Toward Metrics and Heuristics for Structuring Feature Toggles

Rezvan Mahdavi-Hezaveh
North Carolina State University
Raleigh, NC
rmahdav@ncsu.edu

Nirav Ajmeri
North Carolina State University
Raleigh, NC
najmeri@ncsu.edu

Laurie Williams
North Carolina State University
Raleigh, NC
laurie_williams@ncsu.edu

ABSTRACT

Using feature toggles is a technique to turn a feature either on or off in program code by checking the value of a variable in a conditional statement. This technique is increasingly used by software practitioners in software companies to support continuous integration and continuous delivery (CI/CD). However, using feature toggles may increase code complexity, create dead code, and decrease the quality of the code base. *The goal of this research is to aid software practitioners in structuring feature toggles in the code base by proposing a set of metrics and corresponding heuristics to guide feature toggle structuring based upon an empirical study of open source repositories.* We conducted a qualitative analysis of 66 open source repositories on GitHub which use feature toggles in their code base. From this analysis, we identify 12 metrics to measure the effect of incorporating feature toggles in the code. We propose 7 heuristics to guide structuring feature toggles such that they improve the proposed metrics. We conducted a case study of 22 GitHub repositories to evaluate the validity of the identified metrics and heuristics. We observe that larger repositories have self-descriptive feature toggles and guidelines. However, these repositories also have more duplicate code and dead code resulting from feature toggles, and a smaller number of test cases for feature toggles. This observation calls for the need of improved compliance measures. Our findings have important bearings on how feature toggles should be structured.

KEYWORDS

feature toggle, continuous integration, continuous development, open source repository, metric, heuristic

ACM Reference Format:
Rezvan Mahdavi-Hezaveh, Nirav Ajmeri, and Laurie Williams. 2020. Toward Metrics and Heuristics for Structuring Feature Toggles. In *Proceedings of The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Using feature toggles is a technique to either turn a feature on or off in program code by checking the value of a variable in a conditional statement. The concept of feature toggles is similar to configuration options. Configuration options are key-value pairs used by end users to include or exclude functionality in a software system. Despite the fact that feature toggles and configuration options are similar concepts, they have distinguishing characteristics and requirements [26], such as their users and lifetime. Whereas configuration options are used by end users and can exist permanently, feature toggles are used by developers and ideally will be removed from code.

Using feature toggles allows developers to integrate and test a new feature incrementally even if the feature is not ready to be deployed [35]. This technique is often used for continuous delivery (CD) and continuous integration (CI) in software development [31, 35]. Feature toggles can also be used by developers for other purposes, such as to perform experiments and to gradually roll out updates. Using feature toggles may impact the quality of the code base. For instance, adding a feature toggle adds more decision points to the code which would results in increased complexity. This increased complexity drives the need to remove feature toggles when their purpose is fulfilled.

Developers may follow certain styles when incorporating feature toggles in their code. As an example, the value of all feature toggles could be checked through an `isEnabled` method. As an alternative, each feature toggle could have a specific method to check the value of that toggle. For example, if the name of a toggle is `isEditable`, the method to check the value of that toggle could be `isEditableEnabled()`. Not following proper styles may have adverse effects on the code base. *The goal of this research is to aid software practitioners in structuring feature toggles in the code base by proposing a set of metrics and corresponding heuristics to guide feature toggle structuring based upon an empirical study of open source repositories.* Software practitioners prefer to learn through experiences of other software practitioners [28]. To address the goal, we systematically study feature toggle usage in open source software repositories, and (1) propose metrics, (FT-metrics), to measure the effect of feature toggle usage, and (2) develop heuristics to guide structuring of feature toggles based upon the proposed FT-metrics. Accordingly, we state the following research questions:

RQ$_M$ (metrics): What are the metrics to measure the effect of incorporating a feature toggle in the code base?

RQ$_H$ (heuristics): What are the heuristics to guide the structuring of feature toggles in a code base?

RQ$_E$ (evaluation): To what extent do software practitioners incorporate heuristics and how do these heuristics influence the identified feature toggle metrics?

To address the metrics research question (RQ$_M$), we analyze 66 GitHub repositories that use feature toggles. We analyze changes in the repository that are associated with feature toggle usage, and

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

identify a list of metrics. We also study existing design metrics including CK metrics [4] and select the metrics relevant to measuring the effect of feature toggle usage.

To address the heuristics research question (RQ_H), when we analyze the code base of the 66 GitHub repositories, we developed heuristics for using feature toggles in the code that affect the FT-metrics identified in RQ_M. We answer RQ_M and RQ_H iteratively.

To address the evaluation research question (RQ_E), we conduct a case study in which we analyze 22 additional GitHub repositories based on the identified FT-metrics and adoption of heuristics.

We summarize the contributions of this paper:

(1) Identification of 12 metrics, henceforth FT-metrics, to measure the effect of using feature toggles in a repository;
(2) Development of 7 heuristics to guide the structuring of feature toggles; and
(3) The results of a case study involving the analysis of the developed heuristics by developers in open source software, and its influence on FT-metrics.

Organization. In Section 2, we briefly describe the background of feature toggles and prior academic related works. In Section 3, we explain our research methodology. In Section 4, we report our identified FT-metrics. In Section 5, we explain the developed heuristics with examples. We report the result of the case study in Section 6. We discuss the limitations of our study in Section 7. We conclude and describe future work on feature toggles in Section 8.

2 BACKGROUND AND RELATED WORKS

We now provide background and discuss relevant related works.

2.1 Background

Releasing valuable software rapidly leads companies to use CI and CD to make development cycles shorter. CI is a practice of integrating and automatically building and testing software changes to the source repository after each commit [21]. CD is a practice for keeping the software in a state such that it can be released to a production environment at any time [10]. Using feature toggles is one of the techniques that are used by numerous software companies that practice CI and CD [31].

The language constructs to implement feature toggles have been included in programming languages for a long time. However, the first documented use of feature toggles to support CI/CD was at Flickr in 2009 [15]. Listing 1 is an example of a feature toggle usage. In this example, the value of the toggle `useNewAlgorithm` is checked and the use of the new search algorithm depends on the value of the toggle. If the value of this feature toggle is `True`, the new search algorithm is used, otherwise the search function calls the old search algorithm [23].

```
function Search() {
    var useNewAlgorithm = false;
    if(useNewAlgorithm) {
      return newSearchAlgorithm();
    }
    else {
      return oldSearchAlgorithm();
    }
}
```

Listing 1: An example of a feature toggle [23].

Five types of feature toggles are specified in [19, 35]:

- *Release toggles* are used to enable trunk-based development. In trunk-based development, all the developers commit changes to one shared branch instead of multiple branches. Using release toggles to block partially-completed features supports CI/CD in trunk-based development.
- *Experiment toggles* are used to perform experimentation on the software [18, 22], to evaluate new features changes and their effect on user behavior.
- *Ops toggles* are used to control the operational aspect of the system behavior. For instance, when a new feature is released, system operators can disable the feature quickly if it performs unexpectedly.
- *Permission toggles* or long-term business toggles [35] are used to provide the appropriate functionality to a user, such as special features to premium users.
- *Development toggles* are used for turning on or turning off certain features to debug and test code during development.

2.2 Related Works

Our work primarily relates to studies about feature toggles.

Rahman et al. [34] performed a qualitative study on online artifacts and conducted follow-up inquiries to study CD practices in 19 software companies. They reported 11 CD practices used in these companies. Using feature toggles is one of these practices which is used by 13 of the 19 companies. Parnin et al. [31] published 10 best practices from a discussion of researchers and practitioners from 10 companies. One of these best practices—*dark launching*—is the use of feature toggles to incrementally deploy code into production but keep it invisible from users.

