On the Evolutionary Properties of Fix Inducing Changes

Syed Fatiul Huq[a], Md. Aquib Azmain[b], Nadia Nahar[c] and Md. Nurul Ahad Tawhid[d]

Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

Abstract

A major aspect of maintaining the quality of software systems is the management of bugs. Bugs are commonly fixed in a corrective manner; detected after the code is tested or reported in production. Analyzing Fix-Inducing Changes (FIC) — developer code that introduces bugs — provides the opportunity to estimate these bugs proactively. This study analyzes the evolution of FICs to visualize patterns associated with the introduction of bugs throughout and within project releases. Furthermore, the association between FICs and complexity metrics, an important element of software evolution, is extracted to quantify the characteristics of buggy code. The findings indicate that FICs become less frequent as the software evolves and more commonly appear in the early stages of individual releases. It is also observed that FICs are correlated to longer Commit intervals. Lastly, FICs are found to be more present in codes with fewer lines and less cyclomatic complexity, which corresponds with the law of growing complexity in software evolution.

Keywords

software evolution, fix-inducing changes, data mining,

1. Introduction

Software projects evolve over time [1] to introduce new features while fixing bugs [2] that appear in parallel. The conventional way of handling the bugs is by detecting faulty codes with test cases [3] based on user reports and writing patches [4] that eliminate the fault. In this way, a bug is fixed only after it is written. Another way of managing bugs is proactively understanding how bugs occur in systems. In this preventive process, Fix-Inducing Changes (FICs) — code that introduces bugs which induce a later fix [5] — are analyzed. FICs can be tracked from a project's change history by looking for instances of bug fixes and the code changed in these fixes. An FIC provides information about the code changes, the developer writing the bug, and the state of the development process at the time of introducing the bug. These can unveil important characteristics of the project, processes and developers that potentially cause bugs.

Studies analyzing FICs have observed how these are related to or affected by properties of the software development lifecycle. For instance, Sliwerski et al. [5], apart from coining the term, related FICs with two developer activities: the day of coding and the amount of code in a single Commit. Yin et al. [6] observed how bug fixes themselves can introduce new bugs. Other studies include relations with code smells[7], code coupling [8], developer sentiment [9, 10] and more.

Since FICs are a component of the software's history, these hold information about the software's evolution. As software evolves, so does its management, personnel, design and code. This changing environment can affect the introduction of bugs and vice versa. The existence and properties of such correlations can be established by analyzing the evolution of FICs. Observing the evolution of FICs can also help uncover its relation with other evolutionary factors, for instance, the system's complexity [11]. With this aim, this study answers the following Research Questions (RQs):

RQ1: How do FICs evolve with the software? This RQ observes how FICs change in frequency and ratio as the software system evolves. The evolution of the software is measured with its releases.

RQ2: How do FICs exist within releases? A single release depicts the software team's complete flow of activities. The flow starts with the team taking in new requirements to update the features of the software to their finalization, testing and deployment. This RQ observes how FICs appear and change in this flow.

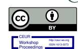
RQ3: How do FICs relate to Commit interval? The interval between Commits signify the amount of tasks assigned to developers, along with gaps between activities. This RQ answers whether FICs behave differently than regular Commits in terms of these intervals.

RQ4: How do FICs relate to system complexity? According to Lehman's law of evolution, system complexity is a vital part of a software's evolution [11]. The law dictates that complexity increases as the software evolves. Since FICs are instances where bugs are introduced, and bugs can be affected by system complexity, this RQ observes the relation between the two entities. Specifically, in this RQ, FIC is correlated to Lines of Code (LoC) and Cyclomatic Complexity (CC) as commonly used metrics to quantify complexity [12, 13].

QuASoQ 2020: International Workshop on Quantitative Approaches to Software Quality, 1st December 2020, Singapore

EMAIL: bsse0732@iit.du.ac.bd (S. F. Huq); bsse0718@iit.du.ac.bd (Md. A. Azmain); nadia@iit.du.ac.bd (N. Nahar); tawhid@iit.du.ac.bd (Md. N. A. Tawhid)

© 2020 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).


