How does quality deviate in stable releases by backporting?

Jarin Tasnim
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
jarin.tasnim20@usask.ca

Debasish Chakroborti
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
debasish.chakroborti@usask.ca

Chanchal K. Roy
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
chanchal.roy@usask.ca

Kevin A. Schneider
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
kevin.schneider@usask.ca

***Abstract*—Software goes through continuous evolution in its life cycle to sustain bugs and adopt enhanced features. However, many industrial users show reluctance to upgrade to the latest version, considering the stability and intuitive solace of the release they are using. This boosts the need to derive change patches from state-of-the-art versions to older software versions. This phenomenon is frequently supported by 'Backporting' in the industrial setting as the intent for backward patch propagation stood principally to sustain older releases, and the contribution does not count up to the upstream repository. However, it is yet unknown whether backport can act as a credible threat for stable releases. In this study, we aim to empirically quest backports to reveal the evolution trend of code entities through maintenance and pinpoint how they pull stable releases into the weak spectrum. The breakdown shows code entities often encounter gradual transformation in size, complexity and coupling due to consecutive commits on them. However, the numerics of outlier quality degradations are not insignificant at all in this context which calls for further investigation into why and when they may occur. Moreover, we observed that vulnerable change transmission often materializes with quality degradation. Understanding these issues and consequences is crucial for effectively supporting the backporting process for stable release maintenance.**

***Index Terms*—Code Quality, Backporting , Software Engineering → Software Maintenance**

I. Introduction

With the expanding numeral of complex and large-scale contemporary software systems, assuring high quality and efficient tracking of software has now become one of the primary concerns both in industrial and academic settings [1]. Many robust and broad software companies maintain multiple releases of a single software for eclectic reasons. First, they support targeted users by leveraging user-specific features and requirements in distinct releases. Second, this scheme allows the administrator to ensure the security and stability of intended release by determining maintenance strategy. In addition, users often portray reluctant nature to update their software regularly owing to their intuitive doubts on compatibility issues, reliability and typical human-centric perspective [2], [3]. Thus, many release versions require maintenance support in their lifetime before finally being declared as 'Legacy software'. For instance, Python 2.0 came in 2000. Although the stakeholders stated they would discontinue support for Python 2 in 2008 after the release of significant feature enhancements in the Python 3 series, the authority had to ensure maintenance support for stable releases of the Python 2 version series up until 2020 because of the existing projects on it and user demands. Even after announcing Python 2 as a legacy artifact in 2020, many users have had several pleas to seek support in their system contextualized in the Python 2 release series [4], [5].

However, Python 2 is not a mere circumstance, but sustaining aid and maintenance in distinct stable releases is a common and well-established scheme in the current open source platform. Think about the Linux distribution system, which has more than 300 customized imprints that swing on the same underlying Linux kernel. Distinct distributions (i.e., Ubuntu, Fedora, Linux Mint and so on) uphold and maintain their own stable releases by fixing bugs and adding new features, as users do not always keep up to date with the latest versions. To aid systematic tracking and managing changes in multiple stable releases of a software system, 'Backporting' is familiarized in Version Control Systems (VCS) [6]–[9]. Backporting refers to accommodating software patches or updates from a recent version to an older version of the same software.

In the most widely utilized VCS, Git, we may backport code directly from one version to another. However, this may propagate unnecessary changes when comparing two branches to make a pull request. Thus, to sustain the identical issues in stable releases, reversed code dispatch is made possible by assembling a duplicate pull request which contains a particular code patch or its slightly adaptive patch to the preceding stable versions.

Simply put, for a change set fixing a maintenance task in the default repository, backport refers to the process of deriving

