Toward Heuristics and Metrics for Structuring Feature Toggles

Anonymous authors

***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 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 a code base. *The goal of this research is to aid software practitioners in structuring feature toggles in the code base by proposing and evaluating a set of heuristics and corresponding complexity, comprehensibility, and maintainability metrics to reduce technical debt resulting from feature toggle usage based upon an empirical study of open source repositories.* We identified 81 GitHub repositories that use feature toggle in their development cycle: analysis set (59 repositories) and evaluation set (22 repositories). We conducted a qualitative analysis of 59 repositories in analysis set. From this analysis, we proposed 7 heuristics to guide structuring feature toggles and identified 12 metrics to support the principles embodied in the heuristics. We conducted a survey of practitioners from all 81 repositories in our dataset. We used a five agreement options based on Likert scale. The median value of agreement level for each heuristic is more than 4 which shows that survey respondents agree with the usefulness of proposed heuristics and relation between heuristics and supporting metrics. We also conducted a case study of 22 repositories in evaluation set to evaluate the identified heuristics and metrics. We observe that larger repositories have self-descriptive feature toggles and guidelines (more comprehensibility). 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 (less maintainability). This observation calls for the need of improved compliance measures. Our findings have important bearings on how feature toggles should be structured.**

***Index Terms*—feature toggle, continuous integration, continuous development, open source repository, heuristic, metric**

I. 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. Using feature toggles allows developers to integrate and test a new feature incrementally even if the feature is not ready to be deployed [1]. This technique is often used for continuous delivery (CD) and continuous integration (CI) in software development [1], [2]. Feature toggles can also be used by developers for other purposes, such as to perform experiments or 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 results in increased complexity. This increased complexity drives the need to remove feature toggles when their purpose is fulfilled. Feature toggles structured incorrectly could result in technical debt [**?**]. Thus, arises a need for guidelines on how to structure and manage feature toggles.

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 one `isEnabled` method. As an alternative, each feature toggle could have its own 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()`. Some feature toggle style choices may have adverse effects on the code base and may lead to technical debt [**?**]. *The goal of this research is to aid software practitioners in structuring feature toggles in the code base by proposing and evaluating a set of heuristics and corresponding complexity, comprehensibility, and maintainability metrics to reduce technical debt resulting from feature toggle usage based upon an empirical study of open source repositories.* Software practitioners prefer to learn through experiences of other practitioners [3]. To address the goal, we systematically study feature toggle usage in open source software repositories, and (1) develop heuristics, (FT-heuristics), to guide structuring of feature toggles, and (2) propose metrics, (FT-metrics), to support the heuristics. Accordingly, we state the following research questions:

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

RQ_M (metrics): What are the metrics to support the effect of incorporating proposed heuristics in the code base?

RQ_E (evaluation): To what extent do the practitioners incorporate the heuristics, and how do the metrics support those heuristics?

We answer RQ_M and RQ_H iteratively. To address the heuristics research question (RQ_H), we analyze the code base of the 59 GitHub repositories and develop heuristics for incorporating feature toggles in the code.

To address the metrics research question (RQ_M), we analyze 59 GitHub repositories that use feature toggles to identify metrics which support heuristics. Specifically, we analyze files in the repository that are associated with feature toggle usage, and study existing design metrics including CK metrics [4] and select the metrics that can be related to feature toggles.

To address the evaluation research question (RQ_E), we conduct a survey of practitioners in GitHub repositories in our dataset to measure the agreement with FT-heuristics and FT-metrics. We also conduct a case study in which we analyze 22 additional GitHub repositories based on the adoption of FT-heuristics and FT-metrics.

We summarize the contributions of this paper:

(1) Development of 7 heuristics, henceforth FT-heuristics, to guide the structuring of feature toggles;
(2) Identification of 12 metrics, henceforth FT-metrics, to support the principles embodied in the heuristics in a repository;
(3) The results of a case study involving the analysis of the developed heuristics by developers in open source software, and their supporting FT-metrics; and
(4) A database of GitHub repositories that use feature toggles in their development cycle.

Organization: Section II describes the background of feature toggles and prior related works. Section III explains our research method. Section IV details FT-metrics. Section V explains the FT-heuristics with examples. Section VI presents the result of our evaluation via a developer survey and a case study. Section VII discusses the limitations of our study. Section VIII concludes with discussion on future directions.

II. Background and Related Works

We provide background and discuss relevant related works.

A. 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 [5]. CD is a practice for keeping the software in a state such that it can be released to a production environment at any time [6]. Using feature toggles is one of the techniques that are used by numerous software companies that practice CI and CD [2].