CEUR Workshop Proceedings (CEUR-WS.org)

For this study, eight Java repositories from GitHub with a total of 142,555 Commits have been analyzed. From the Commits, FICs are detected, release information are extracted and relevant metrics are calculated. The findings show that FICs decrease as the software evolves, while remaining more prevalent early in release cycles. Statistically analyzing the data shows that FICs contain larger intervals and their code reduced LoC and CC than regular Commits.

2. Related Work

Sliwerski et al.[5] introduced the term Fix-Inducing Changes (FIC), providing a process that detects FICs in version history from Concurrent Versions System (CVS) with bug reports from Bugzilla. Moreover, they showed a relation between FICs and number of files changed. Antoniol et al.[14] showed that FICs create adverse effect and produce unexpected results. They presented a robust approach to detect groups of co-changing files in CVS repositories.

Yin et al.[6] identified and analyzed incorrect bug fixes which introduce new ones instead. They analyzed the code of operating systems namely, FreeBSD, Linux and OpenSolaris. This approach also combined version control systems and bug repository to categorize changes. They proved that Fix Inducing Fix (FIF) can cause crashes, hangs, data corruption or security problems.

Bavota et al.[7] showed that 15% refactoring tasks induce bugs, analyzing 52 kinds of refactoring on 3 Java projects. They detected inheritance related refactoring as the most error-prone refactoring.

In order to analyze FICs, various works focused on different properties of change that would induce the bugs. Levin et al.[15] and Menzies et al.[16] focused on source code changes of affected files. Fukushima et al. [17] introduced developer experience, time of day, time interval of commit and some other properties of change that would induce bugs. Sadiq et al. [8] related FICs with change couplings to find that recent change couples provide better insight on new errors. Huq et al. [9] showed that developer sentiment is related with FICs, where positive comments and reviews in Pull Requests can lead to buggy Commits.

Weicheng et al.[18] explored the relation between developer Commit patterns in GitHub and software evolution. They used four metrics to measure the Commit activity of developers and code evolution: changes, interval, author and source code dependency. Moreover, this paper showed techniques to visualize these metrics for a given project. They developed a tool named Commits Analysis Tool (CAT) that finds that the changes in previous versions can affect the file which is dependent on it in the next version.

Osman et al.[19] extracted bug-fix patterns by mining change history of 717 open source projects. They manually inspected the patterns to retrieve the context and reasons that cause those bugs.

So far, FICs have been analyzed to derive relationships with different project metrics. While the evolution of Commits has been observed, the evolutionary properties of FICs have yet been studied.

3. Methodology

This study observes how Fix-Inducing Changes (FIC) evolve throughout the lifetime of software projects and how these relate to complexity metrics. The methodology of the study is divided into three parts, described as follows.

3.1. Fix-Inducing Change (FIC) Detection

FICs are changes to code that causes problems to the software system. FICs are the introduction of bugs or errors to the software, inducing fixes in the future. Hence, these can be detected from the changes that fix bugs and errors.

This study utilizes Commits, the documented changes in software projects that are managed through version controlling. Commits contain the exact lines and files where changes are made along with information of and message from the developer who posted these. The detection of FICs through Commits is conducted in the following steps, influenced by the process of [7]

1. All Commits are fetched from GitHub repositories.
2. Commit messages are extracted to detect terms such as “bug”, “fix” or “patch”. These terms signify that the aim of the Commit is related to the management of bugs.
3. Now the changes in these Commits are analyzed. Since the study deals with Java projects, it is first checked whether the changes occur in “.java” files. Commits with no changes to such files indicate that the Commits dealt with non-code entities of the software (configuration files, documentation etc.). Furthermore, the changes made in “.java” files are analyzed to see whether the changes were code comments, which also signifies the absence of code entities.
4. Then, the type of the edit made by the Commit is checked. There are three types of edit: Insert, Delete and Replace. An Insert edit means that a patch code is added onto the existing code base. However, it does not help to track which part of the previous code was buggy. There is no way of tracking back to a Commit that introduced a