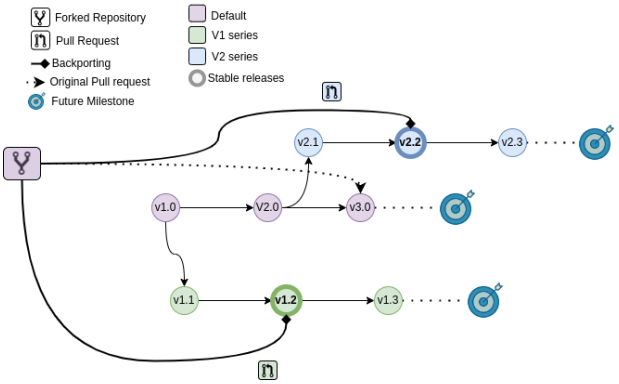


Figure 1: Backport code propagation among different versions

the same patch or its slightly adaptive patch to the preceding stable versions. For instance, a bug incurred in the default development scheme, which also exists in v_1 and v_2 series (in Fig 1). The fix for that specific bug has been incorporated (dotted line) in the default repository by forwarding code propagation from a forked repository. This porting/patch transmission is called 'Forwardporting' because the direction of the code flow is parallel to the upstream development. Thus, the contribution directly adds up to the milestone. However, to support the same bug in old stable releases in the v_1 and v_2 series, duplicate pull requests are propagated into the $v_{1.1}$ and $v_{2.2}$, which consist of the same or slightly modified changes of the original pull request to make the fix compatible with the targeted release. As the tendency of code flow in such a scenario is reversed to the upstream pipeline to sustain maintenance crises in stable releases, they are typically referred to as 'Backport' both in the corporate jargon [10]–[12] and on the academic ground [13]–[15].

It is evident that patch transmissions are often offered separate times and attention in their life cycle based on their eclectic purpose for transition and perspective in the industry setting [15], [16]. As the underlying motivation and the pragmatic procedures to institute backports are distinct from upstream development, they are not always inspected and reviewed in a consistent norm [17]. Thus, they can introduce vulnerable issues (like violating quality concerns [18]–[21], highly coupled codes [22], [23], lack of cohesion, code smells [24]) to the existing stable system. Specifically, in the backporting scenario, the code propagations are not consistently pre-determined and deliberate owing to the incremental release traits of legacy software [15], [25]–[28]. Often cases, backports are given less attention and effort [13], [27]. Furthermore, an empirical exploration disclosed that roughly 50% of backports ought to embrace more than 13% mutations on top of the original pull request to support the identical issue in a stable release paradigm [13]. It is yet unknown whether the mutation fixes in backports to make the change patches compatible with prior releases impose any threat or not. Besides, the strategies [13], [29] and security concerns [30]–[33] observed to accomplish backporting, and the challenges encountered by the practitioners are also significantly distinct from forward transmission scheme. There has been a large domain of research [34], [35] that try to investigate quality aspects in forward patch transmission, yet to our knowledge, no attempts has been made to identify and investigate code entity propensity and quality degradation in the backporting scenario. Thus, what changes are backported and how quality attributes are influenced in this scheme require further illustration. Understanding the density, type, and impact of vulnerable propagations in stable software systems can assist developers in identifying potential threats with each code propagation.

II. STUDY DESIGN

Pull requests are backported to support concerns in the existing stable branches, so intuitively we assume them as a component of stable release maintenance. The pursuit of the analysis is to scrutinize the change chronology of backports across distinct releases in software projects from 2 vital perspectives: (1) to investigate the evolving nature of stable releases in terms of size, complexity and coupling through backporting maintenance initiative; (2) to identify whether the adaptive changes incorporated in backport to fit in target release impose any threat to that release. More specifically, the study seeks to address the following two research questions:

1) **How do distinct dimensions of stable releases evolve in the backporting scenario?**
   Ironically, software aging has always been considered closely associated with software evolution [36]–[39]. Thus, how the dimension of stable releases regains maturity throughout the maintenance endeavour before finally being regarded as a 'stale release' is an open question. Setting this inquiry was driven by the intent to investigate to what extent the common wisdom suggests 'Code structure become more complex and coupled as a consequence of continuous maintenance and evolution activities' is authentic in the backporting scenario. To our knowledge, no studies have tried to investigate an in-depth characterization of different dimensions in which a software system expands in stable release backport maintenance. Specifically, we examine how distinct characteristics (size, complexity, coupling) in stable releases evolve to comprehend the 'Backporting' maintenance trend. For instance, we strive to understand whether backports can cause a code entity to become instantly complex or its complexity materializes 'gradually' throughout the evolving scheme of stable release maintenance activities. Apprehending the evaluation tendency of stable releases can assist in making better administrative maintenance decisions and formulating further strategies for Backporting process.
2) **Do the transformations included in backport to fit in the target release inflict any vulnerability/threat to that release?**
   The second research question seeks to empirically pinpoint whether the customized modifications introduced to the original pull request to make it compatible with

prior stable releases are susceptible to introducing any vulnerability or not. If the practitioner processes comparatively less knowledge regarding the composition, structure and exposure of the target release, associated changes in backports can inflict threats to the stability of that release. In addition, backport may take different merging times and effort as the discussion duration required for backports are more than typical pull requests [13]. Thus, identifying how safe they are to deal with is important.

