests, and features. As a consequence, they cannot distinguish between what parts of a (tangled) commit are relevant for a feature, for a different feature, or represent an unrelated activity (e.g., a refactoring). In turn, the derived tags would most likely not be the ones we are interested in. Moreover, the categorizations are often rather abstract (e.g., perfective), and do not detail what the developers actually intended to do.

For these reasons, we decided to manually inspect each commit, examining its message and code changes to identify its intention related to a feature. As an example, a feature to cool lasers has been enhanced (tag: **Enhancement**) in one commit to expand its functionality by enabling it to track the flow of water. We assigned a single tag for each combination of commit and feature. In case multiple tags would apply, we used the one we perceived as most relevant based on the feature's description and the commit message as well as changes. This way, we tried to capture the core of a commit's change intention with respect to a specific feature. Please note that a commit can nonetheless have different tags for different features, if that commit modified multiple features (i.e., one commit can map to any number of features and their tags). More specifically, the same commit may modify two features, and thus will have two tags (potentially the same). Those multiple tags for one commit represent the difference between the total (2956) and unique (1940) commits we analyzed, and which we summarize in Table 1. We executed this process for each pull request and its related commits before moving to the next pull request.

Establishing Tags. We started our tagging using the classification proposed by Swanson (1976), but, as we suspected, it was too coarse-grained to properly capture the relations between pull requests, commits, and features. For this reason, we started to introduce more detailed tags, employing an open coding to derive these tags. Finally, we ended up with the following 11 tags to code a commit's intentions:

1. **New Feature** describes a commit that introduces a feature into the code. Interestingly, such commits varied widely in terms of size. As extreme cases, pull request 26825[5] involves a single line of code to define two pins on the control board to enable a new feature. In contrast, pull request 18071[6] introduced 16,162 lines of code for a new UI library of a specific printer.

2. **Removal** specifies that a feature is removed (i.e., all lines are deleted without being added somewhere else), signaling the end of life of that feature. One example is pull request 24229,[7] which removes the feature introduced in pull request 24074.[8]

3. **Rework** documents that a commit essentially re-implements a feature. For instance, a feature for laser graphics on LCD screens introduced in pull request 16068[9] was completely reworked in pull request 15335.[10]

4. **Revert** means that a commit undoes a previous commit, which was happening especially when larger changes were merged but contained many bugs.

5. **Bug Fix** commits correct unintended or erroneous behavior of Marlin.

6. **Enhancement** specifies that a commit adds functionality to a feature or broadens its support on the firmware. For example, pull request 26485[11] changes the code of the feature introduced in pull request 26328[12] to make it work on another platform.

7. **Refactor** documents that a commit changes the code of a feature, for instance, to optimize performance, but does not alter its functionality.

8. **Cleanup** commits remove code of a feature but do not alter its functionality (e.g., removing dead code).

9. **Formatting** means that a commit changes only the spacing of the code to change its layout.

10. **Comment** describes that a commit changes only comments, but no actual code.

11. **Whitespace** commits change only whitespaces.

[4] https://github.com/MarlinFirmware/Marlin/pull/21255.

[5] https://github.com/MarlinFirmware/Marlin/pull/26825.
[6] https://github.com/MarlinFirmware/Marlin/pull/18071.
[7] https://github.com/MarlinFirmware/Marlin/pull/24229.
[8] https://github.com/MarlinFirmware/Marlin/pull/24074.
[9] https://github.com/MarlinFirmware/Marlin/pull/16068.
[10] https://github.com/MarlinFirmware/Marlin/pull/15335.
[11] https://github.com/MarlinFirmware/Marlin/pull/26485.
[12] https://github.com/MarlinFirmware/Marlin/pull/26328.

Overall, the first seven tags (1. to 7.) are the most important changes to us, since they modify the feature code or its behavior. The last four tags (8. to 11.) are behavior-preserving adjustments, which we still tagged for completeness. Again, please note that we refer to *label* if we mean the developer specified labels on GitHub, and **tags** if we refer to the above tags we assigned to commits.

Validating Tags. Any qualitative and manual analysis of software changes with their respective developer comments is prone to subjective interpretations. Aiming to keep the tagging consistent, the first author performed a full tagging of all commits regarding the features involved. For this purpose, the first author started by performing an initial tagging, discussing the respective findings and problematic cases with the last author. Then, the first author continued with the tagging independently, again clarifying any occurring problems or potential confusions (e.g., regarding the types of tags) with the last author. To identify subjectivity bias and assess to what extent our data is reasonable, the third author performed an independent validation in the end. Note that the third author was not involved in the initial tagging or discussions of problematic cases, so that they could execute a fully independent assessment. For this purpose, the third author picked a random sample of 100 tagged commits (6%) from our dataset, inspected them on GitHub, and tagged them themselves according to the strategy described above. Overall, the third author fully agreed with 85 of the first author's tags and disagreed with seven of the tags. For eight commits, the tagging differed for slight nuances between **Enhancement** and **Refactor**, but we considered both tags as reasonable. Thus, the first and third author agreed on 93% of the commits, improving our confidence in the reliability of the tagging.

2.4. Feature-labeling reliability (RQ_3)

To check the reliability of the feature labels defined by the Marlin developers and their alignment to the definition of a feature by Apel et al. (2013), we compared the 189 labeled pull requests we collected for RQ_2 to that definition. We also aimed to check whether other pull requests for Marlin introduce features according to the definition by Apel et al., but without being labeled as such. This comparison allows us to reason about the context of our findings and to understand different notions that exist among developers.

For this purpose, we first collected all pull requests spanning the same time period as our previous sample (i.e., June 11th, 2019–July 28th, 2024). Then, we removed all pull requests that are labeled as *PR: New Feature* or were not merged. Furthermore, we excluded all pull requests that were labeled as bug fixes or clean ups, which according to our insights were not connected to new features. From the remaining dataset, we identified the six labels from our previous analysis that are most commonly associated with our tag **New Feature** (number of remaining pull requests between parentheses): *PR: Improvement* (765), *C: LCD & Controllers* (472), *A: STM32* (174), *C: Calibration* (110), *C: Motion* (89), and *C: Peripherals* (81). For each of these six labels, we randomly selected 30 of the remaining pull requests, resulting in a total of 180 pull requests, and thus a similar sample size as the feature-request one (i.e., 189). Finally, we inspected each of these pull requests with its associated artifacts (e.g., commits, code, discussions, messages) to decide whether we would consider it to represent a new feature or not.

3. Results and discussion

In the following, we present and discuss our results for each research question individually.

3.1. Marlin labeling practices (RQ_1)

We finalized our data collection on July 28th, 2024. At this point, the Marlin repository contained 12,528 pull requests, of which 9114 were merged and 3414 were not merged. We found a total of 662 merged pull requests labeled *PR: New Feature*.

Marlin Labels ($RQ_{1.1}$). In total, we found 80 labels for issues and pull requests in Marlin. Unfortunately, the Marlin documentation we found did not explain all of them. To obtain a better understanding of the system and its pull requests, we inspected each label, the documentation, respective pull requests, and their associated code changes. Explanations for all labels are in our dataset,[2] but essentially Marlin uses nine prefixes to categorize labels (cf. (6) in Fig. 1). Unfortunately, these prefixes were also not explained in the documentation, and five labels had no prefix. Through our inspection, we derived the meanings of the prefixes as follows (numbers indicate how many pull requests with the label exist in total in Marlin):

A: (498) represents *Architecture*, since it is used in combination with labels linked to different microcontrollers.

Bug/Bug? (425) are used to label issues and pull requests on known or suspected software *Bugs*.

