Linguistic Documentation of Software History

Miroslav Tushev* and Anas Mahmoud†
The Division of Computer Science and Engineering
Louisiana State University
*mtushe1@lsu.edu, †amahmo4@lsu.edu

***Abstract*—Open Source Software (OSS) projects start with an initial vocabulary, often determined by the first generation of developers. This vocabulary, embedded in code identifier names and internal code comments, goes through multiple rounds of change, influenced by the interrelated patterns of human (e.g., developers joining and departing) and system (e.g., maintenance activities) interactions. Capturing the dynamics of this change is crucial for understanding and synthesizing code changes over time. However, existing code evolution analysis tools, available in modern version control systems such as *GitHub* and *SourceForge*, often overlook the linguistic aspects of code evolution. To bridge this gap, in this paper, we propose to study code evolution in OSS projects through the lens of developers' language, also known as code lexicon. Our analysis is conducted using 32 OSS projects sampled from a broad range of application domains. Our results show that different maintenance activities impact code lexicon differently. These insights lay out a preliminary foundation for modeling the linguistic history of OSS projects. In the long run, this foundation will be utilized to provide support for basic program comprehension tasks and help researchers gain new insights into the complex interplay between linguistic change and various system and human aspects of OSS development.**

I. Introduction

Large OSS projects, maintained by large distributed developer teams, tend to have a rich history. Surveys of software professionals revealed that software history is indispensable for developers [1], [2]. Developers frequently ask questions about the history of code to understand change rationale, track bugs to their origin, or trace features to their older versions. However, current version control systems (e.g. *GitHub* and *SourceForge*) provide a single lens on history through commits. Commits record changes per code file and line along with developers' descriptions of these changes. However, a large percentage of change information is rarely documented [3]. Such non-informative commits have been identified as one of the main challenges facing OSS developers examining software history [4], [5].

To address these limitations, in this Early Research Achievements (ERA) paper, we propose a new window on software history. In particular, we analyze software evolution through the lens of developers' language [6], [7]. The vocabulary of this language, embedded in code identifier names, makes up around 70% of code lexicon [8]. Research on program comprehension revealed that identifiers capture developers' understanding of their system and its application domain at the most primitive level. Such information can be crucial for developers during maintenance sessions [8], [9], [10], [11], [12], [13], [14], [15]. Comments were also found to play a paramount role in arriving at a correct understanding of the program, especially for newcomers who do not have an adequate experience in the internals of the system [7], [16].

In the context of software, language evolution is typically influenced by code evolution activities such as refactoring, feature addition, and bug fixes. In our analysis, we hypothesize that these different software evolution activities have different impacts on the system's vocabulary. Therefore, the ability to capture and model changes in the system vocabulary is expected to uncover the history of events that led to these changes, revealing unique aspects of code evolution that are typically overlooked by existing methods used in modern version control systems, such as *diff* functions or Abstract Syntax Trees (ASTs) [3], [4], [17], [18], [19].

Our analysis is conducted using 32 open source systems, including their revisions and metadata. Our objective is to explore the specific impacts of different maintenance tasks on the linguistic identity of OSS projects and provide a preliminary evidence on the nature and magnitude of this impact.

II. Foundation and Analysis

In this paper, we aim to investigate linguistic change in OSS projects. To conduct our analysis, we adapt Petersen et al.'s [20] statistical model of natural language evolution to OSS projects' code lexicon. According to this model, a *survival-of-the-fittest* effect controls the way words emerge, grow, and vanish throughout human history. Specifically, words are competing actors in a system of finite resources. Words can gain or lose momentum influenced by historical events (e.g., war), new innovations (e.g., penicillin), and socio-technological advances (e.g., the Internet) [21], [22], [23]. Such information can be used to investigate linguistic trends quantitatively and explore questions deeply-rooted in cultural anthropology [20], [24].

In OSS development, the frequent maintenance actions performed on the system as well as the highly dynamic nature of developer teams, apply similar evolutionary pressures on the usage and survival capacity of code lexicon words [25]. To understand and quantify the magnitude of this change, we investigate the impact of different software maintenance activities on OSS code lexicon. In general, we identify three generic categories of such activities:

- **Bug fixes:** these activities include corrective maintenance requests, mainly targeting bugs in the system, or errors in the program's logic.