III. Data Extraction and Analysis

A. Context Selection

We collected the 1000 top-ranked open-source projects from the Gitstar Ranking website [40]. To distinguish backports, we strive to identify how the pull requests are directed to stable version maintenance in pragmatic surroundings (such as Github). Generally, a set of keywords/labels are employed in practice to refer to backports. For instance, in the Ansible project, three tags (backport, backport-request, backport-verified) are used to refer backports. Tags are also employed to categorize the stable version subjected to accepting patch changes of the pull request. Most of the subject systems obey this convention of directing backports. Thus, in order to retrieve the backports, we matched the labels/ tags of corresponding pull requests. In addition, we utilized 2 inclusion criteria (must contain at least 400 pull requests and release note must reflect that the stakeholders maintain a set of stable releases) and 4 exclusion criteria (small-scale projects are excluded by identifying the total number of files ($N < 100$), and Line of code ($LOC < 2000$), and the total number of commits ($NC < 100$), and brief lifespan ($S < 10$ weeks)).

Incorporating the inclusion and exclusion criteria is paramount, as we desire to construct the dataset to reflect the real-life scenario of projects performing backporting for maintenance purposes. After filtering and satisfying the criteria, we selected the top 3 Python projects (Table I) with the two highest forward and backporting ratios and one lowest ratio (Salt) for the preliminary study. The stable releases of the chosen subject systems [41] are identified by the published contribution document and release notes.

We targeted three aspects of code evolution (size, complexity and coupling) and selected the mostly utilized metrics in existing literature [42] to exploit these traits of the codebase evolution. For instance, the choice of utilizing the McCabe's cyclomatic complexity [43] to exploit the complexity dimension is driven simply by the fact that it is the most widely recognized standard to confine the stability and tier of confidence in code entity. The Table II furnishes an overview of selected metrics and their description. Python Radon library [44] is used to compute the metrics value [45].

B. Code entity evolution exploration

Pull requests are composed of a series of commits and each commits encapsulate a number of file/patch changes. We took micro-level code entities (files) to determine how they evolve in distinct versions. To answer our research questions, we mined the change history of release-wise backports and had to develop a metric-based strategy to see how a single code entity evolves in separate releases through backporting. A single file can undergo distinguishable modifications during the maintenance life cycle in different release versions of the same subject system. For instance, considering a static time frame, the complexity of '__init__.py' file (Fig. 2) in the Ansible has distinguishable evaluations across 4 different releases as they meet distinct series of commits.

Table I: Subject system

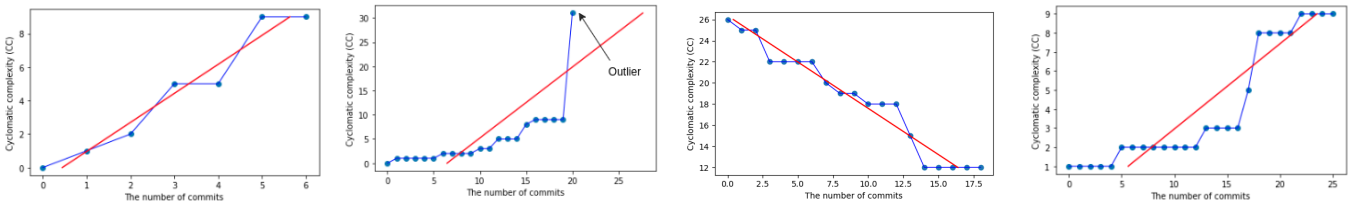
Subject System	# of Pull requests	# of Backportes
Ansible	47712	5985
Kibana	93439	26,888
Salt	37356	46