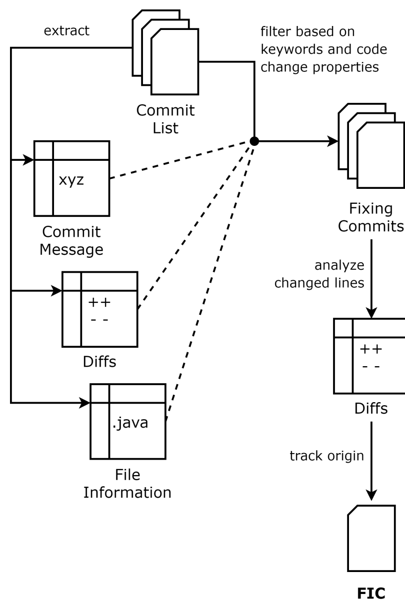


Figure 1: The methodology for identifying FICs from Commits

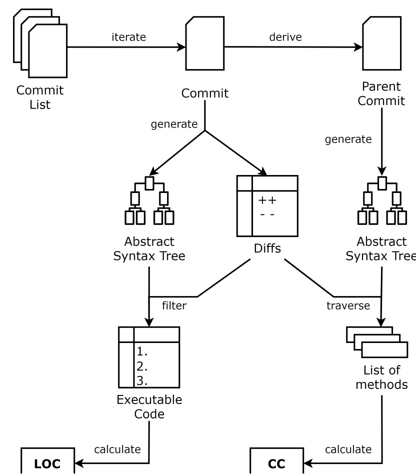


Figure 2: The methodology for extracting complexity metrics from Commits

bug. Hence, Commits with Insert edit types are discarded from further consideration.

5. After the filtering process, the remaining Commits are labeled as "Fixing Commits" or "Fixes". These are the Commits that removed buggy code.
6. Next, origin of each legitimate change in the Fix is tracked using the blame function, which returns the Commit where a specified changed line was last added or modified. These Commits are labeled as FICs.

3.2. Evolution Analysis

To understand the evolution of FICs, the projects' release tags are analyzed. Release tags define iterative final versions of the software in the project's lifetime. Since Commits are assigned these tags, it is possible to categorize Commits based on releases.

To analyze releases, first non-release tags are filtered out based on naming structures. Usually the release tags in most projects abide by the pattern: *"v #.#.#"*. The rest are tags depicting other information like branches or experiments. However, the structure of naming tags can vary with projects. For instance, patterns in projects like ElasticSearch or Commons-lang are *"Elasticsearch #.#.#"* and *"commons-lang-#.#.#"* respectively. Hence, tags are manually analyzed for each repository. Additionally, versions that are release candidates are discarded since these do not depict final releases.

Next, Commits are labeled based on their assigned tags. However, since Commits are automatically assigned all future tags, the labeling was conducted in two parts. First, Commits are extracted from all tags. Then, for every tag, only those Commits that were posted after the previous release tag are assigned to the current one.

As Commits are segmented into releases, analysis for Research Question (RQ) 1 is conducted. For each release, the number of FIC and non-FIC is calculated based on section 3.1. This segmentation is further elaborated for RQ2, by dividing each release into three equal parts. The three divisions are extracted to better understand the early, middle and late stages of a single release.