C: (4656) stands for *Code* and is used to refer to general topics on the Marlin codebase.

F: (1246) means *Feature*, and is used to call out an existing feature of Marlin.

K: (73) refers to *Kinematics*, and thus to the specific motion systems used in Marlin.

Needs: (455) is used to indicate that administrative tasks must be addressed for a pull request.

PR: (6890) stands for *Pull Request* and specifies the type of pull request, such as *PR: New Feature* for introducing a feature.

S: (350) has remained vague to us. Some of the labels describe actions concerning a pull request, while others describe the state of a pull request, which would align to the abbreviation.

T: (752) stands for *Topic*, and the respective pull requests are not necessarily related to the code of Marlin.

Moreover, 4656 pull requests had no label and 14 had a label that did not fit into this categorization.

Labeling Practices ($RQ_{1.1}$). In our understanding, the prefix-based distinction of categories has become quite established and clear within Marlin. Today, most pull requests that get merged have at least one label with the prefix "*PR:*." For example, pull request 26979[13] has a label *PR: Bug Fix* and revolves around fixing a typo. We also observed that the Marlin maintainers started labeling a majority of the pull requests in 2015. Since then, the ratio of new pull requests without labels has decreased. Overall, the most common labels are *PR: Bug Fix* (3105), *PR: Improvement* (2351), *C; LCD & Controllers* (1416), *C: Board/Pins* (1006), *PR: General Cleanup* (846), and *PR: New Feature* (662). We remark that we noticed inconsistencies regarding these numbers between the GitHub API (which we used) and the GitHub UI. In Section 5, we discuss these inconsistencies in more detail.

We retrieved 3146 unlabeled pull requests. Inspecting these, we noticed roughly three categories. First, there are pull requests that contain no further information and may have been mistakes, since they are quickly deleted by the contributor or a maintainer. Second, some changes proposed by contributors are discarded by the Marlin

[13] https://github.com/MarlinFirmware/Marlin/pull/26979.

maintainers, typically with an explanation. Finally, very small changes (e.g., typo fixes) are accepted quickly without labeling the pull request.

Discussion. The Marlin developers have established a clear set of labels for specifying the intentions of pull requests. Nonetheless, around 25% (3146 of 12,528) of the pull requests in Marlin do not have a label. This is due to old pull requests, very small changes being rapidly integrated, or erroneous pull requests being ignored. We argue that this underpins a well-established labeling practice and a structured development process, since most merged pull requests in Marlin today are labeled ones. Bug fixes and improvements being by far the most common labels is not surprising, since these represent typical maintenance activities. Based on our insights, we argue that establishing labels is a good practice for larger software projects with many contributors. Practitioners may utilize Marlin's experiences and practices, while it is an interesting research direction to identify what labeling practices and information may be more helpful to developers. Especially the idea of introducing a higher level categorization (prefixes) that is refined through further keywords seems to be a helpful concept. However, problems may arise if the categorization is not intuitive and not documented, hampering developers' ability to obtain knowledge they may need (Krüger and Hebig, 2024; Krüger et al., 2020b). Based on our experiences of analyzing Marlin pull requests, we argue that documentation about labeling practices and processes is important to avoid confusion and errors (e.g., considering the prefix *S:*).

$RQ_{1.1}$ Marlin's Labeling Practices

Over time, Marlin has established clear labels and labeling practices for issues and pull requests to coordinate contributions, but there seems to be no (external) documentation of these practices. Other projects with many contributors may benefit from adapting such labeling practices and documenting them for contributors.

Label *PR: New Feature* ($RQ_{1.2}$). A highly important label for our case study is *PR: New Feature*, since it should indicate that a new feature is introduced. Thus, this label represents the starting point for us to investigate a feature's life cycle. *PR: New Feature* represents around 5% (662 of 12,528) of all pull requests and around 7% (662 of 9382) of the labeled ones. After inspecting our sample of 189 pull requests labeled as *PR: New Feature* in Marlin, we are confident that these introduce new features that are either a characteristic or an end-user-visible behavior of the system, aligning to the definition by Apel et al. (2013).

In parallel, the pull requests and consequent features are very diverse, covering a wide range of functionalities. Some add configuration options to adjust the behavior of Marlin, others add commands to enable new functionalities. Consequently, some pull requests are large and complicated, while others are small and simple. For instance, pull request 14251[14] introduced a simple feature that can be enabled to leave heaters on after a print is aborted (i.e., end-user visible behavior). A larger and more complex feature was introduced in pull request 18071. It added the LVLGL GUI library for the MKS Robin Nano to Marlin. Lastly, pull request 20940[15] added a "more" menu in the user interface to allow end users to store up to seven custom commands.

Not surprisingly, we noticed that the label *PR: New Feature* occurs often in combination with other labels. Most prominently, 36% of the pull requests labeled as *S: Experimental* and 35% of those labeled *F: CNC/Laser* are also labeled as *PR: New Feature*. The former label is intuitively linked to new functionalities, often leading to a new or improved feature being introduced (e.g., pull request 3110[16] implementing an alternative for LCD-based manual movement). The latter label represents the introduction of CNC and laser capabilities into the formerly purely 3D-printing Marlin firmware. Conversely, other labels also occur often among the 662 ones labeled *PR: New Feature*, for example, *PR: Improvement* (88; 13%), *C: LCD & Controllers* (86; 13%), and *F: Calibration* (64; 10%). These results showcase that, despite the established labeling practices, changes often involve tangled or ambiguous intentions.

We further identified a label *T: Feature Request* being used in 20 pull requests. However, this label is intended for issues only and not pull requests. In fact, we did not observe it occurring in a pull request after September 26th, 2017, in pull request #7755.[17] As an important note, only six of the 20 pull requests labeled *T: Feature Request* have also been labeled as *PR: New Feature*. To cross-check, we retrieved the 35 issues with the label *T: Feature Request* that are directly linked to a pull request, of which again only six had the label *PR: New Feature*. When inspecting issues labeled *T: Feature Request* without a linked pull request, we noticed the same picture. Despite these pull requests missing the label *PR: New Feature*, this label often shows up in the comments of the pull requests. For instance, issue 24928[18] asks about adding a new option for a command to disable input shaping, which is actually introduced in pull request 24951[19] together with some other changes—being labeled as *C: Motion*, *PR Bug Fix*, and *PR: Improvement*. Lastly, we noticed that many issues labeled *T: Feature Request* did actually not propose new features, but rather bug fixes or improvements according to the maintainers of Marlin. Together with our previous insights, this underpins that relying solely on labels, code, and automated analysis to study the evolution of features can easily introduce biases and inconsistencies.

Discussion. Even if labeling practices have been established, they may still be understood differently or used inconsistently by developers. In addition, a lack of documentation and the ability of every developer to simply add multiple labels to issues or pull requests may promote their tangling. This is problematic, since the idea of such labels is to represent the intention of the involved changes and thereby ease comprehension. So, labels could serve as a means to check that only one intention is addressed in a pull requests, keeping it manageable and concise. Instead, the tangling of changes (represented by multiple labels) occurring in real-world pull requests hinders comprehension and analyses. We had to invest substantial manual effort to comprehend what code in a pull request was related to a new feature in cases where multiple labels were involved. For these reasons, we advise practitioners and contributors to avoid unnecessary tangling of different changes and intentions. Research on automated untangling of changes or ensuring that a change only addresses a single intention could help mitigate such cases (Krüger et al., 2023).

$RQ_{1.2}$ New Features and Tangled Changes

In Marlin, feature requests have often been tangled with other (labeled) change intentions, and are sometimes linked to a similar label intended for a different artifact. This reduces comprehension and challenges automated analyses, which is why such tangling should be avoided and techniques to resolve it would be helpful.