Rahman et al. [35] performed a thematic analysis of videos and blog posts created by release engineers to understand the challenges, benefits, and cost of using feature toggles in practice. They also performed a quantitative analysis of feature toggle usage across 39 releases of Google Chrome over 5 years. Rahman et al. reported three objectives of using feature toggles in Chrome: rapid release, trunk-based development, and A/B testing. Whereas their study focused on quantitative analysis of Chrome's version history, our study focuses on qualitative analysis of open source repositories on a larger scale.

Mahdavi-Hezaveh et al. [23] conducted a qualitative analysis of grey literature artifacts and academic papers related to feature toggles and identified 17 feature toggle practices in 4 categories: Management practices (6 practices); Initialization practices (3 practices); Implementation practices (3 practices); and Clean-up practices (6 practices). They also analyzed feature toggle grey literature artifacts related to named companies and conducted a survey to find the frequency of usage of identified practices in these companies. Although some of Mahdavi-Hezaveh et al.'s identified practices such as "Use naming convention" and "Create a clean-up branch", can reflect in the source code of the repositories, they did not inspect the code base of repositories— the focus of our study.

Rahman et al. [36] extracted four architectural representations of Google Chrome: (1) conceptual architecture; (2) concrete architecture; (3) browser reference architecture; and (4) feature toggle architecture. Their study showed that using the extracted feature toggle architecture, developers can find out which module is affected by which feature, and vice versa. Whereas Rahman et al.'s study focused on the impact of using feature toggles on the modular architecture of the system, our study focuses on the effect of incorporating feature toggles in a system.

Other related research include works on configuration options. Sayagh et al. [41] interviewed 14 software engineering experts, surveyed Java software engineers, and conducted a literature review of the peer-reviewed papers in the area of configuration options. The goal of their study was to understand the process required by practitioners to use configuration options in the software system, the challenges they face, and the best practices that they could follow. One of the identified activities in the process of using configuration options is "Quality Assurance" which refers to improving software configuration *comprehensibility*, reducing software configuration's *complexity*, and improving its *maintainability*. We consider their categorization in our study of feature toggles.

Meinicke et al. [25] analyzed highly-configurable programs' traces to identify interactions among configuration options. These interactions can impact the quality assurance of the programs as configuration space can grow exponentially. This concern applies to feature toggles as well if they are not structured correctly.

Open source code repositories on GitHub are an excellent source of learning from software practitioners' experiences. These repositories have been analyzed for a variety of purposes including identifying quality outcomes of CI [45], recommending reviewers for pull-requests [46], and associating technical factors in open source contributions [44]. In our work, we analyze repositories on GitHub which incorporate feature toggles.

3 METHODOLOGY

This section provides an overview of our research methodology and the dataset we created and analyzed to develop FT-metrics and heuristics. Figure 1 outlines our two-phase research methodology we followed to analyze GitHub repositories.

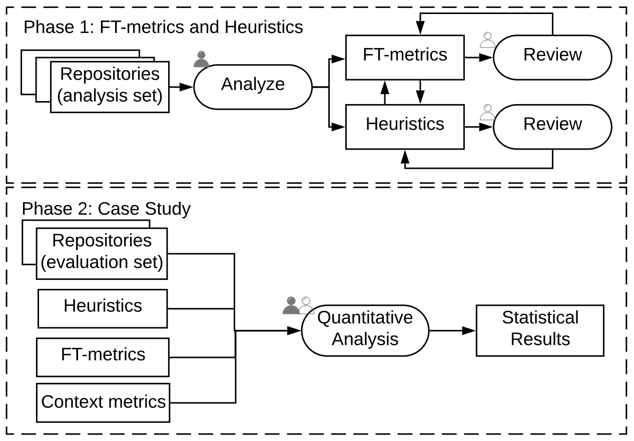

Figure 1: Research methodology outline.

3.1 Dataset–Repositories

First, we searched GitHub repositories for the keyword "feature toggle." The search date was 23-May-2019. GitHub categorizes search results in different categories. We checked the search results in the following result categories: (1) Repositories, (2) Code, (3) Commits, and (4) Issues. After checking the first 10 results in each category, we found that the results in the "Commits" category were the most appropriate for our study. Search results in "Repositories" and "Code" categories listed repositories of feature toggle management systems—studying which is not the focus of this work. Since use of GitHub "Issues" is less frequent than "Commits", repositories under "Issues" are less likely to include all repositories using feature toggles—low recall.

GitHub search returned 465,000 commits with "feature"and "toggle" in their commit messages. By default, GitHub sorts the search results by *best match*, which we found to be appropriate for our search criteria. Through inspection we found most relevant commit messages in the first 40 pages of the search results, each of which contained links to 10 commits. We manually investigated each of the 400 commits and identified 123 commits relevant to feature toggles. We excluded a commit if it met at least one of the following criteria.

(1) Commit URLs that did not exist anymore,
(2) Commits related to toggle/button in the UI of the application. For example, the commit message is "add toggle-all feature" and the change is "added checkbox to check all the options in UI,"
(3) Commits in repositories of feature toggle management systems which is not in the scope of this study,
(4) Commits from the repositories in which we cannot distinguish feature toggles from configuration options.

From 123 commits, we identified 88 repositories that use feature toggles in their development process. To find these repositories, we went through the commits, the files which were changed in the commit, and the changes which were made to the code. We also manually checked the repository to look into issues, pull requests, and documentations to find any details about incorporating feature toggles. We listed the characteristics of these repositories in Table1. [1] The five most popular languages in these repositories include TypeScript, Java, JavaScript, PHP, and C#. Other languages include C, C++, Go, Dart, Scala, Python, and Ruby.

Table 1: Characteristics of the identified repositories.

	Language	# repositories	LOC range
Top Five	TypeScript	19	7,840 to 68,583
	Java	15	227 to 361,213
	JavaScript	15	10,300 to 783,301
	PHP	8	1,567 to 400,034
	C#	7	18,232 to 148,390
Other languages		24	170 to 3,547,167
Total		88	170 to 3,547,167

[1]Data availability: The data we collected in our study is posted here: https://sites.google.com/view/feature-toggles-dataset/. We will turn it into archived open data in case of acceptance.

We split the set of 88 repositories in two (randomly selected and unequal) subsets: (1) *analysis set* consisting of 66 repositories which was analyzed to identify FT-metrics and to develop heuristics; and (2) *evaluation set* consisting 22 repositories to evaluate the identified FT-metrics and developed heuristics.

3.2 Phase 1: FT-metrics and Heuristics

The top block of Figure 1 outlines the steps in Phase 1. To address RQ_M and RQ_H, we manually analyzed the usage of feature toggles in each of the 66 repositories in the analysis set. We looked at individual commits where feature toggles are added, modified, or removed by the developers. In our manual analysis, we identified metrics affected by incorporating feature toggles and developed a list of heuristics. The manual analysis contained the following steps:

(1) We checked the changes made by developers in a commit, including the changed files and changed lines of code;
(2) We searched for the feature toggle configuration file (which contains a list of features) and the feature toggle class, if it existed;
(3) We searched for the usage of feature toggles identified in Step 2 of this process in the code;
(4) We recorded the details of incorporating feature toggles by the developers in the repository, including definition details of the toggles (such as file of definition, provided meta-data, names of them), usage details of the toggles (such as checking the value of the toggle), removing details of the toggles (such as removed lines of codes and changed files); and
(5) We inspected issues, pull requests, guidelines, and comments to find information about incorporating feature toggles.

During the whole process, we took notes about our observations. Based on the notes, we identified three affected aspects of the repository: complexity, comprehensibility and maintainability. For each aspect, we identified a list of metrics to measure effect of incorporating feature toggles and develop heuristics to structure feature toggles, which can potentially improve the identified metrics. We used Meneely et al.'s. [27] criteria to validate identified metrics.