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 [7]. Listing 1 is an example of a feature toggle usage where the value of the toggle `useNewAlgorithm` is checked to determine which search algorithm to call. If the value of the toggle `useNewAlgorithm` is `True`, search function calls the new search algorithm, otherwise calls the old search algorithm [8].

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

Listing 1. An example of a feature toggle [8].

Researchers and practitioners have classified feature toggles in five types [9], [10], [11], [1]: (1) *release toggle* to enable trunk-based development, (2) *experiment toggle* to evaluate new features, (3) *ops toggle* to control operational aspects, (4) *permission toggle* to provide the appropriate functionality to a user, such as special features to premium users, and (5) *development toggle* to turn on or off developmental features. In our analysis, we do not differentiate between these types.

B. Related Work

Rahman et al. [12] performed a qualitative study on online artifacts and conducted follow-up inquiries to study CD practices in 19 software companies and reported 11 CD practices. Using feature toggles is one of the 11 CD practices used by 13 of the 19 companies. Parnin et al. [2] published 10 best practices from a discussion of researchers and practitioners from 10 companies. One of these best practices—*dark launching*—is enabled by feature toggles to incrementally deploy code into production but keep the new code invisible from users.

Rahman et al. [1] 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 quantitatively analyzed feature toggle usage across 39 releases of Google Chrome over 5 years. Rahman et al. reported three objectives of using feature toggles: rapid release, trunk-based development, and A/B testing. Whereas their study limited to analysis of Chrome's version history, our study focuses on qualitative analysis of structuring feature toggles in open source repositories on a larger scale.

Mahdavi-Hezaveh et al. [8] quantitatively analyzed grey literature artifacts and academic papers related to feature toggles and identified 17 feature toggle practices in 4 categories. Although practices such as "Use naming convention" and "Create a clean-up branch", identified by Mahdavi-Hezaveh et al., can reflect in the source code, they did not inspect the code base of repositories which we focus in this paper.

Ramanathan et al. [13] developed a refactoring tool to delete old and unused feature toggles in the code. The tool takes as input the name of a toggle, analyzes the abstract syntax tree of the code, generates a diff on GitHub repository, and assigns it to the author of the feature toggle. Meinicke et al. [14] proposed a semi-automated approach to detect feature toggles in open-source repositories, based on analyzing the repositories' commit messages. They found 100 open-source repositories that use feature toggles with keyword search in commits. They analyzed some aspects of feature toggle usage in these identified repositories, such as the relationship of having short-lived feature toggles and having a toggle owner. These automation tools and approaches can be improved considering our proposed heuristics and identified metrics.

Other related research include works on configuration options. 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 [15]. Despite the fact that feature toggles and configuration options are similar concepts, they have distinguishing characteristics and requirements, 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 are eventually removed from code. However, in reality, a large fraction of feature toggles stay in the code base forever [14]. However, the focus of this paper is not to compare our findings with similar findings related to configuration options. Our focus is to improve structuring of feature toggles.

Sayagh et al. [16] interviewed 14 software engineering experts, surveyed Java software engineers, and conducted a literature review of the peer-reviewed papers on configuration options to understand the process required by practitioners to use configuration options in software systems, 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 this categorization in FT-metrics.

Meinicke et al. [17] analyzed highly-configurable programs' traces to identify interactions among configuration options. These interactions impact the quality assurance of the programs as configuration space can grow exponentially. If structured incorrectly, this concern applies to feature toggles too. Whereas Meinicke et al.'s focus was interactions in configuration options, our focus is on structuring feature toggles.

III. Methodology

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

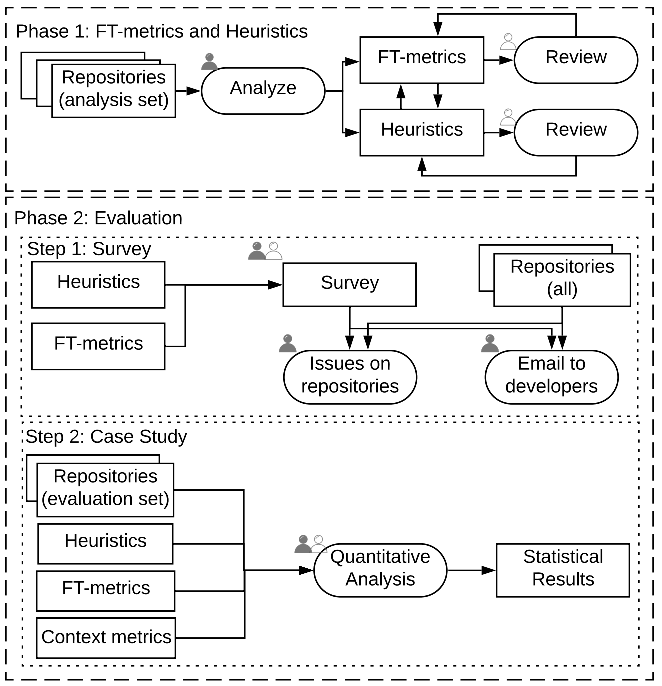

Fig. 1. Research methodology outline.

