When Do You Repeat Yourself? Voices from the Trenches of Linux Kernel Maintainers on Code Duplication

Luan Arcanjo
David Tadokoro
luanicaro@usp.br
davidbtadokoro@ime.usp.br
University of São Paulo
São Paulo, Brazil

Marcelo Spessoto
Rafael Passos
marcelomspessoto@usp.br
rcpassos@ime.usp.br
University of São Paulo
São Paulo, Brazil

Paulo Meirelles
paulormm@ime.usp.br
University of São Paulo
São Paulo, Brazil

Abstract

The Don't Repeat Yourself (DRY) principle is central to software maintainability, but empirical studies challenge its rigid use, describing beneficial cases of duplication. However, these rely on retrospective analyses, leaving a gap in understanding real-time decision-making and socio-technical dynamics. This paper presents an ethnographic study on how the Linux kernel community manages duplication debt via deduplication contributions. Using a clone detection tool called ArKanjo, we conducted a multimethod ethnographic study: first as a complete participant submitting patches to AMDGPU, then as a participant-as-observer mentoring 23 newcomers contributing to AMDGPU and IIO. Analysis of patch reviews suggests maintainers can tolerate duplication to accommodate driver-forking (T1), prioritize readability (T2), reduce integration overhead (T3), and preserve performance (T4). Our findings demonstrate that managing duplication in Linux is a nuanced process in which trade-offs among maintainability, clarity, and practicality outweigh dogmatic adherence to the DRY principle.

CCS Concepts

• **Software and its engineering → Open source model**; • **Human-centered computing → Open source software**.

Keywords

Linux kernel, Code Quality, Deduplication, Maintainers

ACM Reference Format:
Luan Arcanjo, David Tadokoro, Marcelo Spessoto, Rafael Passos, and Paulo Meirelles. 2026. When Do You Repeat Yourself? Voices from the Trenches of Linux Kernel Maintainers on Code Duplication. In *International Conference on Technical Debt (TechDebt '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3794915.3795783

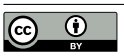
1 Introduction

Having tens or hundreds of developers working together on a software product is a complex and demanding task. Similar or even exact copies are a frequent and often unavoidable outcome. Regarded as a bad smell [5] and a major source of technical debt, *code duplication* (or *code cloning*) can negatively impact a project's maintainability, as any change to a repeated code segment (to fix a bug, for instance) requires replication across all other copies. Developers must therefore remain constantly vigilant to synchronize different parts of the codebase, a seemingly simple yet highly error-prone responsibility [9]. Removing duplicated code (*deduplication*) demands great care to preserve the original behavior, with minor mistakes possibly leading to serious problems in the future.

A widespread dogma among developers is the *Don't Repeat Yourself* (DRY) principle [8], which asserts that to develop reliable and maintainable software, "*every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*" Through static program analysis and mining software repository techniques, some empirical studies conducted on living real-world projects have challenged the DRY principle, showing that duplication can be advantageous depending on the context and evolution of the clones [7, 11, 15, 19]. Nevertheless, these types of analysis primarily operate in hindsight; identifying, cataloguing, and drawing conclusions from the already made and final decisions regarding technical debt management, which, in turn, can obscure **the reasons and the whole story behind accepting certain clone debts or actively blocking repayments in complex projects**. This work aims to capture that process as it unfolds.

To investigate these decision-making processes in a realistic setting, we turn to one of the largest and most influential software systems ever developed: the Linux kernel, a foundational *Free/Libre and Open Source Software* (FLOSS) project with more than 27 million lines of code, excluding non-programming code (verifiable through our replication package, see Section 7), and over 2,000 developers involved in version 6.17 [3]. Although it is known that some Linux device drivers deliberately duplicate code [11], **it can be hard to determine why a specific clone exists**, even when analyzing upstream commits in the mainline and subsystems. One could argue that the debt interest is too low and the debt principal is too high, or that there is a motive for accepting the clone unrelated to technical debt, or that simply no reasonable deduplication proposal has ever been submitted. We are left to speculate (in a well-informed manner) about developers' debt management practices and the reasoning behind each commit we analyze.