The first author performed the whole process. The second author checked the definitions and measurements of the identified FT-metrics, checked the definition and examples of the developed heuristics, and gave feedback in all steps. We report the results of Phase 1 in Section 4 and Section 5.

3.3 Phase 2: Case Study

The bottom block of Figure 1 outlines the steps in Phase 2. To address RQ_E, in Phase 2, we analyzed the default branch of each of 22 randomly-selected GitHub repositories in the evaluation set.

First, using the related commit to the repository, we (1) found the list of feature toggles in the repository; (2) measured each of the identified FT-metrics; and (3) checked whether the repository follows each of the developed heuristics. Additionally, we gathered the following context metrics for each repository: (1) lines of code, (2) language, (3) number of contributors, and (4) number of feature toggles. We used *cloc* [2] to calculate lines of code and identify the language. By the GitHub profile of each repository, we identified the number of contributors for that repository. Number of feature toggles was identified via manual inspection of the repository code base.

[2]http://cloc.sourceforge.net/

Next, to evaluate how heuristics influence identified metrics, for each heuristic, we separated repositories in two groups based on whether they follow (F) or do not follow (NF) that heuristic. To test for statistical significance in the differences between the metric values yielded by the following (F) and not following (NF) groups of repositories, we applied two tailed t-test [20] and Fisher's exact test [9], and compute Hedges' g [13, 16] and Pearson's r [11] (details in Section 6). We report the observed relation between FT-metrics, heuristics, and context metrics in Section 6.

4 FT-METRICS AND ITS ANALYSIS

We now describe the FT-metrics we identify to measure the effect of using feature toggles in the repository. Following the steps in Section 3.2, we first manually analyze 66 repositories in the evaluation set which incorporate feature toggles in their development cycles. We study existing general metrics including CK metrics [4], LOC, dead code, and duplicate code to understand if these could measure the effect of incorporating feature toggles in open source repositories. In addition, we identify a number of metrics based on our observations in the repositories. From our analysis, we identified 12 metrics. Based on their effect on the code base, we categorize these metrics into three categories: *Complexity*, *Comprehensibility*, and *Maintainability*.

4.1 FT-metrics

Below, we list the identified metrics to measure the effect of using feature toggles in the repository. The type of each metric is mentioned after the name of the metric. These metrics are grouped into three categories:

Complexity measures the complexity of using feature toggles in the code. Two metrics to calculate complexity are as follows:

- *M1: Number of added paths in code (Numeric)* is computed using the McCabe's Cyclomatic Complexity [24]. McCabe's Cyclomatic Complexity counts the number of paths in code, which depends on the number of decision points. Incorporating feature toggles adds decision points to the code, so we can use this metric to compute the added complexity. We focus on the change in the code when developers use feature toggles in the code, so we measure the "change" of the Cyclomatic complexity of the code. For example, if adding a feature toggle adds one if statement in the code, we count "+1" for the Cyclomatic complexity of the code. [3] If the complexity of the code was "n" before using the feature toggle, it is "n+1" after using the feature toggle. We are not interested in the actual number of n, but the difference in complexity before and after the use of feature toggles is important to us. The lower the number of added paths in the code, the better.
- *M2: Number of feature toggle value checking methods (Numeric)* is measured based on the concept behind Weighted

[3]This approach is used in the implementation of tools to calculate cyclomatic complexity metric of the code. [4]

Methods per Class (WMC). WMC is one of the CK object oriented metrics [4] which counts the number of methods in a class. The common method between all of the feature toggle classes is the function to check the value of feature toggles. The feature toggle class could contain other methods but we do not consider them. Since developers can check the value of feature toggles in a class in different ways, this metric could help us understand the consequence of using each. To measure this metric, we count the number of feature toggle value checking methods in the feature toggle class and assume all the methods have equal complexities. The lower the number of value checking methods, the better.

Comprehensibility measures how easy it is for practitioners to understand the use of feature toggles in the repository. We compute comprehensibility using the following metrics:

- *M3: Presence of guidelines (Binary)* for using feature toggles is important. Developers should know the processes of adding, modifying, and removing a feature toggle. Absence of guidelines may cause problems such as creation of dead code after feature toggle removal. Thus, it is an important for a project to have guidelines. “yes” is a preferred value for this metric.
- *M4: Intention revealing names (Binary)* for variables and methods is a well-known practice in coding [5]. Each feature toggle's name and related methods should tell the reader what value the feature toggle holds and what task does the code wrapped by the toggle accomplishes. “yes” is a preferred value for this metric.
- *M5: Use of comments (Binary)* as human-readable notes that support the source code is an important coding practice [33]. Using comments in the source code helps developers to understand the purpose and behavior of the feature toggles. “yes” is a preferred value for this metric.
- *M6: Use of description (Binary)* for each feature toggle is helpful to clarify the purpose of using the toggle. The description could be added as an attribute to the feature toggle class for it to be available everywhere the toggle is accessible. Unlike comments which are not available everywhere, object attributes are available everywhere a practitioner wants. “yes” is a preferred value for this metric.

Maintainability measures how easy it is for practitioners to maintain feature toggles in their code. This category includes the following metrics:

- *M7: Number of files (Numeric)* which contain a feature toggle including configuration files, code files, and test cases files. The greater the number of files that need to change to support feature toggles, the greater is the probability to make a mistake. For instance, if the value of a toggle is set in more than one configuration file, when developers decide to change the value of the toggle, they need to change the value in all files. We count the number of files for each feature toggle and then average it for the repository based on the number of toggles. The lower the number of files, the better.
- *M8: Number of locations (Numeric)* where a feature toggle is defined and used. As an example, consider a feature toggle used in two files. In a configuration file, a toggle is used once to set the value of the toggle and in another file the same toggle is used twice in if-statements. In this case, the number of locations for this toggle is three. The lower the number of locations, the higher the maintainability of the code. We count the number of locations where each feature toggle is defined and used, and then average the count for each repository. The lower the number of locations, the better.
- *M9: Lines of code (Numeric)* which should be added or removed when a feature toggle needs to be added or removed from a repository. In general, the number of lines of code is a metric to measure maintainability in software systems [17]. In our definition, this metric measures the effort a developer should expend to make any change to the code related to a feature toggle. We count the lines of codes for defining and testing each feature toggle and then average it for all toggles in each repository. The lower the lines of code, the better.
- *M10: Presence of duplicate code (Binary)* is a previously-defined code smell [37]. Duplicate code is a problem of repeating the same block of the code in the repository. If a duplicated code contains a feature toggle, in case of updating or removing the toggle all the occurrences of the duplicate code need to be updated or removed. Forgetting to update or remove feature toggles from duplicate code could be dangerous. Not having duplicate code is better when measuring this metric. “no” is a preferred value for this metric.
- *M11: Presence of dead code (Binary)* is one of the drawbacks of using feature toggles in an incorrect way. Dead code is a part of the code which is not used in any execution path of the code. In 2012, developers in Knight Capital Group, an American global financial services firm, updated their algorithmic router which accidentally repurposed a feature toggle and activated functionality which was unused for 8 years. Knight Capital Group lost nearly 400 million dollars in 45 minutes, which caused the group to go bankrupt [1]. This is an example of why dead code should be avoided when using feature toggles. If developers decide to remove a feature toggle, the toggle should be removed from all parts of the code including configuration files, code, and test cases. Not having dead code is better when measuring this metric. “no” is a preferred value for this metric.
- *M12: Presence of test cases (Binary)* for feature toggles is a metric to measure whether or not feature toggles are tested similarly to other parts of the code. Feature toggles may remain in the code for a long time so they should be treated like any other part of the code and tested. “yes” is a preferred value for this metric. We consider two types of test cases: (1) test cases for checking the values of the feature toggles, (2) test cases for checking the behavior of the code based on value of the feature toggle, If the repository has any type of test cases for more than half of the feature toggles in the code base, we record “yes” for this metric.