Table II: Quality metrics considered in the context of RQ1

Metrics	Description
Line of code (LOC) [46]	The number of line of code
Cyclomatic complexity (CC) [47]–[49]	the McCabe's Cyclomatic Complexity of code entity
Coupling Between objects (CBO) [48]	The number of classes that are coupled to a particular class

Cloning the stable releases $r \subset \{r_0, r_1, r_2 ... r_n\}$ of the selected subject system, we analyze each stable release of a project P_i using BackportHistoryMiner (i.e., our developed tool) to confine how the code entity proliferation across distinct versions are correlated. For each commit, changes backported $c \subset \{c_1, c_2, c_3 ... c_n\}$ on a specific file to the stable version r_i , BackportHistoryMiner track them in their chronological order. Mining the complete release-specific backport change history of a project, we aim to determine the consecutive set of commits prompted in a particular file that led to degrading its quality and investigate specific traits (size, complexity and coupling) transition. To account for how a characteristic (suppose complexity) of a file f_i evolves during the maintenance drudgery, in a software release r_i, we assemble the following strategy to label them:

1) if the regression line of file f_i gradually leans (in Fig. 2c) toward and cross the enhance/degrade threshold as a result of the sequential commit occurrence after the release of repository r_i, then f_i is tagged as 'Gradual-Complexity-Enhance (GCE) ' or 'Gradual-Complexity-Degrade (GCD)' respectively. In this context, the enhance/degrade threshold accounts for the acknowledged threshold of the chosen metrics. For instance, in the case of Cyclomatic Complexity (CC), as the traditional benchmarking urges [50], [51], it is considered a reasonable and minimal risk for a code entity if it's less than 10, medium if $10 < CC < 20$ and destructive if it goes beyond 20.
2) if the conjunction between the commit-regression line and threshold is not a result of a series of backporting commits, instead, a data point in the regression line act



(a) S-2.4 (gradual increase - NC) (b) S-2.5 (outlier degrade - OCD) (c) S-2.6 (gradual enhance - GCE) (d) S-2.7 (gradual increase - NC)

Figure 2: The cyclomatic complexity of '__init__.py' file evolve in the 4 stable releases of Ansible project
Blue: The original line connecting the metric value points **Red:** The best-fit regression line of metric value points

as an outlier as well as is responsible for crossing the enhance/degrade metric threshold, then we denote it as a ' Outlier-Complexity-Enhance (OCE)' (in Fig. 2b) or 'Outlier-Complexity-Degrade (OCD)'.

3) if the regression line may or may not leans toward the enhance/degrade metric threshold line but does not cross the line (in Fig. 2a and 2d), then we denote it as 'Not-Complex (NC)' entity.

One primary intent behind utilizing the regression scheme to visualize the evolution trend is that the standard deviation was minimal for release-wise sample data while considering a fixed time window. That implies that the metric data points were not too dispersed within the transition. Instead, most data points were cluttered near the best-fit line. However, we do acknowledge that the relationship between the metrics measures over the consecutive commit changes can be non-linear. Thus, we would like to work on identifying these distinctive patterns for future studies.

To downsize computational constraints, metrics computation is commenced from the final commit on each file f_i after release. In the successive commits, the metrics calculation has been accessed on added or modified segments in each file in the commit to save computational stretch. Table III depicts the proportional distribution of tagged changes across releases. Gradual trait transformation is deemed a usual phenomena in backport propagation. The first value of each cell represents the proportion of specific shift and the second/third value with $\uparrow$ / $\downarrow$ represents the proportional numeric of regression slope lean toward enhance/degrade threshold respectively.

C. Vulnerability Check

To distinguish whether the adaptive changes impose any threat to the target release, we used 'Safety' Python library [52] to check any existent third-party vulnerabilities in the backported code patches. It is important to note that vulnerable third-party libraries / frameworks incorporated into backports may have been admissible at the timeframe when it was backported. Nevertheless, it may be depreciated or declared vulnerable in today's context. Thus, we consider the time window of third-party vulnerable library lifetime using CVE security vulnerability database [53] and compare whether the vulnerability is enlisted during the usage scenario. In total, we found 42 vulnerabilities in Ansible, 37 in Kibana and 7 in Salt, which are well-documented and certified by the CVE dataset.