Leveraging a function-level clone detection tool we developed specifically for the context of Linux, called *ArKanjo* [1], we conducted a multimethod ethnographic study regarding code duplication. Ethnography is a well-established qualitative method for understanding people, their cultures, and work practices [4]. It provides insights into community members' values, beliefs, and



TechDebt '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2485-5/2026/04
https://doi.org/10.1145/3794915.3795783

practices [16]. In this study, we detected duplications and submitted deduplications to two Linux subsystems.

Through interactions with maintainers, we gained insights into how the Linux community perceives code quality related to code clones in the project's "development trenches". Contrary to the DRY principle, our findings confirm that, in some contexts, maintainers can tolerate duplication to improve readability. We also identified reasons coupled to **enhancing performance and avoiding integration overhead**. This empirical work also offers a distinct approach to researching technical debt management practices related to code duplication in FLOSS projects, beyond indicating that deduplication contributions are a viable entryway for newcomers in the Linux kernel.

2 Related Work

The works by Kamiya et al. [10] and Li et al. [13] proposed techniques to identify duplications, using the Linux codebase as a case study. Kapser and Godfrey [11] discussed patterns of cloning used in software, including situations where this could be positive, and Rahman et al. [15] contested the notion of clones as a bad smell. Tornhill [19] claims that clones evolving independently might not be problematic, whilst Juergens et al. [9] argue that inconsistent changes to clones that should evolve together are what primarily lead to faults. Wang and Godfrey [20] explores the use of cloning as a development technique for the SCSI drivers in the Linux Kernel, where similar devices had drivers born from code cloning. Harder [7] acknowledges the harmful case of cloned bugs but provides empirical evidence contradicting popular beliefs regarding the adverse effects of clones. The work concludes without providing clear guidance for practitioners and leaving an open path for future research, such as this one.

Our study provides additional empirical evidence and a distinct methodological perspective for researchers and practitioners interested in technical debt, code quality, and socio-technical practices within the Linux kernel project. **We obtained key insights by analyzing maintainers' reasoning when they accepted or rejected submitted changes (*patches*)**. In this respect, we make a new contribution to a long-standing discussion among software engineering researchers about when and how duplicate code matters.

3 Research Strategy

First, we needed a tool for detecting code duplication suitable for the Linux context. After reviewing existing static analysis tools such as *PMD*, *Duplo*, *Simian*, and *CodeClimate* (the analysis of the candidate tools is available in our replication package), we identified practical limitations when applying them to specific Linux kernel subsystems. Many were commercially licensed, used only token-based matching, or, like some FLOSS alternatives, were limited to file-scope comparisons, missing intra-file duplications. To address these constraints, we developed ArKanjo, a custom command-line tool for detecting duplicated functions in large C codebases. Released under LGPLv3, it uses Term Frequency-Inverse Document Frequency (TF-IDF) vector embeddings and cosine similarity that identify the four clone types described in the literature (Type-1 to Type-4) [2], including within the same file. This lightweight Command Line Interface (CLI) prioritizes maintainability, keeping our focus on code analysis and community interactions rather than tool development.

The Linux kernel is organized into subsystems like the process scheduler, memory management, and device drivers, each typically overseen by a dedicated maintainer or team. Our study focuses on two subsystems representing different contribution contexts. The first is the *AMDGPU DRM drivers subsystem*[1], which natively enables AMD GPU functionality in Linux. This substantial subsystem contains over 391,000 lines of code across more than 1,089 files. Similar to the SCSI subsystem studied by Wang and Godfrey [20], AMDGPU has a multi-layered architecture with generic higher layers and lower layers comprising hardware-specific drivers implementing the same basic functionality. The second subsystem is the *Industrial Input/Output* (IIO) subsystem, chosen because it is considered an accessible entry point for new contributors. IIO provides support for devices performing analog-to-digital or digital-to-analog conversions and contains over 281,772 lines of code across more than 755 files.

Using ArKanjo, we assessed the viability of reducing the identified duplications through a two-phase ethnographic study employing participant observation methodology [6]. In the first phase, adopting the role of *complete participant* [6], we engaged directly with the Linux kernel community by submitting deduplication patches to the AMDGPU drivers. In the second phase, shifting to the role of *participant-as-observer* [6], we introduced a similar activity in a university course, in which students were tasked with proposing patches for the AMDGPU or IIO subsystems. Throughout this process, we provided guidance and mentorship to the students as they interacted with the community.