4.2 Analyzing FT-metrics

Meneely et al. [27] extracted 47 criteria to validate software metrics. They proposed a 6 step process to choose appropriate metric validation criteria among the list of the 47 criteria. Following their process, all the 11 listed advantages of using a metric are mapped to some or all of our FT-metrics. So, we checked all the 47 criteria. As a result, the following criteria are followed by all metrics in FT-metrics:

Prior validity, Actionability, Appropriate Continuity, Appropriate Granularity (medium and coarse), Association, Construct validity, Constructiveness, Definition Validity, Empirical validity (checked after doing a case study), Internal consistency, Internal validity, Monotonicity, Metric reliability, Nonuniformity, Process or Product Relevance, Renaming Insensitivity (except for Intention revealing name), Repeatability, Unit validity. The rest of the criteria do not apply to FT-metrics or are not followed by all of the FT-metrics.

5 HEURISTICS

In this section, we describe the heuristics we derive from our analysis of repositories in Phase 1 of our methodology. For each heuristic, we discuss examples found in the repositories of following or not-following that heuristic. Name of each example starts with letter H (as Heuristic) followed by the number of the heuristic. It is followed by subscript FE if the example is following example, and NFE if it is a not-following example. The number at the end of subscript is a counter of examples for that heuristic. For instance, $H1_{FE1}$ means: First example for heuristic 1; a heuristic "following" example.

For each heuristic, we determine the FT-metrics which addressed by the heuristic via comparing the metrics in following and not-following examples of the heuristic. Table 2 shows the mapping of the in FT-metrics and the heuristic that may impact each metric.

Table 2: FT-metrics addressed by each heuristic.

Categories	Metrics	H1	H2	H3	H4	H5	H6	H7
Complexity	M1(Paths)				✓			
	M2(Methods)	✓						
Comprehensibility	M3(Guidelines)			✓				
	M4(Intention)		✓					
	M5(Comments)		✓					
	M6(Description)		✓					
Maintainability	M7(Files)	✓			✓	✓		✓
	M8(Locations)				✓	✓		
	M9(Code)	✓				✓		✓
	M10(Duplicate)					✓		
	M11(Dead)	✓		✓	✓			✓
	M12(Test cases)						✓	

5.1 Shared Method to Check Value

Heuristic 1

Using a *shared method* to check the value of all feature toggles can help reduce complexity and increase maintainability.

The value of a feature toggle is checked in a conditional statement of code. One approach to access the value of a toggle is to call a feature toggle value checking method from the feature toggle class, which is often named `isEnabled()`[23]. The lower the number of feature toggle value checking methods (M2), the lower is the code complexity. Having fewer feature toggle value checking methods (M2) also decreases (1) the number of files (M7) and the lines of code (M9) that need to be modified to implement and maintain a feature toggle; and (2) the probability of creating dead codes (M11) when deleting feature toggles because an associated feature toggle value checking method (M2) does not need to be deleted.

Metrics addressed: (1) Number of feature toggle value checking methods (M2), (2) Number of files (M7), (3) Lines of code (M9), (4) Presence of dead code (M11).

Developers follow different styles in their code to the check value of a feature toggle using methods. We list three examples below.

Following Example ($H1_{FE1}$): Listing 2 shows an example of using one shared feature toggle value checking method [29]. Here, the name of the toggle is passed to the method and the value of the toggle is checked in the list of feature toggles. When adding a feature toggle, the developer should add the toggle only to the configuration file or database. By doing so, the toggle can be used anywhere in the code. Adding the toggle to the configuration file minimizes the number of modified files and lines of code. For removal, the toggles needs to be deleted from the configuration file or the database and the part of the code where it is used. No modifications are needed to the value checking method.

```
export const isFeatureEnabled = (feature: Feature) =>
    (window as any).appSettings[feature] === 'on' || (
     window as any).appSettings[feature] === 'true';
```

Listing 2: One shared IsEnabled value checking method [29].

Not-Following Example ($H1_{NFE2}$): The code in Listing 3 shows an example of each feature toggle having its own function to check its value [42]. When developers want to define a new feature toggle, instead of adding a toggle and its value to the configuration file, they define a new customized `isEnabled()` function in the file contains all other `isEnabled()` functions. The number of files which should be changed is one, but the number of lines of codes that should be changed is larger compared to $H1_{FE1}$.

```
public bool IsAggregateOverCalculationsEnabled() {
        return true;
}
```

Listing 3: IsEnabled function for one toggle [42].

5.2 Self-Descriptive Feature Toggles

Heuristic 2

Adding a *descriptions* of feature toggles as a meta-attribute in the configuration file, using intention-revealing names for toggles, and including comments when using the toggles can improve comprehensibility.

Having self-descriptive code makes understanding the code easier and reduces its maintenance effort. Feature toggles are part of the code which may remain in the code for a while so they should be treated similar to other parts of the code.

Metrics addressed: (1) Intention revealing names (M4), (2) Use of comments (M5), (3) Use of description (M6).

We list below examples of having and not having self-descriptive feature toggles in code:

Following Example ($H2_{FE1}$): Listing 4 is an example of having self-descriptive feature toggles [43]. Each toggle in the configuration file of CFS-Frontend repository has an intention reveling name and description, which makes the purpose of the feature toggle clear.

```
"EnableCheckJobStatusForChooseAndRefresh": {
      "type": "bool",
      "metadata": {
        "description": "Enable checking calc job status
     prior to choosing and refreshing"
      },
      "defaultValue": true
    }
```

Listing 4: A feature toggle with description in a configuration file [43].

Following Example ($H2_{FE2}$): Adding comments is a way to make code understandable. Feature toggles are part of the code so including comments clarifies the usage of feature toggles. In wp-calypso [3], intention revealing names and comments are used to explain code related to feature toggles. Two of these example comments (pertaining to `autorenewal` toggle in the code) are: *"The toggle is only available for the plan subscription for now, and will be gradually rolled out to domains and G suite"* and *"remove this once the proper state has been introduced."* Including comments is an approach to improve code clarity. Comments, however, are not traceable and could be missed by developers.

Following Example ($H2_{FE3}$): In the configuration file of refocus [38], the feature toggles are grouped in two groups: long term toggles and short term toggles. The following is the description developers provided in the file as a comment: "Defining a toggle in either 'shortTermToggles' or 'longTermToggles' has no bearing on how the toggle behaves—it is purely a way for us keep track of our intention for a particular feature toggle. It should help us keep things from getting out of hand and keeping tons of dead unused code around." For adding short term toggles they comment "add a new toggle here if you expect it to just be a short-term thing, i.e. we'll use it to control rollout of a new feature, but once we are satisfied with the new feature, we'll pull it out and clean up after ourselves." In addition, this team uses intention revealing name for feature toggles, and the configuration file is well commented.

A practice suggested by practitioners [23] is to "Determine the type of the toggle" before adding it to the code. When developers specify if the feature toggle is a short-lived toggle or a long-lived toggle, they can plan to remove the toggle on time. Also, if they plan to limit the number of feature toggles in the code, they have a list of short-lived toggles which are potential toggles to remove from the code first.

Not-Following Example ($H2_{NFE4}$): An example of not having self-descriptive feature toggles[5] is using names which are not meaningful. In this repository, a toggle was named `FEATURE_TOGGLE_520`, which does not convey its purpose.

[5]This feature toggle is now removed from the code. The link to the removing commit is https://github.com/hmcts/div-case-orchestration-service/commit/d8145dafd34670f0b559fc0c5a6d56f61c7285c0

5.3 Guidelines for Managing Feature Toggles

Heuristic 3

Including *guidelines* for adding or removing feature toggles can help improve the comprehensibility and maintainability.