- **Feature additions:** these activities typically include requests for new major or minor functionality to be added to the system.
- **Improvements:** improvement activities typically include perfective, adaptive, and preventive maintenance requests, focusing on improving existing code by, for instance, enhancing the efficiency of underlying algorithms, improving code structure by refactoring, or improving interfaces.

Resolving the relationship between these different activities and code change will provide a fundamental understanding of the dynamics of linguistic change in OSS projects and bridge a very important gap in code evolution research. In what follows, we describe our data collection and analysis process in greater detail.

A. Data Collection

To understand the relationship between maintenance activities and linguistic change, we selected 32 projects from GitHub, covering four programming languages: Java, C#, Python, and JavaScript. The criteria for selecting the projects were ***a)*** the project should have a long history, or a large number of releases, ***b)*** the project should be relatively popular, which can be quantified through the number of stars the project received on *GitHub* [26], and ***c)*** the maintenance activities for each release of the project should be explicitly classified by project maintainers. For example, the Java project *NewPipe* in our dataset marks feature additions as *New* (e.g., `New: Basic Media Support`), bugs as *Fixed* (e.g., `Fix random popup player crash #2133`), and other perfective tasks as *Improvements* (e.g., `Improvement: clearing watch history using options menu`). This criterion was enforced to ensure the validity of the results. Specifically, it can be challenging to accurately classify the type of maintenance activity if the issues related to the activity are not explicitly defined [27], [28].

To collect our data, we used the GitHub API [29] to download the top starred Java, C#, Python, and JavaScript projects (8 x 4) along with their corresponding public releases. This API provides a convenient way for directly downloading project data (e.g., releases, commits, issues, and contributors), thus enabling access to all forms of events instigated by code evolution activities. Selecting a *smaller* set of well-maintained projects helps to mitigate the data validity threats often associated with running experimentation on large-scale datasets of OSS projects [27]. The descriptive statistics of our 32 selected projects are provided in Table I.

To extract source code lexicon from our projects, we used regular expressions. Code lexicon consists of all words used in source code except for any keywords reserved for the programming language itself. Regular expressions treat code artifacts as raw text files. Therefore, the code itself does not have to compile, or even be complete, for regular expressions to work. After identifiers are extracted, we further split any compound words into their constituent words based on camel-casing (e.g., `userID` is split into `User` and `ID`) or any special characters (e.g. underscore) typically used in code naming conventions (e.g., `file_type` is split into `file` and `type`). After atomic words of code identifiers are extracted, stemming is applied to reduce words to their morphological roots [30]. The accuracy of our indexing tool (%97) has been independently verified in our previous work [31].

B. Analysis and Results

The objective of our analysis is to examine whether linguistic change can be predicted by the different maintenance activities performed on the project. To expose such effect, we use Granger Causality, an econometric technique that is used to test if one variable precedes another in a stationary time-series [32]. Precisely, given the variables *A* and *B*, we test to see if the values of *A* and the previous history of *B* predict the values of *B* better than the history of *B* alone. The assumption behind using this test is that the different releases of the system can be represented as points in a time series.

One of the parameters of Granger Causality is *lag*. The operationalization of lag depends on the problem, or the context. In our case, a lag of 1 represents the value of linguistic change for release *i* and maintenance activities submitted in the release notes for the same release *i*. Lag of 2 measures the correlation between linguistic change of *i*-th release and maintenance activities submitted for *i-1* release, and so on. Usually, in time series analysis, several lag values are tested. In our analysis, we measure lag of 1 and 2 for short term impacts and lag of 5 to capture long term impacts.

To quantify the magnitude of linguistic shift in OSS projects, we rely on Petersen et al.'s word-frequency model [20]. According to this model, the linguistic change rate ($\Delta\lambda_{i,j}$) between two releases r_i and r_j of the system can be calculated as the number of different words (words birth and death) between the two releases divided by the number of unique words in both releases. The average λ_s of a system s which has n releases can be calculated as the average $\lambda_{i,j}$ between each two consecutive releases in the time series of the system:

$$\Delta\lambda_{i,j} = 1 - \frac{|r_i \cap r_j|}{|r_i \cup r_j|}, \quad \lambda_s(t) = \frac{1}{n}\sum_{i=1}^{n-1}(\Delta\lambda_{i,i+1}) \qquad (1)$$

As an example of our analysis, Fig. 1 shows linguistic change at different releases of *Synapse*—an open source research collaborative platform—in comparison to the number of bug fixes, feature additions, and improvements for each release. This word-frequency model can be more sensitive to change than more coarse-grained, or topic based, models such as Latent Dirichlet Allocation (LDA) [33]. Specifically, the operational complexity and the inherent sparsity of the textual information of code often leads to generating incoherent topics that can hardly describe change [34], [35], [36], [37].