3.1 Complete Participant Observation Study

We first conducted a complete participant observation experiment, in which the main author (a graduate student) served as a first-time contributor to the AMDGPU drivers, collecting artifacts of his experience along the way. We ran the ArKanjo tool on the AMDGPU codebase and manually analyzed the largest duplications by line count, finding one pair we judged promising to try to deduplicate. Given the duplicated function pair, we observed that the functions' files contained multiple other clones. Thus, we proposed a simple systematic approach to mitigate all the duplicate functions in the context, not just the initial function pairs. After approaching this refactoring with a systematic strategy, we sent a patch to the AMDGPU drivers mailing list for the feedback of maintainers, while documenting the process, interactions, and impressions (also available in our replication package, see Section 7).

In C programming, communication between source files is achieved by creating header files that specify libraries [12]. Since the Linux kernel is primarily written in C, our systematic strategy consisted of eliminating code clones by consolidating them into a single library, thereby replacing duplicate instances across the codebase. For each flagged function, we used the tool to locate all duplications of that function in the AMDGPU codebase, since it may have been replicated beyond the two occurrences initially

[1]Specifically, the *Display Core* component, the *hardware-abstraction layer* (HAL) for display functionality.

detected. This strategy resulted in a collection of functions and corresponding duplication occurrences in code files.

To identify shared code more effectively, we extended this approach to search for other common functions across all collected files. We then applied specific refactoring methods to each shared function. If the functions were identical across files, we removed the duplicates and created a single function in the library. If modifications existed, we applied case-specific refactoring.

3.2 Participant-as-Observer Observation Study

To broaden our study and observations, we adopted a participant-as-observer approach, involving students as FLOSS project newcomers to make practical contributions to the Linux kernel. This approach formed the core activity of a university course [18], in which we guided students to remove code duplications in the Linux kernel as one possible pathway for contributing to the project in the first phase of the course.

The 2025 course offering had 37 students (25 undergraduate and 12 graduate), who were asked to form groups of two or three members, but graduate students could work individually [18]. To assist students in this task, we (a professor and three more experienced graduate students) prepared several alternatives for contributing to the Linux kernel, including simpler contributions such as cleaning coding style issues. In this context, we specified two options for removing duplications in the Linux kernel.

Deduplication Option 1. We ran ArKanjo on the IIO subsystem to generate a list of duplicated function pairs, then curated it to highlight actionable cases for students. Specifically, we (i) kept only pairs within the same file, since cross-driver duplications could involve multiple maintainers or distinct interfaces; (ii) removed very short duplications; (iii) removed semantical Type-3 and Type-4 clones, for the higher complexity in refactoring them; and (iv) manually ranked the remaining pairs, adding annotations to guide students toward entries likely to be accepted. In this option, students were tasked with selecting a recommended entry, devising a way to remove the duplication, and submitting their patches to the IIO mailing list for review.

Deduplication Option 2. We offered an experience similar to our initial complete participant observation, where students managed the entire workflow: running the tool, analyzing results to identify potential deduplications, creating a patch, and submitting it to the driver's maintainers. This option provided more freedom of choice but made the students' task more complex.

Of the students who chose to work on the duplication issues, 23 (16 undergraduate and 7 graduate) formed 11 groups to pursue the first alternative. One graduate student opted for the second alternative. None of these students had prior experience contributing to the Linux kernel, making this their first attempt to submit a patch as newcomers to the project.

It is important to stress that, **no matter the chosen approach, we supported all groups by providing review cycles on their contributions before submission**, to prepare students for the interactions with the maintainers and alleviate rough, but natural, newcomers' mistakes. After the submission, we helped students understand the feedback from maintainers and gave directions on how to move forward. This close, yet indirect, participation through intimate mentoring characterizes the second phase as our participant-as-observer observation.

For this second phase, our primary data source was the mailing list threads, where all development occurred (also available in our replication package, Section 7). Using blog posts, the groups documented their experiences submitting a patch to the Linux kernel, and we also leveraged those to complement our observations.

4 Results and Discussion

In the **complete participant observation** study (first phase), we sent the first revision of the patch on August 9, 2024, and a resend followed on October 9, 2024, due to a lack of initial feedback. Nevertheless, for the first two versions, the maintainers' response requested minor changes for coding style, license use, and alignment with existing conventions. In the third version, the feedback was to move the new generic library functions into an existing file rather than creating a new one. Then, the fourth version was accepted and integrated into the kernel on February 25, 2025, removing 406 lines in total and impacting three generations of drivers.