In addition, the adaptive changes can make the existing code entity vulnerable if the patch changes comprise data/functionality access modifier manipulation, improper data injection, or inappropriate architectural change. To address them, we used 'Shlex' library [54] to check lexical vulnerabilities that can potentially knit in the backported patch. This way, 21 vulnerabilities are detected in Ansible. These vulnerabilities impose potential threats to the stability and call for further large-scale research.

It is important to investigate whether those files that encounter quality degradation by receiving backporting change patches are likely to be harbouring these vulnerabilities. Identifying this correlation will assist in a better discernment of the challenges and vulnerable aspects of backported changes. We perform hypothesis testing to investigate how frequently vulnerable propagation occurs for those files whose quality is degraded for backport release maintenance. Thus, the files containing vulnerable propagation are tagged in the following manner:

1) if the vulnerability does not exist in file f_i for the final commit before a release of r_i, then given a set of backport commits $c \subset \{c_0, c_1, c_2 ... c_n\}$ are responsible for embodying a vulnerable issue, we tag it as $v_{gradual}$.
2) if the vulnerability is manifested in a commit on a file f_i which is also responsible for deviating the metrics value out of the existing threshold, we tag it as $v_{outliar}$.
3) if a vulnerability is pre-existing f_i for the final commit before a release of r_i, then backported commits are responsible for resolving them, we tag it as v_{fix}, else if its remains as it was, we tagged it v_{exists}.

For all the tagged files for vulnerabilities, we mapped whether they encountered quality degradation. Then the statistical hypothesis is performed with a confidence level of 5% (0.05) to visualize whether the vulnerabilities are associated with quality degradation or not. The outcome of the hypothesis testing is depicted in Table IV.

IV. Discussion

Table III shows that all three projects contain a significant proportion of gradual and outlier quality degradation that goes

Table III: Distribution of the tagged regression slope

Project	LOC			CBO			CC		
	Grad	Out	Not	Grad	Out	Not	Grad	Out	Not
Ansible	27% 64%↑ 36%↓	4.6% 53% 47%	68.4%	8.3% 50.8% 49.2%	12.2% 48% 52%	81.2%	11.2% 71% 29%	9.7% 58% 42%	79.1%
Kibana	18.5% 46.8% 52.2%	10.1% 42% 58%	71.4%	17.4% 33% 66%	2.8% 73% 27%	79.8%	5.6% 36.7% 63.3%	3.4% 37.5% 62.5%	91%
Salt	7.8% 60.5 39.5%	3.2% 51.4% 48.5%	89%	4.03% 33.3% 66.6%	0.37% 61.5% 38.5%	95.6%	9.1% 41.6% 58.4%	0.6% 62.4% 37.6%	90.5%

grad - Gradual shift, out - Outlier, not - Not shifted

Table IV: Correlation between vulnerable propagation and quality degradation

Project	H_0	P-value	Description
Ansible	Rejected	0.0489	Significantly correlated for Ansible releases
Kibana	Accepted	0.0834	Not Correlated
Salt	Accepted	0.239	Not Correlated

beyond the accepted threshold range of the metrics. For Ansible, project coupling and complexity were within the expected threshold for 81.2% and 79.1% files respectively. However, the rest proportions of the files are degraded by outliers or consecutive commit changes. This indicates that backports are indeed susceptible to quality degradation. Interestingly, most backports in Ansible that encounter quality degradation are likely to induce vulnerabilities in stable releases (in Table IV). Thus, further research is required for risk mitigation and efficient aiding of backporting for stable release maintenance.

V. Conclusion and Future work

The overwhelming number of outliers is persistent in backport propagations in stable versions, indicating that significant quality degradation is periodic and existent in stable release maintenance. This scenario can jeopardize the stability and compatibility of stable releases. Moreover, the vulnerability inspection infers that if a file is vulnerable to a specific branch, it is highly likely to encounter quality degradation (Ansible scenario), which calls for a further large-scale investigation into why a nd when they are likely to occur. Furthermore, the integrated set of metrics for each factor can furnish a better acuity of the evolution. Therefore, we would like comprehensive coverage of the metric set for our study in the future. Besides, we would incorporate more subject systems to discern how the distinct attributes of code entities degrade/enhance over a broad range. Comparing how quality deviations in backports differ from forward-ported commits can also provide helpful insight in the future.