Management of feature toggles is a challenge for developers and project managers. It is important for a development team to know when and how should they add or remove a feature toggle. If feature toggles are added arbitrarily, a large number of feature toggles may end up in the code after a while. Hence, including guidelines for removing toggles is important. If the development team does not know when to remove a toggle or how to remove one correctly, it could result in dead code.

One way to manage adding and removing feature toggles is using issues and pull requests in GitHub repositories. Using of issues and pull requests is more important in case of removing feature toggles. It helps developers not to forget removing a feature toggle. In [23], one of the feature toggle clean-up practices includes using pull requests. The advantage of using issues and pull requests is that they can be followed on the code; the connection between issues, pull requests, and code is clear and traceable. However, development teams may use other project management systems, such as a Kanban board that is not visible in repositories, to manage adding or removing a toggles.

Metrics addressed: (1) Presence of guidelines (M3), (2) Dead code (M11).

The example $H2_{FE3}$ is a good example of providing guidelines for managing feature toggles. The followings are other examples of following and not following this heuristic:

Following Example ($H3_{FE1}$): In the grid repository [14], developers use pull requests to delete a feature toggle. We use "feature toggle in:title" as search string in the list of issues and pull requests of a repository to find the ones related to feature toggles.

Not-Following Example ($H3_{NFE2}$): In one of the analyzed repositories [2], the guideline for adding feature toggles is provided in the `README` file. The guideline clearly states to first create a feature branch, and then define a feature toggle in the branch for implementing new feature or any significant change. Although the developers have guidelines for adding feature toggles, they do not include guidelines for removing those. In repositories following the approach of adding a feature toggle for every new feature, a large number of feature toggles will exist in the code after a while. Also, how can a developer specify if a change is significant or not? Inability to make a decision about the importance level of any change may cause creating a new feature toggle for any change in the code. With a large number of feature toggles and the absence of removal guidelines, the possibility of creating dead code is high.

5.4 Use Feature Toggles Sparingly

Heuristic 4

Use/edit a feature toggle in *as few locations as possible* in the code to reduce complexity and improve maintainability.

Having more locations to edit makes using feature toggles harder for developers. More the number of files to update, more the effort developers should make. More the number of path in the code, more the complexity it has. During removal, having toggles at more locations in the code increases the possibility of creating dead code.

Metrics addressed: (1) Number of paths in code (M1), (2) Number of files (M7), (3) Number of locations (M8), (4) Dead code (M11).

We list examples of following and not-following this heuristic:

Following Example (H4$_{FE1}$): Feature toggles could be either checked directly in conditional if-statements or be used to set the value of a variable and then the new variable is checked or used in the rest of the code [35]. Listing 5 shows an example of using feature toggles to set value of another variable. The `canFork` variable is checked in the rest of the file instead of checking the three conditions in the example [32]. If the value of the feature toggle needs to be checked in more than one place in a file or it needs to be checked in combination with other variables, using the feature toggle value to set the value of a new variable and checking the new variable in the rest of the code helps in preventing duplicate code and makes removing and updating the toggle easier. In Listing 5, instead of removing or updating the toggle at all locations in the file, it is enough to remove or update the toggle in lines 3 and 4.

```
1 // Set the value of a variable using a feature toggle.
2 const canFork = props.selection.isSingleDocument() &&
3   props.me.feature_toggles &&
4   props.feature_toggles.includes("forking");
```

Listing 5: Use feature toggles to set value to a new variable.

Not-Following Example (H4$_{NFE2}$): One way to store the value of the feature toggle is using configuration files but in some repositories more than one configuration file exists for the same set of feature toggles. For instance, MTC repository [6] which we analyzed has 14 files for feature toggles. To remove or to edit a feature toggle, a developer must remove or edit the toggle in all of the 14 configuration files. Missing any of the edits or deletions could cause issues such as dead code.

As another example, solar repositories [40] has a `setting.ts` file which checks the platform and based on the platform, uses `setting.ts` file in each platform to set values for feature toggles. While the use of multiple files is understandable, it increases possibility of dead code.

5.5 Avoid Redundancy in Feature Toggles

Heuristic 5

Avoiding *duplicate code* when using feature toggles can improve maintainability.

We consider redundancy from two viewpoints:

- Duplicate code: Duplicate code is a code smell [37]. Feature toggles are as important as other parts of the code so having duplicate code when using feature toggles is also a bad smell. Feature toggles (1) add complexity to the code because additional conditional statements in the code, and (2) may create dead code, if they are not removed after their purposes are accomplished.
- Duplicity in feature toggle identification: Redundant feature toggles add to the number of feature toggles and create duplicate code. Having the number of feature toggles as less as possible could help developers in managing the added complexity.

Metrics addressed: (1) Duplicate code (M10), (2) Number of files (M7), (3) Number of locations (M8), (4) Lines of codes (M9).

The following is an example of a toggle related duplicate code:

Not-Following Example (H5$_{NFE1}$): In refocus repository [39], the code in Listing 6 is used in a file twice. It is duplicate code with usage of feature toggles. It can make removing the toggle harder as developers need to make changes at multiple places.

```
if (featureToggles.isFeatureEnabled('
     enableWorkerActivityLogs') && jobResultObj &&
     logObject) {
    mapJobResultsToLogObject(jobResultObj, logObject);

    if (featureToggles.isFeatureEnabled('
     enableQueueStatsActivityLog')) {
        queueTimeActivityLogs.update(jobResultObj.
     recordCount, jobResultObj.queueTime);
    }
activityLogUtil.printActivityLogString(logObject, 'worker
     ');
}
```

Listing 6: Duplicate code [39].

Not-Following Example (H5$_{NFE2}$): In a file of the OpenConext-manage repository [30], we observe that two different toggles are defined for similar usage. In the configuration file, a feature toggle is named `exclude_oidc_rp` with the default value as `False` and the other toggle is `show_oidc_rp` with the default value as `True`. These two toggles should be combined into one toggle. If the toggles are used in one subsystem, it is inappropriate to have two toggles with different names for the same purpose. If they are used in different subsystems, it may be better to have toggles with the same name (and possibly different prefixes) to make it clear that they are the same. Having more than one feature toggle for a concept may cause confusion for developers and make removing the feature toggles harder.

5.6 Test Cases for Feature Toggles

Heuristic 6

Including *test cases* for each feature toggle can improve maintainability.

Feature toggles should be treated as other parts of the code. The feature toggles may exist in the code for a long period of time so they should implemented with high quality. One technique to check the quality and correctness of the code is including test cases. Test cases of feature toggles should remain in the code if the development team decides to make the feature permanent.

Metrics addressed: (1) Presence of test cases (M12).

We provide one example of following this heuristic:

Following Example (H6$_{FE1}$): In an analysed repository, MTC [7], when developers decided to make a feature wrapped in a feature toggle permanent, they removed unit tests for the disabled toggle and kept the tests for the enabled toggle with changed names.

5.7 Complete Removal of a Feature Toggle

Heuristic 7

Ensuring complete *removal* of a feature toggle by removing it from source code files, configuration files, and test cases can improve maintainability.

Feature toggles should be removed when the purpose of using the toggles is accomplished. The code related to the feature toggle should be removed from all files in the source code, including configuration files and test files. Incomplete removal can cause problems such as creating dead code.

Metrics addressed: (1) Number of files, (2) Lines of code, (3) Dead code.

Followings are some examples of removing feature toggles:

Following Example (H7$_{FE1}$): The commit [8] is an example of complete removal of a feature toggle. The developers remove the toggle from the configuration files, code, and test cases.

6 PHASE 2: CASE STUDY

In this section, we explain Phase 2 of the methodology in Figure 1, as we discussed in Section 3.3. We propose following specific sub-research questions to investigate RQ$_E$ on evaluating FT-metrics and proposed heuristics.