Since we experienced a considerable delay in the review process, we directly contacted a maintainer to understand the cause. From this conversation, we understood that it is commonplace for drivers to be cloned (sometimes entirely) to provide a solid base and enhance independence between them, allowing developers to make changes to a specific one without needing to check for regressions on others. This practice is very similar to that detected in the SCSI subsystem by Wang and Godfrey [20], agrees with Tornhill [19] regarding clones evolving independently, and is also the hardware variation of the forking pattern defined by Kapser and Godfrey [11]. Even so, this driver duplication is a *Self-Admitted Technical Debt* (SATD) [14] listed as a welcomed contribution in the official documentation of the Display Core component[2]. In this framing, we can attribute the delay to the change impacting multiple drivers, which resulted in greater principal on the maintainer's side to ensure the repayment was stable. Prioritization of other repayments may also have contributed to the delay. With that in mind, this complete participant observation produced our initial takeaway:

> **Takeaway T1 (Driver Forking):** *Driver developers do not necessarily view duplicated code negatively, often considering the interest low and the principal high; however, even when such duplication is explicitly acknowledged as technical debt, its removal is accepted when repayments are perceived as safe and stable.*

Our initial observations indicate that, although driver forking has been previously reported in large-scale systems, our experience in the Linux kernel shows that removing such duplication can be accepted in practice, pointing to opportunities for further collaboration and empirical studies on technical debt management in this context.

To further expand T1 and possibly derive other takeaways, we examined the contributions made through our **participant-as-observer observation**, substantiating them with quotes from the

[2]https://www.kernel.org/doc/html/v6.17/gpu/amdgpu/display/display-contributing.html#reduce-code-duplication.

maintainers that emerged during the review process[3]. This broader analysis aimed to capture diverse maintainer perspectives and contextual factors influencing their decisions.

Table 1: Summary of newcomer deduplication contributions.

GID	SS	SML	Patch Status	Takeaway	Diff
0	AMD	100%	Accepted (v4)	T1	+114/-520
1	AMD	90%	Accepted (v2)	T1	+90/-489
	IIO	100%	Dropped (v1)	T1	+2/-12
2	IIO	100%	Dropped (v2)	T2, T3	+7/-14
3	IIO	100%	Dropped (v1)	T2	+23/-27
4	IIO	100%	Accepted (v1)	T2	+12/-30
5	IIO	100%	Accepted (v3)	T2	+2/-7
6	IIO	90%	Accepted (v1)	T2	+17/-70
7	IIO	90%	Dropped (v1)	T2	+95/-92
8	IIO	90%	Accepted (v3)	-	+20/-36
9	IIO	90%	Accepted (v4)	T4	+20/-16
10	IIO	90%	Accepted (v6)	T4	+149/-243
11	IIO	90%	Dropped (v1)	T2, T4	+32/-58

Table 1 summarizes all the contributions sent by the main author and student groups (GID)[4]. It contains the information about the chosen subsystem (SS) and the similarity threshold (SML). Each entry also has the patch status, takeaways that may have surfaced from the interaction, and a code differential (Diff) corresponding to the final patch versions before it was dropped or accepted into the code base. The code differential for a set of patches corresponds to the sum of added and removed lines of each patch.

Groups 1 to 11 all sent one patch to IIO from the deduplication option 1 described in Section 3, yet Group 1 also submitted one to AMDGPU using the deduplication option 2.

The AMDGPU patch of Group 1 resembled the patch from the complete participant phase, but it received feedback and was merged much more quickly (within 10 days). This experience corroborates the nuanced perception described in T1, further opening the discussion on how developers, on the one hand, consciously and consistently fork drivers as a well-documented practice, and, on the other hand, acknowledge duplications from this practice as debt and welcome repayments. The IIO patch of Group 1 targeted a driver using the *Register Map* API, which requires defining the `regmap_config` struct. It inlined two functions returning false used as callbacks in `regmap_config`, so this deduplication was inherently incorrect (the fields expect function pointers), as the maintainer ambiguously communicated: *"Take another look at what you are doing here."* This exemplifies the API/library protocol templating pattern noted by Kapser and Godfrey [11], which, from our analysis of the case, is also rooted in the practice of driver forking, reaffirming T1, once again.