Although we targeted the evolution trend in code proliferation in these three dimensions of size, complexity, and coupling in backports for the initial study, we acknowledge that the spectrum of code quality does not restrict to this scope. Indeed, numerous questions remain for further investigation, including whether the backport changes are cohesive, if they tend to be code smell prone over time, and how frequently technical debt roots are found in the change patches. Needless to say, the coverage of the future breakdown should not be tied to the quality scope of the stable release maintenance. Instead, this quest calls future researchers to concentrate on the vast scale of the procedure of the backporting strategies. For example, are there multiple strategies for backporting? How do the quality footprints vary with these strategies? What is the underlying reasoning for this deviation? Finding responses to these questions is crucial to satiate the motivation of providing an efficient yet safe backport management scheme for stable release maintenance.

VI. Acknowledgments

This research is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery grants, and by an NSERC Collaborative Research and Training Experience (CREATE) grant, and by two Canada First Research Excellence Fund (CFREF) grants coordinated by the Global Institute for Food Security (GIFS) and the Global Institute for Water Security (GIWS).

References

[1] D. Kirk, T. Crow, A. Luxton-Reilly, and E. Tempero, "On assuring learning about code quality," in *Proceedings of the Twenty-Second Australasian Computing Education Conference*, 2020, pp. 86–94.

[2] S. Farhang, J. Weidman, M. M. Kamani, J. Grossklags, and P. Liu, "Take it or leave it: a survey study on operating system upgrade practices," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 490–504.

[3] M. van Genuchten, R. Mans, H. Reijers, and D. Wismeijer, "Is your upgrade worth it? process mining can tell," *IEEE Software*, vol. 31, no. 5, pp. 94–100, 2014.

[4] P. O. Documentation, "Sunsetting python 2," https://www.python.org/doc/sunset-python-2/, May 2020.

[5] N. C. P. Notes, "Python 3 question and answers," https://python-notes.curiousefficiency.org/en/latest/python3/questions_and_answers.html#when-is-the-last-release-of-python-2-7-coming-out, 2012.

[6] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy, "If your version control system could talk," in *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, vol. 11. Citeseer, 1997.

[7] N. B. Ruparelia, "The history of version control," *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 1, pp. 5–9, 2010.

[8] L. Haaranen and T. Lehtinen, "Teaching git on the side: Version control system as a course platform," in *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, 2015, pp. 87–92.

[9] B. De Alwis and J. Sillito, "Why are software projects moving from centralized to decentralized version control systems?" in *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. IEEE, 2009, pp. 36–39.

[10] Redhat, "Backporting security fixes," https://access.redhat.com/security/updates/backporting, 2022.

[11] Crowdstrike, "What is backporting?" https://www.crowdstrike.com/cybersecurity-101/backporting/#, 2022.

[12] O. L. Blog, "Backporting patches using git," https://blogs.oracle.com/linux/post/backporting-patches-using-git, 2022.

[13] D. Chakroborti, K. A. Schneider, and C. K. Roy, "Backports: Change types, challenges and strategies," in *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, 2022, pp. 636–647.

[14] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 97–106.

[15] R. Ramsauer, D. Lohmann, and W. Mauerer, "Observing custom software modifications: a quantitative approach of tracking the evolution of patch stacks," in *Proceedings of the 12th International Symposium on Open Collaboration*, 2016, pp. 1–4.

[16] X. Zhang, Y. Yu, G. Georgios, and A. Rastogi, "Pull request decisions explained: An empirical overview," *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.

[17] D. M. German, G. Robles, G. Poo-Caamaño, X. Yang, H. Iida, and K. Inoue, "" was my contribution fairly reviewed?" a framework to study the perception of fairness in modern code reviews," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 523–534.

[18] G. Gousios and A. Zaidman, "A dataset for pull-based development research," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 368–371.

[19] R. Pham, L. Singer, O. Liskin, F. F. Filho, and K. Schneider, "Creating a shared understanding of testing culture on a social coding site," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 112–121.

[20] M. Lavallée and P. N. Robillard, "Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 677–687.

[21] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.

[22] M. Mondal, C. K. Roy, and K. A. Schneider, "Insight into a method co-change pattern to identify highly coupled methods: An empirical study," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 103–112.