3.2. RQ_2: Feature evolution

After understanding Marlin and its labeling practices, we analyzed and tagged 2956 commits connected to the features introduced via 189 pull requests labeled *PR: New Feature*. Specifically, these 189 are the pull requests merged from June 11th 2019 until the end of our data collection (covering roughly five years). For the same period, another 39 pull requests labeled *PR: New Feature* were rejected and 14 were still open, and thus not part of our analysis.

[14] https://github.com/MarlinFirmware/Marlin/pull/14251.
[15] https://github.com/MarlinFirmware/Marlin/pull/20940.
[16] https://github.com/MarlinFirmware/Marlin/pull/3110.
[17] https://github.com/MarlinFirmware/Marlin/pull/7755.
[18] https://github.com/MarlinFirmware/Marlin/issues/24928.
[19] https://github.com/MarlinFirmware/Marlin/pull/24951.

Table 1
Overview of the tags we assigned to commits (total: tag-commit combination covering a commit multiple times if it changed different features; unique: removed the duplicate commits from total; **New Feature** + tags: number of commits with the tag **New Feature** and at least one other tag).

Tag	Commits		
	Total	Unique	**New Feature**+tags
New Feature	189	189	79
Removal	2	2	0
Rework	18	18	4
Revert	19	10	1
Bug Fix	353	321	2
Enhancement	367	316	39
Refactor	1779	902	89
Cleanup	51	44	0
Comment	101	86	12
Formatting	63	43	2
Whitespace	14	9	0
sum	2956	1940	(*uniquecommits*)79

In Table 1, we provide an overview of our tags. We show the total number of commits, how many of these are unique (i.e., not counting multiple occurrences among the 189 features), and to what extent commits with the tag **New Feature** are tangled with other tags. The numbers underpin again that changes often impact multiple features at the same time. In fact, some commits occurred in a substantial share of the 189 features. By far the largest example is the one commit of pull request 25908,[20] which involves 950 lines of code. With this change, the developers removed the two widely used macros `EITHER` and `BOTH`. This impacted 62 of the 189 features we identified from pull requests. Afterwards, the numbers drop strongly, with the next four commits touching between 26 and 21 features each. Interestingly, these four commits are not associated with a pull request themselves, but have been committed by maintainers. The scattered nature of features causes tangled changes and redesigns that cause essentially global changes—again complicating comprehension, evolution, and (functional) correctness.

By computing metrics on the commits, we found that each feature was on average (mean values) impacted by 15.6 commits (median 9, standard deviation 20). However, we also remark that this number can vary heavily between individual features. For instance, 98 features have one to nine commits associated to them, while seven connect to more than sixty commits. Not surprisingly, features with more commits also change many more lines of code. For example, the 24 features with 15 to 19 commits change on average 531 lines of code, while the seven features with 60 or more commits change around 3592 lines of code on average. Considering the actual time, we found that all 189 features together are impacted by a commit every 168 days on average. Again, there are strong differences between different features, ranging from intervals of 21 up to 398 days. This underpins the diversity and varying importance of the features in our sample.

Tags. As we can see in Table 1, we mostly tagged commits as **Refactor**, contributing roughly 60% (1779 of 2956) of all changes to features. These are also the commits that touch multiple features most often by far, namely in around 49% of the cases (877 commits occur multiple times, difference between total and unique). The tags **Enhancement** and **Bug Fix** represent the next largest shares. That these three categories occur most often in a feature's life cycle is logical, since they represent common maintenance activities.

The tag **New Feature** is quite intuitive, and we essentially tagged only the one commit that actually introduced a feature into Marlin with it (i.e., the merge of the pull requests labeled *PR: New Feature*). Conversely, the most interesting tag to us is **Removal**, which indicates

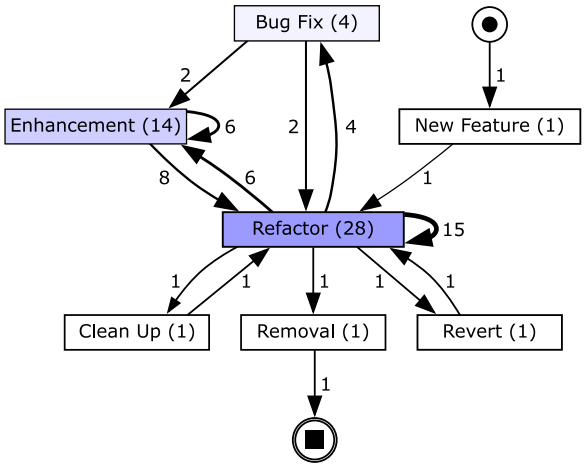

Fig. 2. DFG of pull request 16452 (https://github.com/marlinfirmware/marlin/pull/16452).

that a feature was actually removed from Marlin. We expected this to be a rare case, and it only occurred twice across all 189 features. First, pull request 24229[21] removed a feature on the same day it was introduced (after a bit more than five hours).[22] Essentially, this was a revert; however, the feature was formally introduced and then removed because it did not work as intended on Apple's MacOS. Since then, it seems that the feature has not been reintroduced into Marlin. Second, pull request 24427[23] removed the support for a series of stepper drivers.[24] Interestingly, that feature was a re-implementation of a previous feature that was specific to certain drivers and generalized these to work on the entire family of stepper drivers.[25] Lastly, there was one more commit that dropped support for a platform from Marlin.[26] However, this did not remove a feature, but only parts of its functionality.

Feature Life Cycles. For analyzing the evolution of all 189 features we identified, we performed process mining on the tagged commit traces of every feature using the tool PM4PY (Berti et al., 2023). The resulting *Directly Follows Graphs* (DFGs) (van der Aalst, 2019) provide a visualization of the life cycles of sets of features. Here, nodes represent tagged activities and edges the transitions between them, with the numbers indicating how often we observed each node and transition for that feature. As one example from our dataset, we depict the DFG of pull request 16452[24] in Fig. 2. We can see that the DFG is centered around the tag **Refactor**, which represents 28 of the 50 commits touching the feature. Further, we can see that refactoring is the only type of commit connected to every other type of commit and often transitions to itself. Consequently, the life cycle of this feature is connected primarily to refactorings following refactorings. This is in line with the general ratio of refactorings (cf. Table 1) and life cycles of other features. The life cycle of this feature slightly deviates from the average regarding enhancements (overrepresented), bug fixes (underrepresented), and its removal (exceptional). This feature also includes a commit tagged as **Revert**, a tag that is often connected to hardware. In this case, the hardware abstraction layer structure was refactored to apply the singleton design pattern, but then reverted due to that solution having too many problems as well.

We further created a synthesis of all 189 DFGs, which we display in Fig. 3. Please note that we have simplified this figure for readability by

[20] https://github.com/MarlinFirmware/Marlin/pull/25908.

[21] https://github.com/MarlinFirmware/Marlin/pull/24229.
[22] https://github.com/MarlinFirmware/Marlin/pull/24074.
[23] https://github.com/MarlinFirmware/Marlin/pull/24427.
[24] https://github.com/MarlinFirmware/Marlin/pull/16452.
[25] https://github.com/MarlinFirmware/Marlin/pull/13498.
[26] https://github.com/MarlinFirmware/Marlin/pull/20153.