Groups 2 and 5 both refactored similar predicate functions that were logical negations of each other. Group 5 first used macros, but after maintainer feedback, both groups converged on keeping one function and having the other return its negation. Although initially correct, both solutions underwent significant changes driven by maintainer concerns about readability. For Group 2, a maintainer said that *"[...] the naming as* `_reg_check()` *is not helpful as it doesn't indicate anything specific is being checked."*, while for Group 5, the feedback explicitly highlighted the precedence of readability over deduplications: *"I think the old code is more readable than hiding the values in a macro even if it is duplicating a few lines of code."* Even though the deduplication of Group 3 was not similar to that of Group 2 and Group 5 in terms of implementation (it used the parametized method), it also brought forth the readability factor: *"In my view this isn't a significant enough reduction to justify the more complex code."*

Groups 4, 6, 7, and 11 used a parameterized approach with a new generic function parameter. The same maintainer reviewed all three, but only Groups 4 and 6 had their patches accepted. For Group 4, we think this was because it applied to a straightforward iterative algorithm and also fixed a bug, helping its approval, whereas we were not able to identify a solid, explicit reason for accepting the patch for Group 6. Although deduplication does not always reduce lines, the patch from Group 7 added lines, which the maintainer cited as a drawback. Still, the real reason for the rejection was that the deduplication scattered parts of a complex code that resided close together before: *"Wrapping this up doesn't provide any real advantage, requiring as it does the reviewer to look at this function AND where the value is set rather than seeing them in one place."* Needing to jump to a function definition (to see its behavior) or a function call (to see its concrete parameters) to understand code is a cognitively demanding task for developers [17]. The reason for rejecting the patch from Group 11 was that the deduplication lowered readability, as the maintainer was *"[...] not sure the code reduction is sufficiently to cover the resulting loss of readability."*.

It is not a novelty that understandability can motivate cloning, so much so that refactorings can even break the conceptual cohesiveness of the code [11]. The empirical observations described corroborate this perception, as this was the motivation to drop patches from Groups 2, 3, 7, and 11. At the same time, the accepted patches from Groups 4, 5, and 6 reduce readability, if we consider the same arguments posed by maintainers in the dropped patches (which were close to these accepted ones). Analogous to T1, these divergent insights on duplication and readability across multiple reviews show nuance on this topic and leave space for future research toward clarification. In this sense, we formulated our second takeaway:

> **Takeaway T2 (Readability):** *Maintainers prioritized readability over removing duplications in many different contexts. Accepting debt from duplication may be worth it if it results in easier-to-read, more understandable code. Nevertheless, the criteria used to accept or drop similar repayments that impact readability requires further research.*

Going back to Group 2, maintainers questioned the merit of the proposed patch. While the previous interactions highlighted how maintainers often favor readability over strict code deduplication, this case added a new dimension to our understanding: the *cost-benefit* evaluation of proposed changes. A maintainer pointed

[3]We **do not claim** that these quotes fully represent the maintainers' stance on code duplication or any technical debt management matter.

[4]Since the first author was also a newcomer, the first phase experience was included as Group 0.

out that the effort required to propagate such modifications after acceptance (i.e., during the upstreaming process) could outweigh the perceived benefits: "*[...] such patches might not worth it since the proposed improvement is very small (and questionable) while the upstreaming process still requires some effort.*" From this feedback, we crafted our third takeaway:

Takeaway T3 (Integration Overhead): *Maintainers may drop a duplication repayment if it is not considered impactful enough, because just integrating it into the Linux upstream incurs considerable principal.*

In the interaction of Group 11, there was also a hint that performance could be considered when evaluating the worth of a deduplication, whilst reaffirming that readability was a priority in that context: *"I'd be slightly interested to see the optimized output of the two approaches, but this is far from a high performance path so we care a lot more about readability here."* Nonetheless, this notion surfaced aggressively with Group 10, where the contribution could completely remove both duplicated functions using the inline method by leveraging helpers; however, one of the functions had a call that, if eliminated, would result in a significant performance impact: *"Note that is not an appropriate change for the large reads though [...] This is the one case were* `spi_write_then_read()` *is probably not appropriate due to the large buffers that are potentially involved."*

Groups 8 and 9 applied the extract method with helpers to eliminate duplicate code in the function body. Group 8 refined its patch based on trivial maintainer feedback, leading to smooth acceptance without yielding insights. Group 9 followed a similar path but produced an interaction that heightened performance concerns, where a maintainer claimed that *"Even though there is less code repetition, we now have an extra comparison [...]"*, with another maintainer endorsing this in version three by saying: *"There is always a balance/trade-off between modularity and execution speed. I agree with* `anonymous-maintainer`*'s reply in the first patch [...]"*, even though the patch was accepted in version four.