The results of our Granger Causality analysis are presented in Table II. In general, the results show that feature additions were the most significant predictors of linguistic change, followed by improvements and bug fixes, which were able to significantly predict linguistic change in a smaller number

TABLE I: Descriptive statistics for our sample of 32 projects, including the total lines of code (LOC) and average number of releases, bug fixes, feature additions, and improvements for the set of projects sampled from each programming language.

Language	LOC	Releases	Bug fixes	Feature Additions	Improvements
Java	8.10M	61	270	186	306
C#	7.95M	80	272	197	123
Python	14.39M	87	308	145	291
JavaScript	14.89M	117	588	193	209

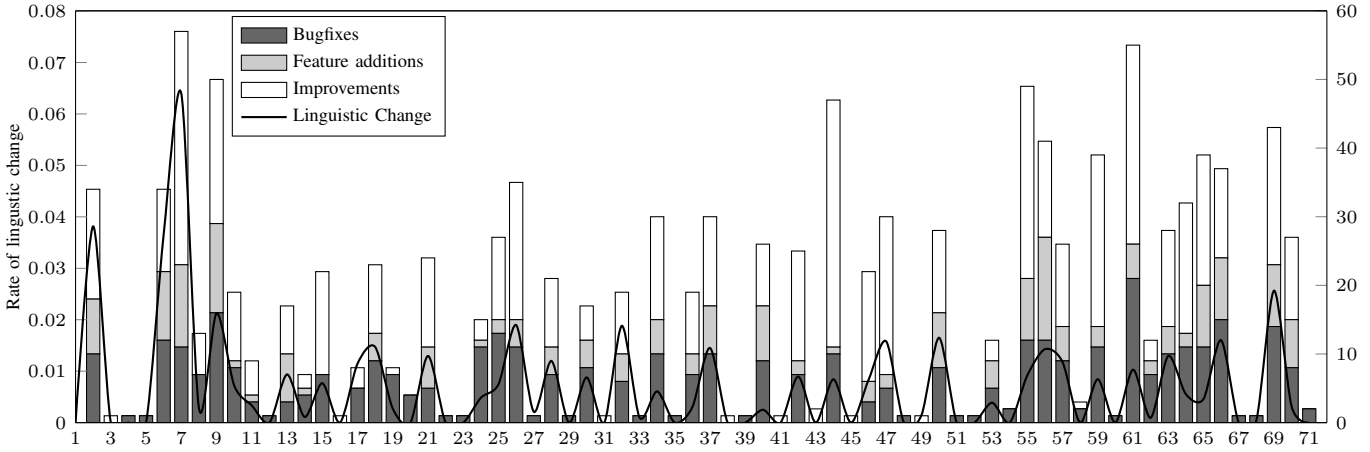

Fig. 1: Linguistic Change (Eq. 1) for *Synapse*, compared to the number of bug fixes, feature additions, and improvements

of projects. We also notice the differences between different programming languages, for instance, bug fixes significantly predicted linguistic change for five out of the eight C# projects. For Python projects, feature additions and improvements caused the most disturbance in code lexicon. In contrast, JavaScript projects showed significantly less linguistic sensitivity to feature additions and improvements. Furthermore, for the majority of the projects where the lag of 1 is significant, the lags of 2 and 5 are also significant, which suggests that bug fixes, feature additions, and improvements have a long-term impact on linguistic change, not limited by a single release.

Different maintenance activities can predict linguistic change at different levels of accuracy. Feature additions are the best predictors of change. However, the magnitude of the effect seems to be influenced by the programming language.

III. Conclusion, Impact, and Next Steps

In this paper, we conducted a preliminary analysis, using a frequency-based statistical model of language evolution [20], to study linguistic change in OSS projects. Our analysis revealed that different types of software maintenance activities have different impacts on code lexicon. Overall, the effect of maintenance activities is non-uniform. This effect even varies among programming languages, indicating that different maintenance activities might have different linguistic footprints.