SRQ$_{MH}$ (Metrics and heuristics:) What is the relation between adoption of heuristics and values of the FT-metrics?

SRQ$_{CS}$ (Code size): How does the size of the project relate to number of feature toggles, values of FT-metrics, and adoption of identified heuristics?

SRQ$_{CC}$ (Contributor count): How does the number of project contributors relate to the number of feature toggles, values of FT-metrics, and adoption of identified heuristics?

To address these proposed sub-research questions, we conduct a case study on 22 repositories selected as part of the evaluation set.

6.1 Repository Inspection

We inspect each repository in the evaluation set and identify the feature toggles in its code base. Next, for each feature toggle, we compute all FT-metrics, as described in Section 4.

For each repository, we also identify whether or not the repository follows the heuristics we propose in Section 5. We list below our criteria for marking whether a repository follows a heuristic.

H1 is marked as *followed* if the values of all feature toggles are checked using a shared value checking method. H1 is not applicable to repositories that check primitive values of the toggles in conditional statements.

H2 is marked as *followed* if the repository follows at least two of the following three FT-metrics: (1) intention revealing names, (2) use of comments, (3) and use of description.

H3 is marked as *followed* if the repository has guidelines including systematic documentation or suggestion of using pull requests to manage feature toggles.

H4 is marked as *followed* (or *not followed*) based on an expert's (first author's) subjective judgement that a feature toggle could or could not be used in fewer files or locations. For example, if a repository has more than one configuration file with a list of all feature toggles and their values, H4 is marked as *not followed.*

H5 is marked as *followed* for a repository, if it does not have redundant feature toggles and duplicate code containing use of a feature toggle. This heuristic is marked based on subjective judgement of an expert (first author). For example, if expert finds duplicate usage of the same feature toggle in a file which seems to be avoidable, H5 is marked as *not followed.*

H6 is marked as *followed* for a repository, if the majority (> 50%) of feature toggles have associated test cases.

H7 is marked as *followed* if there is no trace of feature toggles in a repository's code base for which there are associated commit message referring to toggle removal. This heuristic applies only to repositories that has commits about feature toggle removal.

We applied the following statistical tests in our analyses.

Two tailed *t*-test assuming unequal variances, at 5% significance level, to measure the difference between the means[20] for numeric metrics.

Fisher's exact test at 5% significance level, to measure the difference between the means [9] for binary metrics.

Hedges' *g* to measure the effect sizes (the amount of difference) because it is well suited for small sample sizes [13, 16].

Pearson's *r* to measure the strength and direction of correlation (linear relationship) between two variables [11].

6.2 FT-metrics and Heuristics (SRQ$_{MH}$)

We now address SRQ$_{MH}$ on the relation between the adoption of the seven heuristics and the values of the twelve metrics. Table 3 summarizes our results and reports the average (μ) for each metric. For binary metrics, the preferred value of the metric is converted to 1 to compute average. Table 4 shows the p-values of two tailed *t*-test for context metrics and numeric metrics, and Fisher's exact test for binary metrics. The first row for each metric in Table 3 corresponds to repositories which follow a heuristic (F) and the subsequent row corresponds to the repositories which do not follow that heuristic (NF).

H1 (shared methods): We observe that the repositories ($n = 11$) which use a *shared method to check values* for all feature toggles have higher number of contributors ($\mu = 87$) and feature toggles ($\mu = 32$) compared to those which do not use a shared method. These teams may have a greater understanding that maintaining feature toggles is challenging when the size of a repository grows in terms of contributors and new features.

NSA: Rephrased below to address Laurie's comments; text needs to be improved Another important observation is that repositories that follow H1 has significantly more number of feature toggles. But despite of more toggles, there is no significant difference in the number of files (M7), number of locations (M8), or lines of code (M9) that need to be updated for maintaining a toggle. This observation is evidence that H1 supports maintainability.

H2 (self-descriptive toggles): As in H1, we notice that the repositories ($n = 7$) with *self-descriptive feature toggles* have more contributors and more feature toggles compared to repositories without self-descriptive feature toggles. This observation is consistent with our observation that large repositories also opt for shared methods to check feature toggle values (H1). In addition to improving three comprehensibility FT-metrics (M4–M6), we observe that repositories following H2 also have significantly lower presence of duplicate code (M10) and dead code (M11) (Hedges' $g > 0.8$, indicating a large effect size) compared to the repositories which do not follow H2. RMH: based on Fisher's exact test, M4 and M5 are statistically significant in groups. (not surprising). p-value for having dead code is 0.0512 which is on the border of significance

H3 (guidelines): Similar to H1 and H2, repositories ($n = 6$) which include *guidelines for feature toggle* have a larger number of contributors and feature toggles. The heuristic has obvious relation with the metric of presence of guidelines (M3) ($p < 0.05$ in Fisher's exact test). We, however, observe that mere presence of guidelines on structuring feature toggles does not necessarily improve the occurrence of duplicate code (M10) and dead code (M11) resulting from feature toggle incorporation. This observation calls for the need of other methods to improve adoption of guidelines on structuring feature toggles.

H4 (sparing use): In contrast to H1, H2 and H3, the heuristic on *using feature toggles sparingly* (H4) is followed more in repositories ($n = 18$) with a lower number of contributors and a lower number of the feature toggles. Compared to the repositories which do not follow H4, the ones which follow H4 have smaller number of added paths in the code (M1), smaller number of files (M7), smaller number of locations (M8), fewer lines of codes (M9), and significantly lower presence of duplicate code (M10) ($p < 0.05$ in Fisher's exact test, Hedges' $g > 0.8$).

H5 (redundancy): We observe that repositories ($n = 17$) which follow *avoiding redundancy in feature toggle* heuristic (H5) on average having fewer contributors and a lower number of feature toggles compared to repositories which do not follow this heuristic. We note that following H5 does improve maintainability including presence of duplicate code (M10), which is significantly lower ($p < 0.05$ in Fisher's exact test, Hedges' $g > 0.08$) in repositories which follow H5 compared to those which do not follow H5.

H6 (test cases): The heuristic of *including test cases* is directly related to the metric of presence of test cases (M12). We notice that repositories ($n = 8$) with a lower number of feature toggles are more likely to follow this heuristic. This could be attributed to the fact that maintaining a large number of feature toggles is difficult. As new features are introduced frequently, the contributors are more likely to not include test cases unless mandated.

H7 (removal): We observe that repositories with a larger number of contributors and feature toggles are more likely to not follow the *removal* heuristic. We also observe that the repositories ($n = 7$) which follow H7 are more likely to use comments (M5) and descriptions (M6). These repositories have (significantly) less dead code (M11) and are more likely to have test cases (M12). RMH: Based on Fisher's exact test dead code has p-value less than 0.05 but test cases have big p-value. Test cases are not significant even in t-test

Table 3: Observations from the case study for repositories which follow and which do not follow heuristics. 'n': # of repositories out of 22 which follow a heuristic; M1–M12: FT-metrics; F:following, NF:not following; Values in the table are the average metric values. Bold is better.