These interactions, coupled with the low-level development context of Linux and driver development, which is can be related to devices with limited resources, produced our fourth and final takeaway:

Takeaway T4 (Performance): *Interactions with maintainers hinted that duplications can be forgiven if they improve performance in contexts where resources are limited. In these scenarios, avoiding performance debt is prioritized.*

As mentioned previously, T1 substantiates with close ethnographic observation and adds context to what the literature presents. Analogously, T2 also deliberates on what is already described in the field's studies, but in a direction that challenges them. On the other hand, takeaways T3 and T4 go beyond and offer nuanced perspectives on the debt management decisions Linux maintainers practice, pointing to a distinct source of principal stemming from the project's upstreaming workflow and the low-level context of Linux development, where having duplication can be favored over performance debt.

In this sense, we briefly examine another aspect of importance stemming from this work: using ArKanjo to create deduplication opportunities in the Linux kernel can be an excellent introduction for newcomers.

To illustrate this point, we display some statistics concerning the 13 contributions contemplated in this paper: (i) Collectively, the patches proposed the insertion of 585 lines and the removal of 1626 lines; (ii) 8 out of 13 contributions were accepted, resulting in a success rate of 62%; (iii) for accepted contributions, the code diff value was +255/-1152, averaging +31.9/-144.0 additions/removals; (iv) For rejected contributions, the code diff value was +181/-231, averaging +36.2/-46.2 additions/removals.

5 Threats to Validity

We outline limitations using standard validity categories.

Internal Validity. Selection bias may affect findings. Targeting large duplications in AMDGPU likely captured complex, conservative cases, while curating simpler duplications for participants may have oversimplified challenges and favored easier patches. Results may not generalize to medium-complexity or inter-file duplications.

External Validity. Findings are based on two Linux kernel subsystems (AMDGPU and IIO) and 23 student contributors. The kernel is a unique FLOSS project with specific norms, so maintainer perspectives may vary. The lack of previous kernel experience from participants also limits generalization to experienced contributors.

Construct Validity. Only Type-1 and Type-2 duplications were studied, excluding more complex clones, although ArKanjo also detects Type-3 and Type-4 clones. This restricts insights into maintainers' attitudes toward semantically similar duplications.

Conclusion Validity. The ethnographic approach provides qualitative insight but not statistical generalization. Conclusions are drawn from 13 patch interactions. In this sense, a larger sample would strengthen robustness.

6 Concluding Remarks

This ethnographic study, combining complete participant and participant-as-observer observations, provided a realistic view of the opportunities and challenges in repaying duplication debts in the Linux kernel. It shows that first-time Linux contributors could effectively use the ArKanjo tool to identify duplicated functions and submit meaningful patches.

More interestingly, the study employs a novel ethnographic approach to engage with technical debt literature that challenges the DRY principle, while refining the discussion within the Linux project. The first two findings empirically corroborate the literature on the benefits of duplication by detecting the driver-forking pattern (T1) and duplication motivated by readability (T2); however, contrasting observations across both contexts highlight significant nuances that call for future research. The latter two findings highlight project-specific debt management in Linux: change propagation considerably increases principal (T3), and performance preservation in resource-constrained settings (T4) further justifies tolerating clones.

Without disregarding its limitations, this work shows that the Linux community has particular technical debt management practices, carefully weighing the pros and cons of accepting versus

repaying duplication debts. Maintainability, clarity, and practical trade-offs shape maintainers' judgments, suggesting that consciously asking *"**When** do you repeat yourself?"* can be better than mindlessly following the *"Don't Repeat Yourself"* principle.