In terms of practical impact, resolving the micro relations between change in code lexicon and the different types of maintenance activities can help developers to diagnose and reverse the symptoms of code aging, make informed design and maintenance decisions, and ensure a sustainable and stable delivery process. For instance, capturing the linguistic footprints of maintenance activities can help developers to pinpoint the specific activities responsible for linguistic antipatterns (LAs), or inconsistencies among the naming of a code entity. LAs were found to be a main symptom of code aging. Such linguistic anomalies, often introduced during maintenance sessions, lead to a steeper learning curve, misunderstandings, and eventually bug-prone code in the project [38], [39], [40]. Therefore, linguistic change analysis can be very important for the stability of OSS projects, given that OSS environments often lack an organizational structure that enforces the conformance to a specific naming convention [6].

Our work can also have practical impacts on OSS projects sustainability. For instance, creativity is often defined as a function of new features proposed and implemented [41], [42]. Our analysis showed that new features came with new vocabulary, thus more linguistic change. However, higher levels of linguistic change might destabilize the project. In OSS environments, project stability is a necessity for quality control [43]. Therefore, resolving the interdependency relations between linguistic change, creativity, and quality can help to define the levels of linguistic change that can be optimal for keeping the system's quality under control, at the same time, do not restrain OSS developers' creativity.

TABLE II: Granger Causality results for each individual project in our analysis. The variables provided are predictors of Linguistic Change. ***p**-values* are indicated: **p<0.05; †p < 0.01; ‡p < 0.001*. Empty cells indicate that no change of that specific type was reported in the project.

	Bug fix			**Feature addition**			**Improvement**		
Lag	**1**	**2**	**5**	**1**	**2**	**5**	**1**	**2**	**5**
				C#					
aspboilerplatenet	0.912	0.952	0.970	0.190	0.126	0.429	0.348	0.475	0.703
cake	0.000‡	0.000‡	0.003†	0.000‡	0.000‡	0.000†	0.000‡	0.000‡	0.002†
optikey	0.000‡	0.002†	0.000‡	0.041*	0.038*	0.296			
ckan	0.000‡	0.000‡	0.003†	0.000‡	0.001†	0.057	0.024†	0.070	0.218
azure-pipelines	0.275	0.503	0.434	0.002†	0.014*	0.242	0.002†	0.011*	0.116
VisualStudio	0.000‡	0.000‡	0.002†	0.005†	0.013*	0.218	0.079	0.054	0.2248
elasticnet	0.5067	0.7769	0.5209	0.4518	0.7887	0.6495			
hearthstone	0.000‡	0.000‡	0.012†	0.000‡	0.000‡	0.000‡			
				Java					
flym	0.000‡	0.000‡	0.006†	0.000‡	0.000‡	0.001†	0.000‡	0.000‡	0.000‡
RxJava	0.966	0.941	0.878				0.795	0.956	0.826
mongo-java-driver	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡
qksms	0.199	0.315	0.705	0.009†	0.034*	0.020*	0.356	0.228	0.621
checkstyle	0.001‡	0.012*	0.009†	0.000‡	0.000‡	0.000‡			
filedownloader	0.917	0.7102	0.809	0.000‡	0.002†	0.002†	0.000‡	0.000‡	0.000‡
newpipe	0.784	0.489	0.452	0.000‡	0.000‡	0.004†	0.801	0.025*	0.046*
android-catcher	0.001†	0.002†	0.002†	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡
				Python					
synapse	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡
erpnext	0.267	0.211	0.575	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡
tautulli	0.935	0.942	0.662	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.001†
kinto	0.501	0.138	0.012*	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡
aiohttp	0.000‡	0.000‡	0.022*	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡
netbox	0.662	0.756	0.721	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡
h	0.164	0.417	0.209	0.000‡	0.002†	0.117	0.012*	0.051	0.056
conan	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡
				JavaScript					
RocketChat	0.015*	0.031†	0.098	0.001†	0.005†	0.008†	0.970	0.891	0.938
ghost	0.136	0.314	0.278	0.000‡	0.000‡	0.000‡	0.692	0.872	0.184
habitica	0.677	0.904	0.918	0.352	0.601	0.709	0.474	0.764	0.510
vuejs	0.818	0.318	0.776	0.519	0.859	0.639	0.481	0.628	0.942
redux-form	0.358	0.495	0.967	0.998	0.968	0.769			
semantic-ui	0.000‡	0.000‡	0.001†	0.626	0.764	0.947	0.000‡	0.000‡	0.000‡
vuetify	0.000‡	0.000‡	0.010*	0.000‡	0.000‡	0.000‡	0.000‡	0.000‡	0.001†
stylelint	0.002†	0.010*	0.001‡	0.000‡	0.000‡	0.000‡			