			H1	H2	H3	H4	H5	H6	H7
Context	n (# repositories)	F	11	7	6	18	17	8	7
		NE	11	15	16	4	5	14	15
	# contributors	F	87	55	131	21	21	51	52
		NF	9	45	17	173	140	47	87
	# feature toggles	F	33.82	26.43	34.67	14.00	14.76	6.75	27.86
		NF	6.64	17.33	14.81	48.25	38.80	27.93	34.14
Numeric	M1 (Paths)	F	**1.04**	1.87	**0.97**	**0.88**	**0.87**	1.19	1.47
		NF	1.11	**0.71**	1.12	1.99	1.79	**1.01**	**0.40**
	M2 (Methods)	F	**1.09**	**1.20**	**1.00**	**1.10**	**1.11**	3.00	4.00
		NF	8.00	2.75	2.88	5.67	4.50	**1.67**	**1.00**
	M7 (Files)	F	3.94	3.71	5.55	**2.79**	**2.84**	3.87	**3.58**
		NF	**3.24**	**3.53**	**2.85**	7.18	6.15	**3.43**	5.48
	M8 (Locations)	F	5.22	4.81	7.65	**3.45**	**3.36**	5.77	**5.13**
		NF	**3.95**	**4.48**	**3.43**	9.71	8.77	**3.91**	6.28
	M9 (Code)	F	12.48	**7.32**	**5.79**	**9.44**	9.88	19.97	23.46
		NF	**7.04**	10.90	11.25	11.19	**9.35**	**3.93**	**2.35**
Binary	M3 (Guidelines)	F	**0.45**	0.14	**0.83**	**0.28**	0.24	**0.38**	0.29
		NF	0.09	**0.33**	0.06	0.25	**0.40**	0.21	**0.57**
	M4 (Intention)	F	0.82	**1.00**	**1.00**	**0.89**	**0.88**	**0.88**	0.86
		NF	**0.91**	0.80	0.81	0.75	0.80	0.86	**1.00**
	M5 (Comments)	F	**0.36**	**0.86**	0.17	**0.28**	**0.29**	0.25	**0.29**
		NF	0.18	0.00	**0.31**	0.25	0.20	**0.29**	0.00
	M6 (Description)	F	**0.27**	**0.57**	0.17	0.17	0.18	0.13	**0.43**
		NF	0.09	0.00	**0.19**	**0.25**	**0.20**	**0.21**	0.00
	M10 (Duplicate)	F	0.64	**0.86**	0.33	**0.83**	**0.88**	0.50	0.57
		NF	**0.73**	0.60	**0.81**	0.00	0.00	**0.79**	0.57
	M11 (Dead)	F	0.64	**1.00**	0.33	0.67	0.65	0.62	**0.86**
		NF	**0.73**	0.53	**0.81**	**0.75**	**0.80**	**0.71**	0.14
	M12 (Test cases)	F	**0.55**	0.29	**0.67**	0.33	0.29	**1.00**	**0.71**
		NF	0.18	**0.40**	0.25	**0.50**	**0.60**	0.00	0.29

6.3 Code Size (SRQ_{CS})

We observe that code size positively correlates with the average number of contributors ($r = 0.42$) and the number of feature toggles ($r = 0.52$) in a repository. As code size grows, maintainability worsens — the number of files (M7) ($r = 0.38$) and locations (M8) ($r = 0.38$) that needs change during feature toggle maintenance increases. The presence of deadcode (11) and duplicate code (M10) also increase with increase in code size.

6.4 Contributor Count (SRQ_{CC})

We observe that the number of contributors positively correlate with the number of feature toggles ($r = 0.74$) in a repository. As observed with code size, a larger number of contributor worsens maintainability of toggles including files (M7) and locations (M8) that need change and the presence of dead code (M11) and duplicate code (M10). When the contributor count increases, we observe that presence of guidelines (H3) also increases — a positive observation. However, we also note that the mere presence of guidelines does not reflect other improvements. This observation calls for the need of creative measures such as sanctions [12] to improve compliance.

Table 4: Observations from the case study for repositories which follow and which do not follow heuristics. M1–M12: FT-metrics; F:following, NF:not following; Values in the table are the p-values of two tailed *t*-test for context metrics and numeric metrics, and Fisher's exact test for binary metrics. Bold indicate significance at 5%.

		H1	H2	H3	H4	H5	H6	H7
Context	# contributors	0.09	0.82	0.19	0.26	0.26	0.93	0.63
	# feature toggles	0.20	0.72	0.51	0.46	0.51	0.21	0.85
Numeric	M1 (Paths)	0.88	**<0.01**	0.75	0.22	0.23	0.75	0.10
	M2 (Methods)	0.50	0.41	0.32	0.43	0.40	0.40	0.34
	M7 (Files)	0.63	0.91	0.17	0.13	0.17	0.79	0.37
	M8 (Locations)	0.47	0.87	0.09	0.08	0.06	0.38	0.66
	M9 (Code)	0.50	0.58	0.37	0.77	0.93	0.15	0.10
Binary	M3 (Guidelines)	0.15	0.62	**<0.01**	1.00	0.59	0.62	0.59
	M4 (Intention)	1.00	0.52	0.53	0.47	1.00	1.00	1.00
	M5 (Comments)	0.64	**<0.01**	0.63	1.00	1.00	1.00	0.46
	M6 (Description)	0.59	**<0.01**	1.00	1.00	1.00	1.00	0.19
	M10 (Duplicate)	1.00	0.35	**0.05**	**<0.01**	**<0.01**	0.34	1.00
	M11 (Dead)	1.00	**0.05**	**0.05**	1.00	1.00	1.00	**0.03**
	M12 (Test cases)	0.18	1.00	0.14	0.60	0.31	**<0.01**	0.29

7 THREATS TO VALIDITY

Internal validity. One, we searched GitHub for keyword "feature toggle" and checked only a subset of search results. It is possible we missed repositories which use feature toggles in their development process. Two, identifying FT-metrics and developing heuristics could be subjective to the first author's knowledge. To mitigate this threat, we asked second author to review the FT-metrics and heuristics and give feedback.

External validity. One, we use open source repositories from GitHub for our study. Including repositories from other organizations may change the results of our study. Two, to check the generalization of our result, we performed a case study on a set of repositories. If more repositories were analyzed in the case study, we would have a stronger evidence of generalization.

8 CONCLUSION AND FUTURE WORK

Using feature toggles is a widely used technique to support CI/CD pipeline in software development. We manually analyzed 66 repositories and 123 commits related to the use of feature toggles in open source repositories on GitHub. We identified (1) 12 metrics, which we refer as FT-metrics, to measure the effect of incorporating feature toggles, and (2) 7 heuristics to structure feature toggles. We evaluate our proposed metrics and developed heuristics via a case study on 22 repositories. Our result shows that in larger repositories which incorporate feature toggles, maintainability is harder, and presence of dead code and duplicate code is more prevalent. We observe that larger teams include guidelines on how to structure feature toggles, however, they are unable to enforce to these guidelines resulting in duplicate and dead code. We also notice that repositories with a lower number of feature toggles are more likely to have test cases for feature toggles.

An automated tool to compute FT-metrics and recommending heuristics for structuring feature toggles in the code base is a future direction. A larger scale evaluation effort to generalize our findings outside of open source repositories is another direction.

NSA: Fix feature-toggle.bib. Read here: https://www.acm.org/publications/authors/bibtex-formatting

REFERENCES

[1] [n. d.]. Knightmare: A DevOps Cautionary Tale. [Online]. Available: https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/ Accessed 24 April 2019. ([n. d.]).

[2] AusDVA. [n. d.]. myservice-prototype. [Online]. Available: https://github.com/AusDVA/myservice-prototype Accessed 9 October 2019. ([n. d.]).

[3] Automattic. [n. d.]. wp-calypso. [Online]. Available: https://github.com/Automattic/wp-calypso/blob/8d1bf0b0146fc341288059c765e8a3bf8c8bb7ef/client/me/purchases/manage-purchase/purchase-meta.jsx Accessed 21 October 2019. ([n. d.]).

[4] Shyam R Chidamber and Chris F Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering (TSE)* 20, 6 (1994), 476–493.

[5] Nell B. Dale, Chip Weems, and Mark R. Headington. 2003. *Introduction to Java and Software Design: Swing Update.* Jones & Bartlett Learning.

[6] DFEAGILEDEVOPS. [n. d.]. MTC. [Online]. Available: https://github.com/DFEAGILEDEVOPS/MTC/tree/0eb2d765b6683c90c852ba21c225742f07f050b9/admin/config Accessed 9 October 2019. ([n. d.]).