A. Dataset–Repositories

We analyze GitHub repositories which incorporate feature toggles. First, we searched GitHub repositories for the keyword "feature toggle". Meinicke et al. [14] also used keyword search and identified 100 GitHub repositories that used feature toggles, but the paper was not published at the time of starting our study. Our search date was 23-May-2019. GitHub categorizes search results in different categories. We inspected the search results in the following categories: (1) Repositories, (2) Code, (3) Commits, and (4) Issues. After inspecting 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 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.

GitHub search returned approximately 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 the first 400 search result are most likely relevant to feature toggles. Note that GitHub search skips forks by default, so the commits of forked repositories are excluded automatically. Including forks could skew the results. We manually investigated each of the 400 commits. We examined the commits, the files which were changed in the commit, and the changes which were made to the code. We also manually inspected the issues, pull requests, and documentation in each repository to find details about incorporating feature toggles.

We identified 123 relevant commits after excluding commits which met at least one of the following exclusion criteria: (1) Commit URLs that do not exist anymore; (2) Commits related to toggle/button in the user interface (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; and (4) Commits from the repositories in which we cannot distinguish feature toggles from configuration options. For instance, if a toggle is defined in a way that end-users can change the value of it, it is possibly a configuration option so we exclude the repository.

The selected 123 commits belonged to 81 repositories that use feature toggles in their development process. We list the characteristics of these repositories in Table I.[1]

TABLE I
Characteristics of the identified repositories.

	Language	# repositories	LOC range
Top Five	TypeScript	18	7,840 to 68,583
	Java	15	227 to 361,213
	JavaScript	13	10,300 to 783,301
	PHP	8	1,567 to 400,034
	C#	7	18,232 to 148,390
	Other languages	20	170 to 3,547,167
	Total	81	170 to 3,547,167

[1]Data availability: The data including commits, repositories, and case study details is posted here: https://sites.google.com/view/feature-toggles-dataset/. We will turn this data into archived open data in case of acceptance.

We split the set of 81 repositories in two (randomly selected and unequal) subsets: (1) *analysis set* consisting of 59 repositories which was analyzed to develop FT-heuristics and identify FT-metrics; and (2) *evaluation set* consisting 22 repositories to evaluate the FT-heuristics and FT-metrics.

B. Phase 1: FT-heuristics and FT-metrics

The top block of Figure 1 outlines the steps in Phase 1. To address RQ_H and RQ_M, we manually analyzed the incorporation of feature toggles in each of the 59 repositories in the analysis set. In our manual analysis, we developed a list of heuristics and identified metrics to support heuristics iteratively. The manual analysis for each repository contained the following steps:

(1) Starting with individual commit we used to identify the repository, including the commit message, changed files, and changed lines of code in the commit, we identified the feature toggle configuration file (which contains a list of features) and the feature toggle class. If the configuration file was not found in changed files, we searched the repository for the feature toggle that existed in the commit, and traced it to find the feature toggle configuration file.
(2) We searched for the usage of feature toggles identified in Step 1 of this process in the code and commit history;
(3) We inspected issues, pull requests, guidelines, and comments to find information about incorporating toggles.
(4) We recorded the details of styles of incorporating feature toggles by the developers observed in Steps 2 and 3 of this process, including toggle definition details (such as file of definition, provided meta-data, names of them), toggle usage details (such as checking the value of the toggle), toggle removal details (such as removed lines of codes and changed files).

We develop heuristics to structure feature toggles using notes of details of incorporating styles of feature toggles.In addition, based on the notes, developed heuristics and considering relevant literature on metrics such as [4], we identified metrics to support heuristics and categorized them based on their effect in three categories: complexity, comprehensibility and maintainability. The heuristics embody the actionable recommendations based on current state of using feature toggles in analyzed repositories. Metrics are for supporting the heuristics.