In terms of theoretical impact, our preliminary work in this paper lays out a theoretical foundation for describing and modeling the evolution of developers' language in OSS projects. Our aim is to provide a fundamental understating of how such language emerges and becomes shaped by different models of language selection. According to Croft's [44], understanding language change at a micro- and macro-levels provides a basis for understanding the generation and propagation of language structures, thus provides a description of how a language system may emerge and continue to change over time. The overarching goal in this paper is to advance the state-of-the-art in theoretical software engineering research by articulating a unified theory of linguistic change in OSS projects. According to Sjøberg et al., in order for software engineering to develop into a mature field of science, theory-building should become an integral part of its research and practice [45]. In the long run, a well-defined theory of code lexicon evolution will serve as a core asset that researchers can utilize to quantitatively investigate linguistic trends in OSS and uncover best practices for maintaining successful OSS projects and building vibrant OSS communities.

ACKNOWLEDGMENT

This research is supported by the U.S. National Science Foundation (Award CCF 1821525).

REFERENCES

[1] M. Codoban, S. Ragavan, D. Dig, and B. Bailey, "Software history under the lens: a study on why and how developers examine it," in *IEEE International Conference on Software Maintenance and Evolution*, 2015, pp. 1–10.

[2] S. Grant, J. Cordy, and D. Skillicorn, "Reverse engineering co-maintenance relationships using conceptual analysis of source code," in *Working Conference on Reverse Engineering*, 2011, pp. 87–91.

[3] T. Zimmermann, S. Kim, A. Zeller, and J. Whitehead, "Mining version archives for co-changed lines," in *Mining Software Repositories*, vol. 6, 2006, pp. 72–75.

[4] F. Servant and J. Jones, "History slicing: assisting code-evolution tasks," in *International Symposium on the Foundations of Software Engineering*, 2012, pp. 43:1–43:11.

[5] I. Steinmacher, M. Silva, M. A. Gerosa, and D. Redmiles, "A systematic literature review on the barriers faced by newcomers to open source software projects," *Information and Software Technology*, vol. 59, pp. 67–85, 2015.

[6] M. Allamanis, E. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 281–293.

[7] S. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "Lexicon bad smells in software," in *Working Conference on Reverse Engineering*, 2009, pp. 95–99.

[8] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006.

[9] E. Host and B. Ostvold, "The programmer's lexicon, volume i: The verbs," in *International Working Conference on Source Code Analysis and Manipulation*, 2007, pp. 193–202.

[10] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? A study of identifiers," in *International Conference on Program Comprehension*, 2006, pp. 3–12.

[11] N. Anquetil and T. Lethbridge, "Assessing the relevance of identifier names in a legacy software system," in *Centre for Advanced Studies on Collaborative Research*, 1998, p. 4.

[12] P. Rodeghero, "Discovering important source code terms," in *International Conference on Software Engineering Companion*, 2016, pp. 671–673.

[13] D. Lawrie, H. Feild, and D. Binkley, "Quantifying identifier quality: an analysis of trends," *Empirical Software Engineering*, vol. 12, no. 4, pp. 359–388, 2007.

[14] E. Avidan and D. Feitelson, "Effects of variable names on comprehension an empirical study," in *International Conference on Program Comprehension*, 2017, pp. 55–65.

[15] G. Beniamini, S. Gingichashvili, A. Klein Orbach, and D. Feitelson, "Meaningful identifier names: The case of single letter variables," in *International Conference on Program Comprehension*, 2017, pp. 45–54.

[16] S. Haiduc and A. Marcus, "On the use of domain terms in source code," in *International Conference on Program Comprehension*, 2008, pp. 113–122.

[17] H. Nguyen, A. Nguyen, T. Nguyen, T. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *International Conference on Automated Software Engineering*, 2013, pp. 180–190.

[18] H. Breivold, I. Crnkovic, and M. Larsson, "A systematic review of software architecture evolution research," *Information and Software Technology*, vol. 54, no. 1, pp. 16–40, 2012.