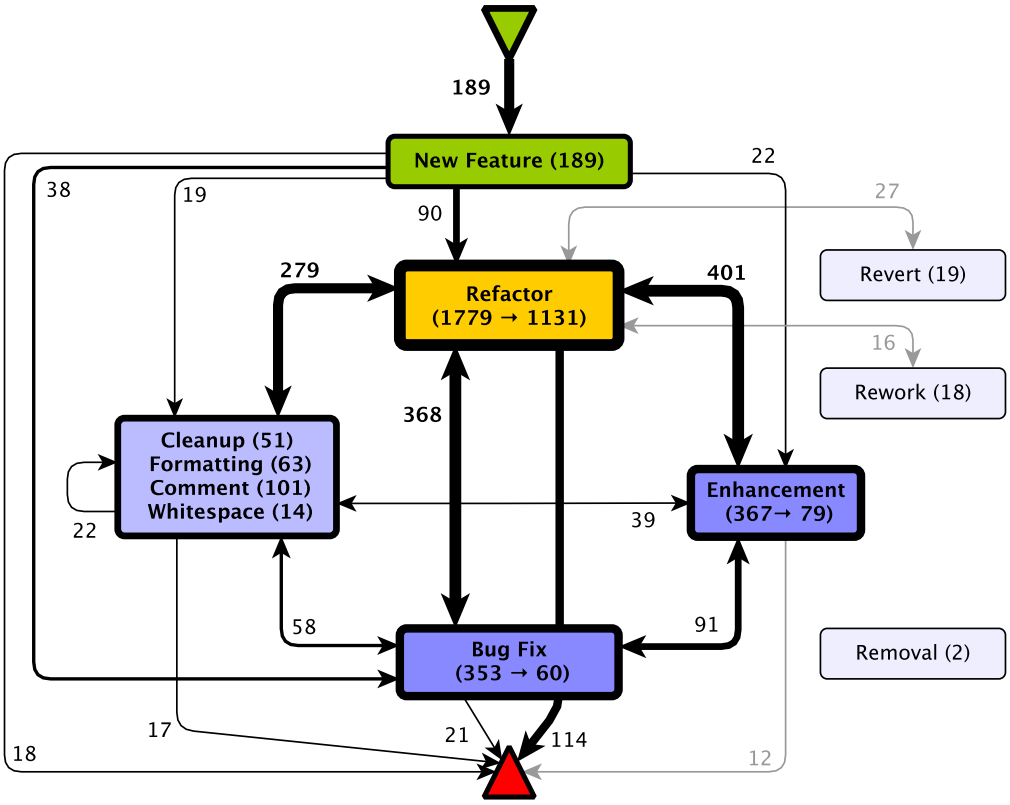

Fig. 3. Synthesis of all DFGs from process mining the 189 pull requests labeled *PR: New Feature*. We have removed transitions that occurred only once for readability. For the same reason, we have merged the four tags that represent commits with behavior-preserving adjustments (8.–11. in Section 2.3). The thicker arrows are, the more subsequent commits have the target tag. To improve readability, we omit large self-loops and indicate the number of consecutive equivalent tags in parentheses: $(x \rightarrow y)$ stands for x tags from which y are not changing.

removing all transitions occurring only once and merging the commits representing behavior-preserving adjustments (i.e., **Cleanup**, **Formatting**, **Comment**, **Whitespace**). In general, we can see that features are primarily refactored or otherwise maintained, exhibiting a typical software life cycle. More precisely, the features we analyzed were mostly changed through refactoring, with around two thirds of the refactorings also following another refactoring. Interactions between other tags and refactoring are those that include enhancements, bug fixes, or behavior-preserving adjustments. Other types of change intentions and transitions between them were much rarer.

This provides a key insight and implication for the developers of Marlin. If a feature is not affected by many refactorings, that feature deviates from a major pattern exhibited for typical Marlin features. Hence, the developers may have to take a closer look at this feature to either include it in future refactorings, remove the feature as obsolete, or simply verify that everything is in order. Likewise, deviations from this typical life cycle provide interesting cases for practitioners and researchers alike. They represent outliers that may be more challenging for practitioners and may help identify quality problems that exist in features. In particular, reverts, reworks, and removals occurred rarely. As we discussed before, reverts typically happen right after another commit in cases when bugs were introduced. Studying such cases may be helpful to identify typical errors developers make when evolving features. Reworks of features can shed light into why and how features are modernized. Lastly, removals are very rare (occurring only twice), but highly interesting. To the best of our knowledge, deprecating or removing features (or entire product lines) has rarely been studied (Cortiñas et al., 2023). Open-source systems with feature removals are a great opportunity to study this phenomenon outside of confidential industry collaborations to shed light into the causes and consequences.

Discussion. Features are constantly evolving. Typically, this involves maintaining them with few deviations. However, particularly these deviations are the most interesting cases for research and likely developers, too. For instance, reworks or removals are rare cases, but understanding their causes could be very helpful for practitioners and guide the development of automated detection as well as support techniques. This also means that features are typically long-living, which, however, may be specific to Marlin, embedded software, or open-source software. The features we analyzed for Marlin were mostly implemented and then maintained; we found few indications of substantial changes like feature removals or larger reworks. Thus, it is an interesting direction for future research to analyze whether features in other variability-rich systems or product lines also remain relatively stable after these have been developed and cover the domain well.

RQ_2: How Marlin's Features Evolve

After their introduction, the Marlin features we analyzed were primarily maintained, most prominently refactored followed by enhancements, bug fixes, and behavior-preserving adjustments. In contrast, they were rarely reworked or removed after their introduction, but these are interesting cases for future research to elicit problems and derive recommendations for practitioners.

3.3. RQ_3 : *Reliability of feature labels*

For RQ_3, we wanted to reflect on the reliability of feature labels, especially because the notion of a feature is not well-defined, and thus developers can have varying understandings of what a feature is Classen et al. (2008) and Berger et al. (2015). It is important to elicit these understandings to identify agreements and disagreements. By doing so, we can develop a unified notion, while also identifying gaps between research and practice that may pose validity threats to feature-oriented research.

Due to Marlin's established labeling practices, it was not surprising that we considered all 189 pull requests labeled as *PR: New Feature* to represent features. Conversely, we also inspected a sample of 180 pull requests that were not labeled as *PR: New Feature*, but whose tags were otherwise typically associated with that label (cf. Section 2.4). Specifically, we randomly picked 30 pull requests for each of the following labels (with total numbers of pull requests from which we sampled): *PR: Improvement* (765), *C: LCD & Controllers* (472), *A: STM32* (174), *C: Calibration* (110), *C: Motion* (89), and *C: Peripherals* (81). As we explained in Section 2.4, we previously filtered out pull requests labeled *PR: New Feature*, *PR: Bug Fix*, or *PR: General Cleanup*. In the following, we discuss our findings of investigating these 180 pull requests, particularly regarding whether they introduce new features according to our understanding of the definition by Apel et al. (2013).

PR: Improvement. We would consider 14 of the 30 pull requests we analyzed for this label to propose a feature. As an example, pull request 15394[27] adds support for the M997 command on the STM32 platform. This allows Marlin to flash firmware on those boards. Pull request 23764[28] adds support for probe temperature compensation to all commands for which this would be useful. In contrast, pull request 22504[29] fixes a problem that occurred after changing tooling, which could cause unexpected extruder moves. Interestingly, while this is a bug fix of unintended behavior (and clearly not the introduction of a new feature), the pull request was not labeled as *PR: Bug Fix*.

C: LCD & Controllers. After inspecting them, we would consider 13 of the 30 pull requests to propose a new feature, despite missing the label. For example, pull request 18326[30] introduces a feature for the MKS UI to display the remaining time after using the M73 command on the screen—which is end-user-visible behavior. Similarly, pull request 26596[31] introduces the I3DBEE TECH Beez Mini 12864 screen to Marlin. In contrast to these two, pull request 15498[32] cleans up the function `LCDPRINT` and removes nonfunctional code. This is a cleanup, and not introducing a new feature.