The first author performed Phase 1 in consultation with the second author. The second author critically examined the definition and examples of the developed heuristics, examined the definitions and measurements of the identified FT-metrics, and gave feedback in all steps. We report the results of Phase 1 in Sections IV and V.

C. Phase 2: Evaluation

The bottom block of Figure 1 outlines the steps in Phase 2. To evaluate the acceptance of our proposed heuristics by developers, we conducted a survey. In the survey, we asked for the level of agreements of with each heuristic and on each supporting metric. We used Likert scale [18] to specify the level of agreement: Strongly disagree, Disagree, Neutral, Agree, and Strongly agree. [2] To distribute the survey, first we submitted *issues* in the 72 repositories in our dataset of 81 repositories that allowed us to submit an issue. Second, we sent emails to 57 developers of 45 repositories when an email address of a feature toggles' contributor was available. We report the result of the survey in Section VI. After evaluating the acceptance by developers of the proposed heuristics, to address RQ_E, we analyzed the default branch of each of 22 randomly-selected GitHub repositories in the evaluation set.

First, using the commit we identified the repository by, 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* [19] 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. The number of feature toggles was identified via manual inspection of the repository code base.

Next, to examine the relationship between the heuristics and the 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 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 [21] for binary metrics.
Hedges' g to measure the effect sizes (amount of difference) because it is well suited for small sample sizes [22], [23].
Pearson's r to measure the strength and direction of correlation (linear relationship) between two variables [24].

We report the observed relation between heuristics, FT-metrics, and context metrics in Section VI.

IV. FT-METRICS

We now describe the FT-metrics we identify to support heuristics of using feature toggles in the repository. We present the Metrics prior to the Heuristics (which we present in Section V) because the use of the metrics make the heuristics easier to understand. The metrics and the heuristics were collected iteratively and simultaneously as we analyzed the repositories. Following the steps in Section III-B, we first manually analyze 59 repositories in the analysis 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

[2]Supplementary material includes the survey questionnaire.

metrics based on our observations in the repositories. From our analysis, we identified 12 metrics. Based on their effect on the code base, existing classification and use of these metrics in the literature, we categorize these metrics into three categories: *Complexity*, *Comprehensibility*, and *Maintainability*.

Below, we list the 12 FT-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:

- *M1: Number of added paths in code (Numeric)* is computed using the McCabe's Cyclomatic Complexity [25]. 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 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. To compute this metric, we count the number of feature toggle value checking methods in the feature toggle class manually and assume all the methods have equal complexities so our weight for all is 1.0.

Comprehensibility measures how easy it is for practitioners to understand the use of feature toggles in the repository. We compute comprehensibility using the following four 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. Guidelines can be provided as a document in repositories or as detailed comments in feature toggles' configuration files.
- *M4: Intention revealing names (Binary)* for variables and methods is a well-known practice in coding [27]. 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. This metric is a subjective metric.
- *M5: Use of comments (Binary)* as human-readable notes that support the source code is an important coding practice [28] that helps developers to understand the purpose and behavior of the feature toggles. "yes" is a preferred value for this metric. If at least half of the feature toggles in a repository have supporting comments next to their definitions or usages, we assign "yes" as value of 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. Listing 4 in the next Section is an example of including descriptions. Unlike comments which are not available everywhere, object attributes are available everywhere a practitioner wants.

Maintainability measures how easy it is for practitioners to maintain feature toggles in their code. This category includes the following six 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 number of the files could be context-dependent. For instance, separate platforms can have separate configuration files.
- *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. 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 [29]. 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.
- *M10: Presence duplicate code (Binary)* is a previously-defined code smell [30]. Duplicate code is a problem of repeating the same block of the code in the repository. In this study, we consider duplicate code that contains feature toggles. If a duplicated code contains a feature toggle, in

[3]This approach is used in the implementation of tools to calculate cyclomatic complexity metric of the code[26].

case of updating or removing the toggle all the occurrences of the duplicate code need to be updated or removed.

- *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. 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. 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. 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.

V. 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 provide examples found in the repositories of following or not-following that heuristic. Our naming convention for the example is H (for Heuristic), followed by the number of the heuristic, followed by the 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: the first following example for heuristic 1.

For each heuristic, we determine the FT-metrics which support the heuristic based on our observations and literature. FT-metrics substantiate the heuristics and the practitioners can follow the the heuristic without the need to do the measurement of FT-metrics. Table II shows the mapping of the FT-metrics and the heuristic.

TABLE II
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)						✓	

A. Shared Method to Check Value

Heuristic 1

Using one *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()` [8]. 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 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).