[19] I. Neamtiu, J. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[20] A. Petersen, J. Tenenbaum, S. Havlin, and E. Stanley, "Statistical laws governing fluctuations in word use from word birth to word death," *Scientific reports*, vol. 2, p. 313, 2012.

[21] J. Michel, Y. Shen, A. Aiden, A. Veres, M. Gray, J. Pickett, D. Hoiberg, D. Clancy, P. Norvig, J. Orwant *et al.*, "Quantitative analysis of culture using millions of digitized books," *Science*, vol. 331, no. 6014, pp. 176–182, 2011.

[22] M. Pagel, Q. Atkinson, and A. Meade, "Frequency of word-use predicts rates of lexical evolution throughout Indo-European history," *Nature*, vol. 449, no. 7163, p. 717, 2007.

[23] R. Solé, B. Corominas-Murtra, and J. Fortuny, "Diversity, competition, extinction: the ecophysics of language change," *Journal of The Royal Society Interface*, vol. 7, pp. 1647–1664, 2010.

[24] E. Lieberman, J. Michel, J. Jackson, T. Tang, and M. Nowak, "Quantifying the evolutionary dynamics of language," *Nature*, vol. 449, p. 713, 2007.

[25] M. Tushev, S. Khatiwada, and A. Mahmoud, "Linguistic change in open source software," in *International Conference on Software Maintenance and Evolution*, 2019, pp. 296–300.

[26] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," in *Joint Meeting on Foundations of Software Engineering*, 2015, pp. 805–816.

[27] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[28] L. Hattori and M. Lanza, "On the nature of commits," in *International Conference on Automated Software Engineering*, 2008, pp. 63–71.

[29] V. Cosentino, J. Luis, and J. Cabot, "Findings from GitHub: methods, datasets and limitations," in *International Conference on Mining Software Repositories*, 2016, pp. 137–141.

[30] M. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[31] S. Khatiwada, M. Kelly, and A. Mahmoud, "STAC: A tool for static textual analysis of code," in *IEEE International Conference on Program Comprehension*, 2016, pp. 1–3.

[32] C. Granger, "Investigating causal relations by econometric models and cross-spectral methods," *Econometrica*, vol. 37, no. 3, pp. 424–438, 1969.

[33] D. Blei, A. Ng, and M. Jordan, "Latent Dirichlet Allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[34] A. DeLucia, M. DiPenta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR methods for labeling source code artifacts: Is it worthwhile?" in *International Conference on Program Comprehension*, 2012, pp. 193–202.

[35] S. Thomas, B. Adams, A. Hassan, and D. Blostein, "Validating the use of topic models for software evolution," in *IEEE Working Conference on Source Code Analysis and Manipulation*, 2010, pp. 55–64.

[36] A. Mahmoud and G. Bradshaw, "Semantic topic models for source code analysis," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1965–2000, 2017.

[37] A. Panichella, B. Dit, R. Oliveto, M. DiPenta, D. Poshynanyk, and A. D. Lucia, "How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms," in *International Conference on Software Engineering*, 2013, pp. 522–531.

[38] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Software Engineering*, vol. 21, p. 2016, 104–158.

[39] E. Aghajani, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on linguistic antipatterns affecting APIs," in *International Conference on Software Maintenance and Evolution*, 2018, pp. 25–35.

[40] S. Fakhoury, D. Roy, S. A. Hassan, and V. Arnaoudova, "Improving source code readability: Theory and practice," in *International Conference on Program Comprehension*, 2019, pp. 2–12.

[41] J. Paulson, G. Succi, and A. Eberlein, "An empirical study of open source and closed source software products," *IEEE Transactions on Software Engineering*, vol. 30, no. 4, pp. 246–256, 2004.

[42] Y. Kidane and P. Gloor, "Correlating temporal communication patterns of the eclipse open source community with performance and creativity," *Computational and Mathematical Organization Theory*, vol. 13, no. 1, pp. 17–27, 2007.

[43] G. Gousios, M. Storey, and A. Bacchelli, "Work practices and challenges in pull based development: The contributor's perspective," in *International Conference on Software Engineering*, 2016, pp. 285–296.

[44] W. Croft, *Evolution: Language Use and the Evolution of Languages*. Springer Berlin Heidelberg, 2013, pp. 93–120.

[45] D. Sjøberg, T. Dybå, B. Anda, and J. Hannay, *Building Theories in Software Engineering*, F. Shull, J. Singer, and D. Sjøberg, Eds., 2008.