3.3. Metrics Extraction

To analyze the relation between FICs and complexity metrics — Line of Code (LoC) and Cyclomatic Complexity (CC) — first the changes to code are extracted. This includes the content of the changed files and numbers of lines which are modified or deleted. In the case of file contents, along with that of the current Commit, contents of its parent Commit are also extracted. Parent Commit is referred to the Commit directly prior to the current Commit. The contents of the parent Commit provides information of the state of code before the current Commit's changes. For FICs, their parents retains the properties of the code where the bug was introduced. With the contents of the current and parent Commits, the two metrics are calculated in the following manner:

1. **LoC**: To calculate the line of code, without considering comments, first the Abstract Syntax Tree (AST) [20] of a program is generated from the

changed files using JavaParser[1]. It is verified whether the changes have been conducted on executable code or not. Therefore, blank spaces are eliminated. Next, the modified lines in the current content are checked to identify whether these are comments based on the AST. If none of the changed lines are executable code, then the file is not further considered. Otherwise, the LoC of the parent content is calculated.

2. **CC**: To calculate cyclomatic complexity, the method in which change was introduced is first identified. This is done by taking each changed line of the current content and tracing its state back in the parent content. The generated AST of the parent content is traversed for each line. Each changed line of executable code is individually assessed and provided an associated method. This provides a list of changed methods for a single file. Each of their cyclomatic complexities are calculated, aggregating all possible paths (If-else, loops, switch statements).

4. Experimentation and Findings

Description of the dataset and the results observed for the 4 Research Questions (RQs) are described as follows.

4.1. Dataset

To conduct this research, eight well known Java projects are chosen from GitHub's repositories. These projects are open source and use GitHub as their primary medium of code storage and version control, enabling the extraction of all necessary Commits. Details of the repositories are displayed in Table 1. The projects comprise of a total of 142555 Commits to analyze. All eight repositories are used to analyze Research Questions (RQ) 1, 2 and 3. RQ4, which requires the source code, utilizes the first five repositories.

Table 1
Repository description of the eight projects

Project Name	Commits	Lifetime (Years)	Contri- -butors
Apache Tomcat	19360	8.5	21
Google Guava	4798	5	187
Mockito	5019	6.7	155
Commons-lang	5396	10	115
Apache Hadoop	21435	4.8	191
Selenium	23550	9.3	435
Elastic Search	44975	8.5	1216
Spring Framework	18022	6.4	369
Total	142555	59.1	2689

4.2. RQ1: Evolution of FICs

The 1st RQ aims to understand how FICs evolve, in terms of frequency and ratio, throughout the lifetime of software projects. The graphs in Figure 3 showcase the evolution of FICs in the eight software repositories analyzed. The different repositories show different types of patterns. In the majority of patterns, as seen in Figures 3(b, c, d, g, h) for projects Guava, Mockito, Commons-lang, Elasticsearch and Spring-framework respectively, FICs appear in the early stages of the projects' lifetime and decrease in newer versions. This indicates that earlier changes to the system tend to contain more instances of bug introduction. This can be due to a rapidly changing and volatile initial requirement, formative and incomplete development processes, lack of collaborative experience among the developers, or an insufficiency of reviewing and testing resources. But as the software evolves, the FICs get reduced, as an indication of bolstered testing and quality assurance processes, and project maturity.

On the other hand, projects Tomcat and Hadoop in Figures 3(a, e) show the opposite trend, where FICs are more predominant in later versions. This could happen due to a decreased level of scrutiny in reviewing efforts, a overhaul of new requirements, or other project and personnel related events. Only Figure 3(f) showcases a slightly more uniform pattern of FICs for the project Selenium. Although there are spikes of FICs occurring in specific versions, there is no apparent progression in the appearance of FICs.

Such visualizations of the evolution of FICs help in observing the history of the project in terms of buggy changes. This can be related to other aspects of projects that coincide with the decrease and increase of FICs to understand what affects the introduction of bugs from a high level perspective.

4.3. RQ2: FICs in Releases

In RQ2, the pattern of FICs within individual releases is observed. In Figure 4, the appearance of FICs within releases is displayed as black circles, where the size of the circle is determined by the proportion of FIC on the total number of Commits in that stage. The releases are divided into three stages: early, middle and late, and for some projects, versions are merged for visibility.