A: STM32. We would consider 14 of the 30 pull requests to propose a new feature. An example is pull request 14539,[33] which allows users to configure pins for stepper drives through their own configuration. Before, these pins were hard-coded in the Marlin source files. Pull request 24760[34] adds support for the Creality V5.2.1 control board to the firmware. In contrast, pull request 22537[35] simplified `#ifdef` directives, but without changing any behavior of the system. Thus, we considered this pull request to be a refactoring.

C: Calibration. We would consider 12 of the 30 pull requests of this label to propose a new feature. For instance, pull request 15376[36] introduced the command M290 to report the current printer status via serial, meaning that the respective printer does not need a screen. Pull request 23033[37] adds new features for probe temperature compensation so that it can function with more probes and allows configuring. The different probes represent individual features that become characteristics of the system. In contrast, pull request 22657[38] improves the usability of the tramming wizard. This is done by clearing up text and making values less ambiguous, but is not adding a feature.

C: Motion. We would consider 10 of the 30 pull requests with this label to propose a feature. For example, pull request 18736[39] enables users to define an explicit sequence for the nozzle wipe. This is end-user-visible behavior as Marlin will perform a specific sequence of physical actions. Pull request 24684[40] adds support for advanced linear movement to ESP32 boards. In contrast, pull request 18342[41] fixes a bug on the CoreXY platform caused by duplicate code. So, this pull request fixed a feature that was already introduced, but did not propose a new one. Interestingly, this pull request was also not labeled as *PR: Bug Fix*.

C: Peripherals. We would consider 16 of the 30 pull requests to propose a feature. For instance, pull request 14667[42] introduces the ability to control two separate strips of neopixel LEDs at the same time. Pull request 26163[43] adds a feature to re-initialize the power supply, which allows to handle cases in which screens get corrupted. In contrast, pull request 21811[44] fixed that Marlin could not be built with `CASE_LIGHT_USE_RGB_LED` enabled. So, this pull request fixed a build error, but it was also not labeled as *PR: Bug Fix*.

Discussion. Based on our manual inspection, we argue that if the Marlin developers assign a label to a pull request, that label is typically reliable. In fact, all 189 feature requests we studied were features from our point of view. However, it seems that many pull request miss relevant labels, as we have exemplified above not only for new features but also bug fixes. Of course, the notion of a feature is somewhat vague, subjective, and differs between developers (Classen et al., 2008; Berger et al., 2015). So, our numbers for new features may differ when asking the original developers or other researchers. Nonetheless, we considered 79 of the 180 sampled pull requests (≈44%) to propose a feature. Even if others would not consider all of these to represent feature requests, a considerable number of pull requests missing relevant labels would remain. This emphasizes that it is important to fully capture what different developers consider to represent a feature to agree on a common understanding. Otherwise, confusions or missing labels may occur, not only challenging the developers, but also researchers trying to research features or other changes based on developers' labeling and according to their understanding.

Something we noticed again is that the Marlin developers are apparently not consistent with their labeling of pull requests. For instance, some pull requests labeled *A: STM32* add control boards to Marlin and are not labeled as *PR: New Feature*. However, there are examples of pull requests, such as 20711,[45] that are labeled *PR: New Feature* and add support for boards, too. This is not consistent and threatens manual as well as automated analyses. Identically, we already exemplified that bug fixes are often not labeled as such. Of the 180 pull requests we sampled, this actually accounts to 31 pull requests.

RQ_3: Marlin Features and Scientific Definition

Labels assigned to Marlin's pull requests seem reliable, and thus feature requests are representing new features. On the contrary, pull requests often miss relevant labels, meaning that their labels are incomplete. Particularly for missing feature-request labels, a problem seems to arise from lacking a precise definition of a feature and developers having different notions about it. Research utilizing such labels and automated analyses must take missing labels into account to avoid threats.

[27] https://github.com/MarlinFirmware/Marlin/pull/15394.
[28] https://github.com/MarlinFirmware/Marlin/pull/23764.
[29] https://github.com/MarlinFirmware/Marlin/pull/22504.
[30] https://github.com/MarlinFirmware/Marlin/pull/18326.
[31] https://github.com/MarlinFirmware/Marlin/pull/26596.
[32] https://github.com/MarlinFirmware/Marlin/pull/15498.
[33] https://github.com/MarlinFirmware/Marlin/pull/14539.
[34] https://github.com/MarlinFirmware/Marlin/pull/24760.
[35] https://github.com/MarlinFirmware/Marlin/pull/22537.
[36] https://github.com/MarlinFirmware/Marlin/pull/15376.
[37] https://github.com/MarlinFirmware/Marlin/pull/23033.
[38] https://github.com/MarlinFirmware/Marlin/pull/22657.
[39] https://github.com/MarlinFirmware/Marlin/pull/18736.
[40] https://github.com/MarlinFirmware/Marlin/pull/24684.
[41] https://github.com/MarlinFirmware/Marlin/pull/18342.
[42] https://github.com/MarlinFirmware/Marlin/pull/14667.
[43] https://github.com/MarlinFirmware/Marlin/pull/26163.
[44] https://github.com/MarlinFirmware/Marlin/pull/21811.
[45] https://github.com/MarlinFirmware/Marlin/pull/20711.

4. Recommendations and prospects

To briefly summarize our key findings and their implications for practitioners (P) and researchers (R):

P: Establish and Document a Labeling Strategy. Marlin uses an extensive labeling strategy, building on 80 labels and nine prefixes. Since this has evolved over time with Marlin becoming larger, more established, and maintained by a growing community, we argue that a labeling strategy is important to manage growing software and communities. Specifically, we argue that practitioners can take Marlin's labels with its prefixes as a starting point for defining their own strategy (cf. Section 3.1). However, we also found various inconsistencies in the labeling, which we argue relates to a lack of documentation and awareness. Consequently, we strongly advise practitioners to document and share their labeling strategy with each other, for which they should avoid typical knowledge-sharing pitfalls identified in previous research (Krüger et al., 2020b; Steinmacher et al., 2015; Riege, 2005; Krüger and Hebig, 2024). In turn, labels of issues and feature requests would be more reliable. This would benefit practitioners by avoiding confusions and making it easier to, for example, identify and assign tasks (e.g., good first issues). For researchers, an established, documented, and reliable labeling strategy would support manual analyses and enable reliable automation. Otherwise, using automation on labels will pose a threat to validity.

P/R: Agree on a Notion of Feature. Despite widespread use, the notion of a feature is still not well-defined and most practitioners as well as researchers have a varying understanding depending on their intuition (cf. Section 3.3. This can cause misunderstandings among developers, result in missing labels, and challenge (automated) analyses. For practitioners, it is therefore helpful to agree on a notion among each other. By unifying the notion of features, it will become easier for practitioners to communicate and to onboard to new projects. As steps in this direction, we advise developers to collaboratively identify and document what a feature should represent for them (e.g., domain abstraction versus configuration options (Nešić et al., 2019)), a definition that applies in this context, and to scope what and what not a feature entails (e.g., code, models, requirements). Particularly the boundaries of one feature to another are important to clarify and document. To support practitioners, researchers have to further explore how to unify the notion of features, particularly to align their work with practitioners' needs and understandings. Otherwise, any study on features that goes beyond configurable code can easily be threatened by conflicting understandings and the practical relevance of the research may be limited.