[23] S. Zaharia, T. Rebedea, and S. Trausan-Matu, "Source code vulnerabilities detection using loosely coupled data and control flows," in *2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2019, pp. 43–46.

[24] A. Uchôa, C. Barbosa, W. Oizumi, P. Blenílio, R. Lima, A. Garcia, and C. Bezerra, "How does modern code review impact software design degradation? an in-depth empirical study," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 511–522.

[25] F. Thung, X.-B. D. Le, D. Lo, and J. Lawall, "Recommending code changes for automatic backporting of linux device drivers," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 222–232.

[26] B. Ray, M. Kim, S. Person, and N. Rungta, "Detecting and characterizing semantic inconsistencies in ported code," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 367–377.

[27] L. R. Rodriguez and J. Lawall, "Increasing automation in the backporting of linux drivers using coccinelle," in *2015 11th European Dependable Computing Conference (EDCC)*, 2015, pp. 132–143.

[28] J. Lawall, D. Palinski, L. Gnirke, and G. Muller, "Fast and precise retrieval of forward and back porting information for linux device drivers," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 15–26.

[29] A. Decan, T. Mens, A. Zerouali, and C. De Roover, "Back to the past–analysing backporting practices in package dependency networks," *IEEE Transactions on Software Engineering*, 2021.

[30] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

[31] A. Decan, T. Mens, A. Zerouali, and C. De Roover, "Back to the past – analysing backporting practices in package dependency networks," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[32] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 682–691.

[33] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 177–187.

[34] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "Code quality analysis in open source software development," *Information systems journal*, vol. 12, no. 1, pp. 43–60, 2002.

[35] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 155–165.

[36] M. M. Lehman and L. A. Belady, *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[37] G. Xie, J. Chen, and I. Neamtiu, "Towards a better understanding of software evolution: An empirical study on open source software," in *2009 ieee international conference on software maintenance*. IEEE, 2009, pp. 51–60.

[38] D. L. Parnas, "Software aging," in *Proceedings of 16th International Conference on Software Engineering*. IEEE, 1994, pp. 279–287.

[39] G. Xie, J. Chen, and I. Neamtiu, "Towards a better understanding of software evolution: An empirical study on open source software," in *2009 ieee international conference on software maintenance*. IEEE, 2009, pp. 51–60.

[40] H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[41] J. Tasnim, D. Chakroborti, K. Schneider, and C. Roy, "How does quality deviate in stable releases by backporting?" Jan. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7552632

[42] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, and C. Soubervielle-Montalvo, "Source code metrics: A systematic mapping study," *Journal of Systems and Software*, vol. 128, pp. 164–197, 2017.

[43] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[44] Python, "Python : Radon," https://pypi.org/project/radon/, May 2013.

[45] M. P. Malhotra, M. K. Shah, J. Rathod, and M. Mehta, "Python based software for calculating cyclomatic complexity," *International Journal of Innovative Science, Engineering and Technology*, vol. 2, no. 3, pp. 546–549, 2015.

[46] M. Lipow, "Number of faults per line of code," *IEEE Transactions on software Engineering*, no. 4, pp. 437–439, 1982.

[47] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante, "Cyclomatic complexity," *IEEE software*, vol. 33, no. 6, pp. 27–29, 2016.

[48] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[49] J. J. Vinju and M. W. Godfrey, "What does control flow really look like? eyeballing the cyclomatic complexity metric," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2012, pp. 154–163.

[50] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *2010 IEEE international conference on software maintenance*. IEEE, 2010, pp. 1–10.

[51] A. Mori, G. Vale, M. Viggiato, J. Oliveira, E. Figueiredo, E. Cirilo, P. Jamshidi, and C. Kastner, "Evaluating domain-specific metric thresholds: an empirical study," in *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, 2018, pp. 41–50.

[52] Python, "Python : Safety," https://pypi.org/project/safety/, May 2013.

[53] P. C. check, "Python : Vulnerability statistics," https://www.cvedetails.com/product/18230/Python-Python.html?vendor_id=10210, May 2013.

[54] S. lexical analysis, "Python : shlex," https://docs.python.org/3/library/shlex.html, May 2013.