It can be seen that for almost all the projects, FICs are more predominant in early and middle stages of releases compared to late ones. The exceptions are Tomcat and Spring framework, where FICs are similarly or more prevailing in the late stages. The high level of appearance of FICs in early and middle stages of a release can be con-

[1]https://javaparser.org/

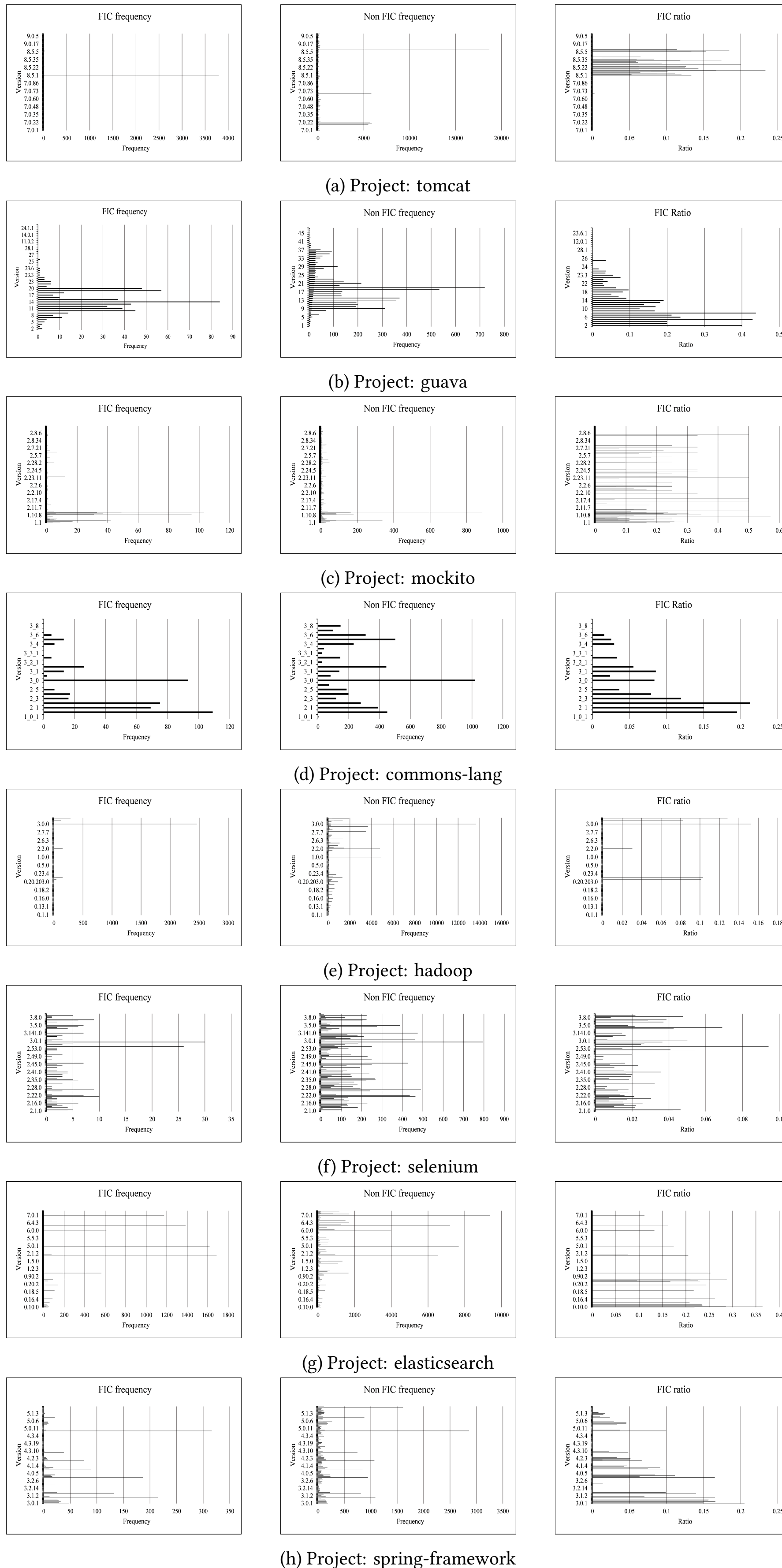


Figure 3: Evolution of FICs: FIC frequency, Non-FIC frequency and FIC ratio

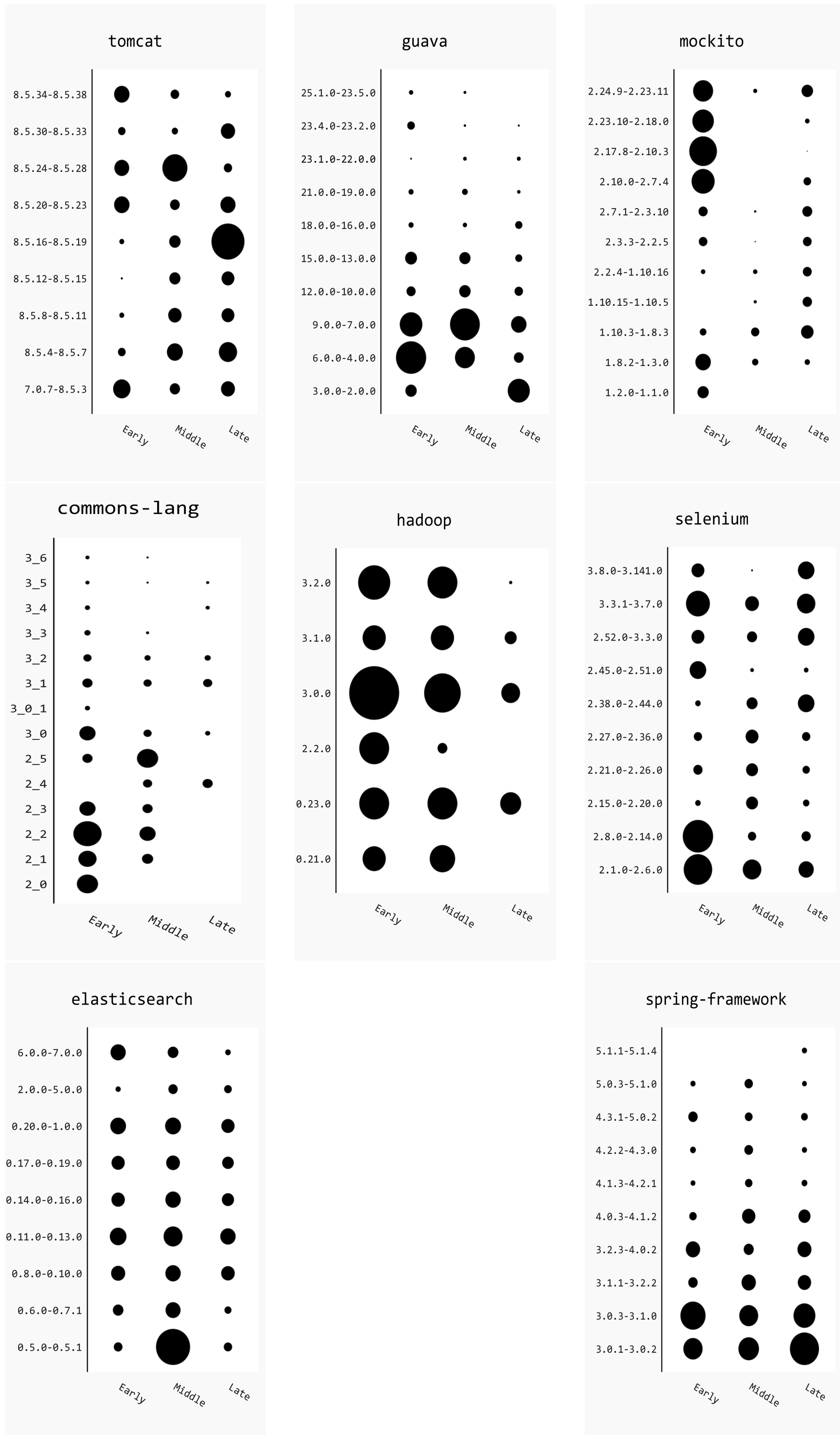


Figure 4: FICs in releases: early, middle and late stages

tributed to a higher emphasis on adding and updating features in those stages. The late stages are more focused on debugging and deployment efforts.