P/R: Untangle Changes. The changes we analyzed often involved multiple intentions at once, a well-known problem in research (Krüger et al., 2024). It essentially makes it impossible to execute automated analyses reliably, and threatens our understanding of developers' work. Moreover, we argue that this is also a problem for practitioners. For instance, if a developer submits a pull requests to add a feature that also involves refactorings and optimizations, other developers (e.g., reviewers) will find it more difficult to understand the change. Also, reviewers may reject the changes due to the tangling, thus causing additional workload for developers and maintainers alike. A problem in this regard is the lack of a usable definition of change intentions. Existing proposals are often rather abstract (Krüger et al., 2024), and we found them infeasible for our purpose. For this reason, we defined more fine-grained and hopefully reusable tags for research and practice (cf. Section 2.3) that can serve as a foundation for refining labeling strategies and scoping changes to untangle them.

R: Reflect on Feature Refactoring. We found that features are mostly changed due to refactorings, followed by enhancements, bug fixes, and behavior-preserving changes (cf. Section 3.2). There are very few cases in which larger or more drastic changes are implemented, such as feature removals, reworks, and reverts. This situation implies to us that features seem to be often redesigned to make them more comprehensive or maintainable. In turn, we are wondering whether all of these refactorings were necessary if practitioners would have (or would be aware of) design principles, patterns, or best practices for designing features. Thus, for researchers, several avenues for future work arise, for instance, identifying the causes and consequences of feature refactorings, performing focused studies on more drastic changes, and defining as well as communicating design practices for features. This could lead to helpful guides for practitioners, for example, criteria that help decide whether a feature needs to be removed or reworked.

R: Be Careful with GitHub Analyses. There can be inconsistencies between GitHub API and UI (cf. Section 5). So, when using these, researchers have to be careful and reflect on potential threats arising from, for instance, deleted data being retrieved. This is a serious threat to mining studies that rely on author information, in which cases "ghost" (i.e., deleted accounts that still account for contributions to a project) accounts pose a problem.[46]

We hope that these insights are valuable for researchers and practitioners alike, guiding communication in developer communities and future research.

5. Threats to validity

Marlin is an open-source project implemented with C and C++. Moreover, it is embedded software that runs on very specific hardware: 3D-printers and related devices. So, the external validity of our work is limited, but this is an inherent limitation of a case study. However, inherently, a case study does not focus on generalizability, but on obtaining detailed, in-principle transferable insights into a phenomenon (Yin, 2018; Ralph et al., 2021). Consequently, our case study is also not generalizable and limited regarding its external validity. Instead of focusing on external validity, we focus on transferability: obtaining insights that can be useful for other developers and systems. By inspecting Marlin as a substantial case in detail, we argue that we achieved this goal and that our findings can be helpful (in adjusted form) in other contexts as well.

The results of our case study depend on our interpretation of features (and our versus the developers' definition of a feature) and other developers' code as well as natural-language texts. While we proceeded with care and used continuous checks, discussions, as well as a validation, it remains a threat that we may have misunderstood something, and thus derived incorrect data. Similarly, our case study is inherently incomplete due to the sheer size of the available data. Some of our results may have been different if we would have considered more feature requests or a longer time period. We aimed to mitigate such internal threats through a thorough manual analysis of a large temporary sample of pull requests.

Lastly, during our study, we noticed differences between the data we retrieved from the GitHub API and what is reported in the GitHub UI. For instance, the UI reported 652 pull requests with the label *PR: New Feature*, while the GitHub API retrieved 662. In fact, of the 10 labels occurring most often in Marlin, not a single one had the same number of pull requests in the API and the UI. As one example, we could not find pull request 4811[47] in the UI. We could only access this

[46] https://github.com/ghost.

[47] https://github.com/MarlinFirmware/Marlin/pull/4811.

pull request by manually filling in the URL in a browser. In this case, the problem seems to be that the pull request involves a "ghost" account. A ghost account[46] represents an account that has been deleted, which also deletes any data connected to that account's repositories (aligning to privacy and data-protection regulations). However, contributions (pull requests, issues, comments) to the repositories of others are kept, and apparently (partly) hidden in the UI—but not in the API. Similarly, we experienced that some pull requests were listed in the UI, but it was not possible for us to open them. While this seems to be a temporary problem, we still noticed that the GitHub API was more reliable and retrieved all data. We are not aware of a previous study reporting this issue, which is why we elaborate it here.

The differences between GitHub API and UI are important to keep in mind, since they may threaten the internal validity. Developers have the right to remove their personal data (and accounts) from social-coding platforms, and the platform's host as well as researchers have to respect these rights (Broneske et al., 2024). Apparently, the API is still reliably recovering anonymized artifacts contributed to others' repositories, and thus should be used to retrieve data. However, it is important to note that according to GitHub's policies, an unknown number of developers may be referred to as "ghosts," and thus author-based metrics can be biased.

6. Related work

Software features are extensively researched in the areas of product-line engineering and variability-rich systems, as well as connected topics like configuring, feature models, or feature forks (Apel et al., 2013; Liebig et al., 2010; Mortara and Collet, 2021; Benavides et al., 2010; Czarnecki et al., 2012; Nešić et al., 2019; Stănciulescu et al., 2015; Zhou et al., 2018). In this context, the problem that features are not well-defined has been raised repeatedly, with several attempts at coming up with a more unified view on this notion (Apel et al., 2013; Classen et al., 2008; Berger et al., 2015). Still, such works also show that developers have varying notions and a fully unified definition is hard to achieve. For this reason, existing studies on feature evolution typically focused on optional features (i.e., configuration options) and code metrics, sometimes in connection with other artifacts like feature models (Fischer, 2021; Passos et al., 2016; Kröher et al., 2018; Schulze et al., 2023; Ludwig et al., 2019). For instance, we have previously performed manual feature identification and feature location on Marlin to compute code metrics and to explain static feature facets (Krüger et al., 2019b, 2018). Another study investigated forking of Marlin, including the identification of feature forks (Stănciulescu et al., 2015). The focus on configurable features, forked variants, and code make it easier to elicit hard metrics, but they ignore the fact that features represent an abstract notion in developers' minds. Our case study complements such research by investigating the evolution of features, including mandatory ones, on the level of change intentions (Krüger et al., 2023, 2024). So, in contrast to most other works, we are not interested in code metrics, but the intentions for which features evolve.

In relation to parts of our findings, many researchers have identified and studied the issue of commits tangling different concerns (Herzig et al., 2016; Kirinuki et al., 2014; Dias et al., 2015; Sothornprapakorn et al., 2018; Krüger et al., 2024, 2023). For example, Herbold et al. (2022) contribute a dataset of bug fixes (i.e., the concern) that are tangled with other changes. Herbold et al. motivate their work by the fact that if researchers build on tangled changes, their findings may not actually study the intended concern but a tangled one. The authors estimate that a large share (up to 47%) of commits labeled as bug fixes may also involve changes with other intentions. We observed the same problem, which is why we introduced our tagging strategy and employed a manual analysis. Identically to the related work we also advise to untangle changes to support researchers and practitioners. So, we complement this previous work by focusing on a different concern: software features.

7. Conclusion

In this article, we reported the results of an exploratory case study of 189 feature requests and 180 additional pull requests in Marlin. We found that Marlin developers follow a defined labeling process to coordinate. However, feature requests and changes to features are often tangled with different intentions and other features, which can complicate the comprehension of how a feature evolves. Regarding their life cycles, features are primarily maintained, including refactorings, enhancements, and bug fixes. It seems rare that they are functionally reworked or removed. Regarding the labeling of features, we found that the label *PR: New Feature* seems very accurate, but quite some pull requests that seem to involve features miss it. Together with inconsistencies we found between the GitHub API and UI (cf. Section 5), this causes hurdles for reliable (automated) analyses.