[7] DFEAGILEDEVOPS. [n. d.]. MTC. [Online]. Available: https://github.com/DFEAGILEDEVOPS/MTC/commit/0eb2d765b6683c90c852ba21c225742f07f050b9#diff-8633026cf78840f2cb5a5b32fe1aa00f Accessed 11 October 2019. ([n. d.]).

[8] DFEAGILEDEVOPS. [n. d.]. MTC. [Online]. Available: https://github.com/DFEAGILEDEVOPS/MTC/commit/0eb2d765b6683c90c852ba21c225742f07f050b9 Accessed 27 February 2020. ([n. d.]).

[9] Ronald Aylmer Fisher. 1992. Statistical methods for research workers. In *Breakthroughs in statistics.* Springer, 66–70.

[10] Martin Fowler. 2013. Continuous Delivery. [Online]. Available: https://martinfowler.com/bliki/ContinuousDelivery.html, Accessed 6 August 2019,. (2013).

[11] David Freedman, Robert Pisani, and Roger Purves. 2007. *Statistics (4th Edition.* W. W. Norton & Company, New York.

[12] Shubham Goyal, Nirav Ajmeri, and Munindar P. Singh. 2019. Applying Norms and Sanctions to Promote Cybersecurity Hygiene. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems.* International Foundation for Autonomous Agents and Multiagent Systems, Montreal, 1991–1993.

[13] Robert J. Grissom and John J. Kim. 2012. *Effect Sizes for Research: Univariate and Multivariate Applications.* Routledge, Abingdon-on-Thames.

[14] guardian. [n. d.]. grid. [Online]. Available: https://github.com/guardian/grid Accessed 27 February 2020. ([n. d.]).

[15] Ross Harmes. [n. d.]. Flipping Out. [Online]. Available: http://code.flickr.net/2009/12/02/flipping-out/, Accessed 6 August 2019. ([n. d.]).

[16] Larry V. Hedges and Ingram Olkin. 2014. *Statistical Methods for Meta-Analysis*. Academic Press, Inc., Orlando.

[17] Ilja Heitlager, Tobias Kuipers, and Joost Visser. 2007. A practical model for measuring maintainability. In *Proceedings of the 6th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 30–39.

[18] Buck Hodges. [n. d.]. Progressive Experimentation with Feature Flags. [Online]. Available: https://docs.microsoft.com/en-us/azure/devops/learn/devops-at-microsoft/progressive-experimentation-feature-flags, Accessed 6 August 2019. ([n. d.]).

[19] Pete Hodgson. [n. d.]. Feature Toggles (aka Feature Flags). [Online]. Available: https://martinfowler.com/articles/feature-toggles.html, Accessed 6 August 2019. ([n. d.]).

[20] Myles Hollander and Douglas A. Wolfe. 1999. *Nonparametric Statistical Methods*. Wiley, New York.

[21] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, Boston.

[22] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M Henne. 2009. Controlled Experiments on the Web: Survey and Practical Guide. *Data mining and knowledge discovery* 18, 1 (2009), 140–181.

[23] Rezvan Mahdavi-Hezaveh, Jacob Dremann, and Laurie Williams. 2019. Feature Toggle Driven Development: Practices used by Practitioners. *arXiv preprint arXiv:1907.06157* (July 2019), 1–25.

[24] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering (TSE)* 2, 4 (1976), 308–320.

[25] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-configurable Systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 483–494.

[26] Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, and Christian Kaestner. 2020. Exploring Differences and Commonalities between Feature Flags and Configuration Options. In *Proceedings of the 42nd International Conference on Software Engineering - Software Engineering in Practice (ICSE-SEIP)*. To appear.

[27] Andrew Meneely, Ben Smith, and Laurie Williams. 2013. Validating Software metrics: A Spectrum of Philosophies. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21, 4 (2013), 1–28.

[28] Geoffrey A Moore. 2009. *Crossing the Chasm: Marketing and Selling Technology Project*. Harper Collins, New York.

[29] navikt. [n. d.]. pleiepengesoknad. [Online]. Available: https://github.com/FeatureToggleStudy/pleiepengesoknad/blob/master/src/app/utils/featureToggleUtils.ts#L7 Accessed 27 February 2020. ([n. d.]).

[30] OpenConext. [n. d.]. OpenConext-manage. [Online]. Available: https://github.com/OpenConext/OpenConext-manage/blob/94262d9ef84761097cc5838bea88fc393a90b76c/manage-server/src/main/resources/application.yml Accessed 9 October 2019. ([n. d.]).

[31] Chris Parnin, Eric Helms, Chris Atlee, Harley Boughton, Mark Ghattas, Andy Glover, James Holman, John Micco, Brendan Murphy, Tony Savor, et al. 2017. The Top 10 Adages in Continuous Deployment. *IEEE Software* 34, 3 (2017), 86–95.

[32] pelagios. [n. d.]. recogito2-workspace-frontend. [Online]. Available: https://github.com/pelagios/recogito2-workspace-frontend/blob/367723732cfa74c4f38e542b88c0a4491789cc04/src/profile/Profile.jsx Accessed 9 October 2019. ([n. d.]).

[33] Grubb Penny et al. 2003. *Software Maintenance: Concepts and Practice*. World Scientific.

[34] Akond Ashfaque Ur Rahman, Eric Helms, Laurie Williams, and Chris Parnin. 2015. Synthesizing continuous deployment practices used in software development. In *Proceedings of the IEEE Agile Conference*. IEEE, 1–10.

[35] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. 2016. Feature Toggles: Practitioner Practices and a Case Study. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*. ACM, Austin, 201–211.

[36] Md Tajmilur Rahman, Peter C Rigby, and Emad Shihab. 2018. The Modular and Feature Toggle Architectures of Google Chrome. *Empirical Software Engineering (EMSE)* 22, 2 (2018), 1–28.

[37] Chanchal Kumar Roy and James R. Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

[38] salesforce. [n. d.]. refocus. [Online]. Available: https://github.com/salesforce/refocus/blob/18116cb7df73c0db70af3c6115342ccf93db6534/config/toggles.js Accessed 21 October 2019. ([n. d.]).

[39] salesforce. [n. d.]. refocus. [Online]. Available: https://github.com/salesforce/refocus/blob/18116cb7df73c0db70af3c6115342ccf93db6534/jobQueue/jobWrapper.js Accessed 9 October 2019. ([n. d.]).

[40] satoshipay. [n. d.]. solar. [Online]. Available: https://github.com/satoshipay/solar/blob/122830c16ba1fcbf82f4b388f92ea0483444c938/src/platform/settings.ts Accessed 9 October 2019. ([n. d.]).

[41] Mohammed Sayagh, Noureddine Kerzazi, Bram Adams, and Fabio Petrillo. 2018. Software configuration engineering in practice: Interviews, survey, and systematic literature review. *IEEE Transactions on Software Engineering (TSE)* (2018).

[42] SkillsFundingAgency. [n. d.]. CFS-Backend. [Online]. Available: https://github.com/SkillsFundingAgency/CFS-Backend/blob/d4461bda36e3d785909350233f833594984823c3/Debugging/CalculateFunding.DebugAllocationModel/FeatureToggles.cs Accessed 21 October 2019. ([n. d.]).

[43] SkillsFundingAgency. [n. d.]. CFS-Frontend. [Online]. Available: https://github.com/featuretogglestudy/CFS-Frontend/blob/39961217b8aabd665c71b108903d87014a41582c/DevOps/frontend-azure.dfe.json#L237 Accessed 27 February 2020. ([n. d.]).

[44] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. 356–366.

[45] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE)*. 805–816.

[46] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 2016. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Information and Software Technology (IST)* 74 (2016), 204–218.