4.4. RQ3: FIC Interval

The 3rd RQ deals with the relation between Commit intervals and FICs. Table 2 shows that, on average, the interval (in minutes) for FICs is longer than that for regular Commits. The p-value of $< 2.2e^{-16}$ solidifies this difference as significant. This indicates that either a large amount of work relates to FICs (as shown by [5]), or that the developer introduces bug when they are away from the code for a long time.

The finding can help in preemptively detecting buggy code. Commits posted after a longer period than average can be given extra emphasis in reviews. Additionally, developers should be suggested not to disassociate from the code for a long time.

4.5. RQ4: FIC and Complexity Metrics

The last RQ observes whether FICs are related to the complexity metrics of software evolution: Line of Code (LoC) and Cyclomatic Complexity (CC). It can be seen in Table 2 that the average LoC of code where FICs occur is lower than that of regular Commits, with a p-value of $< 2.2e^{-16}$, making this difference significant. The result says that a lower LoC relates to buggy code. Hence the smaller, less busy components need to be given more emphasis in coding correctly and reviewing for bugs.

Next, as seen in Table 2, FICs occur in methods with significantly less CC than regular Commit, based on the $6.48e^{-10}$ p-value. A lower CC means that the tasks in methods are logically simpler. And yet, bugs, as statistically shown, tend to be introduced in such methods. Similar to LoC, this result prompts for a higher level of scrutiny when dealing with smaller and simpler methods.

Both of these findings support the evolution of FICs compared to complexity metrics. As described by Lehman's law of evolution[11], complexity rises as software evolves, hence increasing LoC and CC. Similarly, based on RQ1, FICs decrease in ratio in most cases, which is solidified by its inverse relation with the complexity metrics.

Table 2
Results for RQ3 and RQ4

Metric	Commit Type	Average	Standard Deviation	P-value
Interval	FIC	285.54	2027.59	$< 2.2e^{-16}$
	Regular	177.13	1113.37	
LoC	FIC	508.63	507.94	$< 2.2e^{-16}$
	Regular	636.6	760.61	
CC	FIC	3.49	3.64	$6.48e^{-10}$
	Regular	4.04	4.42	

5. Result Discussion

From the findings generated in this study, the following interpretations and applications can be estimated:

1. **Early bugs**: From both Research Questions (RQs) 1 and 2, it can be seen that bugs appear mostly in the early stages of versions and release cycles. This finding solidifies the intuition that early code tends to cause more bugs than later ones when maintenance starts to outrank development, and the software stabilizes. With this intuition proven through data, the finding can be applied to change the way software is developed. A more test-driven approach can be adopted in software projects from the beginning to mitigate the large influx of bugs.
2. **Comparative history**: By graphically extracting the evolution of FICs in software projects, the appearance of bugs can be historically analyzed. This history can unearth valuable insight, for example periods of time or certain releases where FICs peaked in number. These exceptions can be comparatively analyzed with other metrices related to the project. The metrices can range from code properties like components developed, design patterns used etc, or project aspects like type of assignment, assigned developer, developer turnover etc. The proponents may vary from project to project, hence the historical data of FICs can be used as a constant reference to such differing metrices.
3. **Intervals and bugs**: RQ3 provides insight into the correlation between Commit interval and FICs, showing the tendency of larger intervals causing bugs. This finding can be applied in project management, by monitoring the absences of developers. Developers who have been absent from the development process for longer periods should be assigned to tasks that are less sensitive and their work be reviewed more intensely. Furthermore, as also observed by Sliwerski et al. [5], large amount of changes in a single Commits that cause higher time for completion should be regulated for FICs.
4. **Software evolution and complexity**: The last finding demonstrates how FICs are correlated with line of code (LoC) and cyclomatic complexity (CC). These metrices, referred to as complexity metrices in the domain of software evolution, are important in understanding the evolution of FICs. As graphically shown in RQ1, FICs tend to decrease as the software evolves. On the other

hand, complexity increases with the software's age [11]. This indicates an inverted relationship between the two metrices, which is proven in RQ4 where FICs are found to be related with a larger LoC and CC in the code.