To tackle such problems, we advise developer communities to agree on a definition of features, to define and document a respective labeling strategy, and to keep changes with different intentions separated from each other. For researchers, we found that the problems we identified can threaten studies, particularly because (feature) labels are not fully reliable. However, by inspecting the evolution of features on the level of intentions, we have also identified interesting changes to focus on in future research (i.e., removals, reworks, reverts) to derive quality criteria and guidelines for practice.

In the future, we want to substantiate our findings by expanding our study to other systems and longer time periods. To achieve this, feasible automated analyses and mining techniques are an important means. This case study and our dataset provide the foundations for developing such techniques. Lastly, we have focused on how features evolve, categorizing different types of changes. We plan to investigate the underlying causes in more detail, for which larger analyses and a focus on specific changes (e.g., removals) across different systems are necessary.

CRediT authorship contribution statement

Aron van der Hofstad: Writing – review & editing, Writing – original draft, Visualization, Validation, Methodology, Investigation, Conceptualization. **Loek Cleophas:** Writing – review & editing. **Clemens Dubslaff:** Writing – review & editing, Visualization. **Jacob Krüger:** Writing – review & editing, Validation, Supervision, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

A link to a persistent Zenodo repository with our data is in the article.

References

Apel, S., Batory, D., Kästner, C., Saake, G., 2013. Feature-Oriented Software Product Lines. Springer, http://dx.doi.org/10.1007/978-3-642-37521-7.

Benavides, D., Segura, S., Ruiz-Cortés, A., 2010. Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35 (6), 615–636. http://dx.doi.org/10.1016/j.is.2010.01.001.

Berger, T., Lettner, D., Rubin, J., Grünbacher, P., Silva, A., Becker, M., Chechik, M., Czarnecki, K., 2015. What is a feature? A qualitative study of features in industrial software product lines. In: International Software Product Line Conference . SPLC, ACM, pp. 16–25. http://dx.doi.org/10.1145/2791060.2791108.

Berti, A., van Zelst, S., Schuster, D., 2023. PM4Py: A process mining library for Python. Softw. Impacts 17, 1–7. http://dx.doi.org/10.1016/j.simpa.2023.100556.

Broneske, D., Kittan, S., Krüger, J., 2024. Sharing software-evolution datasets: Practices, challenges, and recommendations. Proc. ACM Softw. Eng. 1 (FSE), http://dx.doi.org/10.1145/3660798.

Classen, A., Heymans, P., Schobbens, P.-Y., 2008. What's in a feature: A requirements engineering perspective. In: International Conference on Fundamental Approaches To Software Engineering . FASE, Springer, pp. 16–30. http://dx.doi.org/10.1007/978-3-540-78743-3_2.

Cortiñas, A., Krüger, J., Lamas, V., Luaces, M.R., Pedreira, O., 2023. How to retire and replace a software product line. In: International Systems and Software Product Line Conference. SPLC, ACM, pp. 275–286. http://dx.doi.org/10.1145/3579027.3609004.

Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wąsowski, A., 2012. Cool features and tough decisions: A comparison of variability modeling approaches. In: International Workshop on Variability Modelling of Software-Intensive Systems . VaMoS, ACM, pp. 173–182. http://dx.doi.org/10.1145/2110147.2110167.

Dias, M., Bacchelli, A., Gousios, G., Cassou, D., Ducasse, S., 2015. Untangling fine-grained code changes. In: International Conference on Software Analysis, Evolution and Reengineering . SANER, IEEE, pp. 341–350. http://dx.doi.org/10.1109/saner.2015.7081844.

Fischer, S., 2021. A case study on the evolution of configuration options of a highly-configurable software system. In: International Conference on Software Analysis, Evolution and Reengineering . SANER, IEEE, pp. 630–635. http://dx.doi.org/10.1109/saner50967.2021.00079.

Gousios, G., Pinzger, M., van Deursen, A., 2014. An exploratory study of the pull-based software development model. In: International Conference on Software Engineering . ICSE, ACM, pp. 345–355. http://dx.doi.org/10.1145/2568225.2568260.

Herbold, S., Trautsch, A., Ledel, B., Aghamohammadi, A., Ghaleb, T.A., Chahal, K.K., Bossenmaier, T., Nagaria, B., Makedonski, P., Ahmadabadi, M.N., Szabados, K., Spieker, H., Madeja, M., Hoy, N., Lenarduzzi, V., Wang, S., Rodríguez-Pérez, G., Colomo-Palacios, R., Verdecchia, R., Singh, P., Qin, Y., Chakroborti, D., Davis, W., Walunj, V., Wu, H., Marcilio, D., Alam, O., Aldaeej, A., Amit, I., Turhan, B., Eismann, S., Wickert, A.-K., Malavolta, I., Sulír, M., Fard, F., Henley, A.Z., Kourtzanidis, S., Tuzun, E., Treude, C., Shamasbi, S.M., Pashchenko, I., Wyrich, M., Davis, J., Serebrenik, A., Albrecht, E., Aktas, E.U., Strüber, D., Erbel, J., 2022. A fine-grained data set and analysis of tangling in bug fixing commits. Empir. Softw. Eng. 27, 1–49. http://dx.doi.org/10.1007/s10664-021-10083-5.

Herzig, K., Just, S., Zeller, A., 2016. The impact of tangled code changes on defect prediction models. Empir. Softw. Eng. 21 (2), 303–336. http://dx.doi.org/10.1007/s10664-015-9376-6.

Jiang, J., Lo, D., He, J., Xia, X., Kochhar, P.S., Zhang, L., 2017. Why and how developers fork what from whom in GitHub. Empir. Softw. Eng. 22 (1), 547–578. http://dx.doi.org/10.1007/s10664-016-9436-6.

Kirinuki, H., Higo, Y., Hotta, K., Kusumoto, S., 2014. Hey! Are you committing tangled changes? In: International Conference on Program Comprehension . ICPC, ACM, pp. 262–265. http://dx.doi.org/10.1145/2597008.2597798.

Kröher, C., Gerling, L., Schmid, K., 2018. Identifying the intensity of variability changes in software product line evolution. In: International Systems and Software Product Line Conference . SPLC, ACM, pp. 54–64. http://dx.doi.org/10.1145/3233027.3233032.

Krüger, J., Berger, T., Leich, T., 2019a. Features and how to find them - a survey of manual feature location. In: Software Engineering for Variability Intensive Systems. CRC Press, pp. 153–172. http://dx.doi.org/10.1201/9780429022067-9.

Krüger, J., Gu, W., Shen, H., Mukelabai, M., Hebig, R., Berger, T., 2018. Towards a better understanding of software features and their characteristics. In: International Workshop on Variability Modelling of Software-Intensive Systems. VaMoS, ACM, pp. 105–112. http://dx.doi.org/10.1145/3168365.3168371.

Krüger, J., Hebig, R., 2024. To memorize or to document: A survey of developers' views on knowledge availability. In: International Conference on Product Focused Software Process Improvement. PROFES, Springer, pp. 39–56. http://dx.doi.org/10.1007/978-3-031-49266-2_3.

Krüger, J., Li, Y., Lossev, K., Zhu, C., Chechik, M., Berger, T., Rubin, J., 2024. A meta-study of software-change intentions. ACM Comput. Surv. 56 (12), 300:1–41. http://dx.doi.org/10.1145/3661484.

Krüger, J., Li, Y., Zhu, C., Chechik, M., Berger, T., Rubin, J., 2023. A vision on intentions in software engineering. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE, ACM, pp. 2117–2121. http://dx.doi.org/10.1145/3611643.3613087.