7 Replication Package

All anonymized review threads are available in this Zenodo repository: doi.org/10.5281/zenodo.17503684. Replication resources include the ArKanjo repository (github.com/arkanjo-tool/arkanjo), its documentation (arkanjo-tool.github.io/), and an analysis of candidate tools (github.com/arkanjo-tool/arkanjo/wiki/Evaluation-of-other-duplication-tools). The container for the Linux v6.17 LoC statistic from Section 1 is also provided (github.com/linux-duks/linux-programming-loc).

Acknowledgments

This study is funded by the São Paulo Research Foundation (FAPESP) and the São Paulo State Data Analysis System Foundation (SEADE), under grants 2023/18026-8 and 2025/05395-0.

References

[1] Luan Arcanjo, David Tadokoro, and Paulo Meirelles. 2025. ArKanjo: a tool for detecting function-level Code Duplication in the Linux Kernel. In *DebConf25*, Olivier Barais and Bastien Roucariès (Eds.). IRISA, Brest, France, 3. https://hal.science/hal-05335545

[2] Chang-Feng Chen, Azlan Zain, and Kai-Qing Zhou. 2022. Definition, approaches, and analysis of code duplication detection (2006–2020): a critical review. *Neural Computing and Applications* 34 (08 2022), 1–31. doi:10.1007/s00521-022-07707-2

[3] Jonathan Corbet. 2025. *Development statistics for 6.17*. Linux Weekly News. https://lwn.net/Articles/1038358/ [Online; Last accessed on Jan. 25th, 2026].

[4] W Alex Edmonds and Thomas D Kennedy. 2016. *An applied guide to research designs: Quantitative, qualitative, and mixed methods*. Sage Publications.

[5] Martin Fowler. 2018. *Refactoring: improving the design of existing code* (2 ed.). Addison-Wesley Professional.

[6] Raymond L. Gold. 1958. Roles in Sociological Field Observation. *Social Forces* 36 (1958), 217–223. doi:10.2307/2573808

[7] Jan Harder. 2017. *Software Clones-Guilty Until Proven Innocent?* Logos Verlag Berlin GmbH.

[8] Andrew Hunt and David Thomas. 2000. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., USA.

[9] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter?. In *2009 IEEE 31st International Conference on Software Engineering*. 485–495. doi:10.1109/ICSE.2009.5070547

[10] T. Kamiya, S. Kusumoto, and K. Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (July 2002), 654–670. doi:10.1109/TSE.2002.1019480

[11] Cory J. Kapser and Michael W. Godfrey. 2008. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering* 13, 6 (Dec. 2008), 645–692. doi:10.1007/s10664-008-9076-6

[12] Brian W Kernighan and Dennis M Ritchie. 1988. *The C programming language*. prentice-Hall.

[13] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. 2006. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering* 32, 3 (March 2006), 176–192. doi:10.1109/TSE.2006.28

[14] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 91–100. doi:10.1109/ICSME.2014.31

[15] Foyzur Rahman, Christian Bird, and Premkumar Devanbu. 2012. Clones: what is that smell? *Empirical Software Engineering* 17, 4 (01 Aug 2012), 503–530. doi:10.1007/s10664-011-9195-3

[16] Helen Sharp, Yvonne Dittrich, and Cleidson R. B. de Souza. 2016. The Role of Ethnographic Studies in Empirical Software Engineering. *IEEE Transactions on Software Engineering* 42, 8 (2016), 786–804. doi:10.1109/TSE.2016.2519887

[17] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Portland, Oregon, USA) *(SIGSOFT '06/FSE-14)*. Association for Computing Machinery, New York, NY, USA, 23–34. doi:10.1145/1181775.1181779

[18] David Tadokoro, Rafael Passos, and Paulo Meirelles. 2025. Guidelines for Boosting Long-Lasting FLOSS Contributors. In *DebConf25*, Olivier Barais and Bastien Roucariès (Eds.). IRISA, Brest, France, 6. https://hal.science/hal-05334509

[19] Adam Tornhill. 2018. *Software Design X-Rays: Fix Technical Debt with Behavioral Code Analysis* (1st ed ed.). The Pragmatic Programmers, LLC.

[20] Wei Wang and Michael W. Godfrey. 2011. A Study of Cloning in the Linux SCSI Drivers. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. 95–104. doi:10.1109/SCAM.2011.17