6. Conclusion

This study analyzes GitHub repositories to extract Fix-inducing Changes (FICs) — changes that introduce buggy code — and observes its evolution and characteristics. It is seen that FICs tend to occur in earlier versions and stages of releases. There is also a significant delay in posting FICs than regular Commits. Lastly, when relating with complexity metrics, FICs show up in code with less LoC and less CC than regular Commits. This corresponds with the decreasing FIC and increasing complexity of software evolution.

References

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

[2] M. Monperrus, Automatic software repair: a bibliography, ACM Computing Surveys (CSUR) 51 (2018) 1–24.

[3] N. Chauhan, Software Testing: Principles and Practices, Oxford University, 2010.

[4] T. Ackling, B. Alexander, I. Grunert, Evolving patches for software repair, in: Proceedings of the 13th annual conference on Genetic and evolutionary computation, 2011, pp. 1427–1434.

[5] J. Śliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes?, in: ACM sigsoft software engineering notes, volume 30, ACM, 2005, pp. 1–5.

[6] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, L. Bairavasundaram, How do fixes become bugs?, in: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM, 2011, pp. 26–36.

[7] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, O. Strollo, When does a refactoring induce bugs? an empirical study, in: 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, IEEE, 2012, pp. 104–113.

[8] A. Z. Sadiq, M. J. I. Mostafa, K. Sakib, On the evolutionary relationship between change coupling and fix-inducing changes (2019).

[9] S. F. Huq, A. Z. Sadiq, K. Sakib, Understanding the effect of developer sentiment on fix-inducing changes: An exploratory study on github pull requests, in: 2019 26th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2019, pp. 514–521.

[10] S. F. Huq, A. Z. Sadiq, K. Sakib, Is developer sentiment related to software bugs: An exploratory study on github commits, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2020, pp. 527–531.

[11] M. M. Lehman, Laws of software evolution revisited, in: European Workshop on Software Process Technology, Springer, 1996, pp. 108–124.

[12] C. F. Kemerer, S. Slaughter, An empirical approach to studying software evolution, IEEE transactions on software engineering 25 (1999) 493–509.

[13] G. Xie, J. Chen, 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.

[14] G. Antoniol, V. F. Rollo, G. Venturi, Detecting groups of co-changing files in cvs repositories, in: Eighth International Workshop on Principles of Software Evolution (IWPSE'05), IEEE, 2005, pp. 23–32.

[15] S. Levin, A. Yehudai, Boosting automatic commit classification into maintenance activities by utilizing source code changes, in: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, ACM, 2017, pp. 97–106.

[16] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE transactions on software engineering 33 (2006) 2–13.

[17] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, N. Ubayashi, An empirical study of just-in-time defect prediction using cross-project models, in: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, 2014, pp. 172–181.

[18] Y. Weicheng, S. Beijun, X. Ben, Mining github: Why commit stops–exploring the relationship between developer's commit pattern and file version evolution, in: 2013 20th Asia-Pacific Software Engineering Conference (APSEC), volume 2, IEEE, 2013, pp. 165–169.

[19] H. Osman, M. Lungu, O. Nierstrasz, Mining frequent bug-fix code changes, in: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), IEEE, 2014, pp. 343–347.

[20] I. Neamtiu, J. S. Foster, M. Hicks, Understanding source code evolution using abstract syntax tree matching, in: Proceedings of the 2005 international workshop on Mining software repositories, 2005, pp. 1–5.