Krüger, J., Mahmood, W., Berger, T., 2020a. Promote-pl: A round-trip engineering process model for adopting and evolving product lines. In: International Systems and Software Product Line Conference. SPLC, ACM, pp. 2:1–12. http://dx.doi.org/10.1145/3382025.3414970.

Krüger, J., Mukelabai, M., Gu, W., Shen, H., Hebig, R., Berger, T., 2019b. Where is my feature and what is it about? A case study on recovering feature facets. J. Syst. Softw. 152, 239–253. http://dx.doi.org/10.1016/j.jss.2019.01.057.

Krüger, J., Nielebock, S., Heumüller, R., 2020b. How can I contribute? A qualitative analysis of community websites of 25 Unix-like distributions. In: International Conference on Evaluation and Assessment in Software Engineering. EASE, ACM, pp. 324–329. http://dx.doi.org/10.1145/3383219.3383256.

Levin, S., Yehudai, A., 2017. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In: International Conference on Predictive Models and Data Analytics in Software Engineering. PROMISE, ACM, pp. 97–106. http://dx.doi.org/10.1145/3127005.3127016.

Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M., 2010. An analysis of the variability in forty preprocessor-based software product lines. In: International Conference on Software Engineering. ICSE, ACM, pp. 105–114. http://dx.doi.org/10.1145/1806799.1806819.

Ludwig, K., Krüger, J., Leich, T., 2019. Covert and phantom features in annotations: Do they impact variability analysis? In: International Systems and Software Product Line Conference . SPLC, ACM, pp. 218–230. http://dx.doi.org/10.1145/3336294.3336296.

Mauczka, A., Huber, M., Schanes, C., Schramm, W., Bernhart, M., Grechenig, T., 2012. Tracing your maintenance work – A cross-project validation of an automated classification dictionary for commit messages. In: International Conference on Fundamental Approaches To Software Engineering. FASE, Springer, pp. 301–315. http://dx.doi.org/10.1007/978-3-642-28872-2_21.

Mortara, J., Collet, P., 2021. Capturing the diversity of analyses on the Linux kernel variability. In: International Systems and Software Product Line Conference. SPLC, ACM, pp. 160–171. http://dx.doi.org/10.1145/3461001.3471151.

Nešić, D., Krüger, J., Stănciulescu, Ş., Berger, T., 2019. Principles of feature modeling. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering . ESEC/FSE, ACM, pp. 62–73. http://dx.doi.org/10.1145/3338906.3338974.

Palmer, S.R., Felsing, M., 2001. A Practical Guide to Feature-Driven Development. Pearson.

Passos, L., Teixeira, L., Dintzner, N., Apel, S., Wąsowski, A., Czarnecki, K., Borba, P., Guo, J., 2016. Coevolution of variability models and related software artifacts: A fresh look at evolution patterns in the Linux Kernel. Empir. Softw. Eng. 21 (4), 1744–1793. http://dx.doi.org/10.1007/s10664-015-9364-x.

Pohl, K., Böckle, G., van der Linden, F.J., 2005. Software Product Line Engineering. Springer, http://dx.doi.org/10.1007/3-540-28901-1.

Queiroz, R., Passos, L., Valente, M.T., Hunsen, C., Apel, S., Czarnecki, K., 2017. The shape of feature code: An analysis of twenty C-preprocessor-based systems. Int. J. Softw. Syst. Model. 16 (1), 77–96. http://dx.doi.org/10.1007/s10270-015-0483-z.

Ralph, P., bin Ali, N., Baltes, S., Bianculli, D., Diaz, J., Dittrich, Y., Ernst, N., Felderer, M., Feldt, R., Filieri, A., de França, B.B.N., Furia, C.A., Gay, G., Gold, N., Graziotin, D., He, P., Hoda, R., Juristo, N., Kitchenham, B., Lenarduzzi, V., Martínez, J., Melegati, J., Mendez, D., Menzies, T., Molleri, J., Pfahl, D., Robbes, R., Russo, D., Saarimäki, N., Sarro, F., Taibi, D., Siegmund, J., Spinellis, D., Staron, M., Stol, K., Storey, M.-A., Taibi, D., Tamburri, D., Torchiano, M., Treude, C., Turhan, B., Wang, X., Vegas, S., 2021. Empirical standards for software engineering research. arXiv:2010.03525. URL: https://arxiv.org/abs/2010.03525.

Razzaq, A., Wasala, A., Exton, C., Buckley, J., 2018. The state of empirical evaluation in static feature location. ACM Trans. Softw. Eng. Methodol. 28 (1), 2:1–58. http://dx.doi.org/10.1145/3280988.

Riege, A., 2005. Three-Dozen knowledge-sharing barriers managers must consider. J. Knowl. Manag. 9 (3), 18–35. http://dx.doi.org/10.1108/13673270510602746.

Rubin, J., Chechik, M., 2013. A survey of feature location techniques. In: Domain Engineering. Springer, pp. 29–58. http://dx.doi.org/10.1007/978-3-642-36654-3_2.

Schulze, S., Engelke, P., Krüger, J., 2023. Evolutionary feature dependencies: Analyzing feature Co-changes in C systems. In: International Working Conference on Source Code Analysis and Manipulation . SCAM, IEEE, pp. 84–95. http://dx.doi.org/10.1109/SCAM59687.2023.00019.

Sothornprapakorn, S., Hayashi, S., Saeki, M., 2018. Visualizing a tangled change for supporting its decomposition and commit construction. In: Annual Computer Software and Applications Conference. COMPSAC, IEEE, pp. 74–79. http://dx.doi.org/10.1109/compsac.2018.00018.

Steinmacher, I.F., Graciotto Silva, M.A., Gerosa, M.A., Redmiles, D.F., 2015. A systematic literature review on the barriers faced by newcomers to open source software projects. Inf. Softw. Technol. 59, 67–85. http://dx.doi.org/10.1016/j.infsof.2014.11.001.

Stănciulescu, Ş., Schulze, S., Wąsowski, A., 2015. Forked and integrated variants in an open-source firmware project. In: International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 151–160. http://dx.doi.org/10.1109/icsm.2015.7332461.

Swanson, E.B., 1976. The dimensions of maintenance. In: International Conference on Software Engineering. ICSE, IEEE, pp. 492–497.

van der Aalst, W.M.P., 2019. A practitioner's guide to process mining: Limitations of the directly-follows graph. Procedia Comput. Sci. 164, 321–328. http://dx.doi.org/10.1016/j.procs.2019.12.189.

Viegener, S., 2021. Empirical Evaluation of Feature Trace Recording on the Edit History of Marlin (Bachelor's thesis). University Ulm.

Wang, J., Peng, X., Xing, Z., Zhao, W., 2011. An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. In: International Conference on Software Maintenance . ICSM, IEEE, pp. 213–222. http://dx.doi.org/10.1109/icsm.2011.6080788.

Wilde, N., Buckellew, M., Page, H., Rajlich, V., Pounds, L., 2003. A comparison of methods for locating features in legacy software. J. Syst. Softw. 65 (2), 105–114. http://dx.doi.org/10.1016/s0164-1212(02)00052-3.

Yin, R.K., 2018. Case Study Research and Applications: Design and Methods. Sage.
Zhou, S., Stănciulescu, Ş., Leßenich, O., Xiong, Y., Wąsowski, A., Kästner, C., 2018. Identifying features in forks. In: International Conference on Software Engineering. ICSE, ACM, pp. 106–116. http://dx.doi.org/10.1145/3180155.3180205.
Zhou, S., Vasilescu, B., Kästner, C., 2020. How has forking changed in the last 20 years? A study of hard forks on GitHub. In: International Conference on Software Engineering . ICSE, ACM, pp. 445–456. http://dx.doi.org/10.1145/3377811.3380412.