Following Example ($\mathbf{H1_{FE1}}$): Listing 2 shows an example of using one shared feature toggle value checking method [31]. 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 [31].

Not-Following Example ($\mathbf{H1_{NFE2}}$): The code in Listing 3 shows an example of each feature toggle having its own function to check its value [32]. 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 [32].

B. *Self-Descriptive Feature Toggles*

Heuristic 2

Adding a description field as a meta-attribute for each feature toggle 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).

Following Example ($H2_{FE1}$): Listing 4 is an example of having self-descriptive feature toggles [33]. 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 [33].

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 [34], 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 [35], 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 names for feature toggles, and the configuration file is well commented.

A practice suggested by practitioners [8] 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 at an appropriate 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[4] is using names which are not meaningful. In this repository, a toggle was named `FEATURE_TOGGLE_520`, which does not convey its purpose.

C. *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, dead code could result.

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 [8], 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).

Following Example ($H3_{FE1}$): In the grid repository [36], 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 [37], 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 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 may exist in the code. Also, how can a developer specify if a

[4]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

change is significant or not? The 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 toggles and the absence of removal guidelines, the possibility of creating dead code is high.

D. 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. The additional number of files to update causes an increase in the effort developers should make. The more number of path in the code, the more the complexity the code 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).

Following Example (H4$_{\mathbf{FE1}}$): 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 [1]. Listing 5 shows an example of using feature toggles to set the value of another variable. The `canFork` variable is checked in the rest of the file instead of checking the three conditions in the example [38]. 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, the toggle can be removed or updated 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$_{\mathbf{NFE2}}$): One way to store the value of the toggle is using configuration files, but in some repositories more than one configuration file exists for the same set of feature toggles. For instance, the MTC repository [39] 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. The reason for using more than one file could be project-specific such as managing multiple platforms but it increases the complexity and decreases the maintainability of the 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.

E. Avoid Redundancy in Feature Toggles

Heuristic 5

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

Duplicate code is a code smell [41]. 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 are added to the code; and (2) may create dead code, if they are not removed after their purposes are accomplished. One of the refactoring patterns to solve the issue of duplicate code is *Extract Method* [41] that could be used with feature toggle's if-else statement fragment of code.

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

Not-Following Example (H5$_{\mathbf{NFE1}}$): In the refocus repository [42], the code in Listing 6 is used in a file twice which is duplicate code with usage of a toggle. 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 [42].

F. 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).

Following Example (H6$_{\mathbf{FE1}}$): In the MTC repository [43], 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.

G. *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. 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 [44]. This is an example of why dead code should be avoided when using feature toggles.

Metrics addressed: (1) Number of files (M7), (2) Lines of code (M9), (3) Dead code (M11).

Following Example ($H7_{FE1}$): The commit [45] is an example of complete removal of a toggle. The developers remove the toggle from the configuration files, code, and test cases.

VI. Phase 2: Evaluation

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

SRQ_{AD} (Developers' agreement): To what extend developers agree with proposed heuristics and the effects of following them on identified FT-metrics?

SRQ_{HM} (Heuristics and metrics): 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, adoption of identified heuristics, and values of FT-metrics?

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

To address these proposed sub-research questions, we conduct a survey on developers in 81 repositories and a case study on 22 repositories selected as part of the evaluation set.

A. *Developers Agreement by Survey (SRQ_{AD})*

To evaluate the proposed heuristics with actual developers, we designed a survey in which we asked the developers the extent to which they agree on our proposed heuristics and their support by FT-metrics. We used five agreement options based on a *Likert* scale: Strongly disagree(1), Disagree(2), Neutral(3), Agree(4), and Strongly agree(5). N/A option was also provided for developers who do not want to answer a question. To reach out to developers and elicit responses on the survey questionnaire from them, we initially submitted 72 issues in the repositories in 72 repositories in our dataset. The settings of the rest of the repositories did not allow us to submit an issue. We received 8 responses after reaching out via issues. Next, we sent 57 emails to developers of 45 repositories associated with feature toggles' commits and changed files and received 12 responses. For the rest of the repositories, no email address of feature toggles' contributors was available.

Figure 2 shows the extent to which the developers agree to our proposed heuristics. Based on the median values of the level of agreements, which are showed in Right side of this Figure, survey respondents are agree with the correctness of all of the proposed heuristics and their support by FT-metrics ($\tilde{x}$ >4). Also, developers were asked about difficulty of managing feature toggles and increment of technical debt by using feature toggles. The median values of their responses show that managing feature toggles is *somewhat difficult* for them and they are *agree* that using feature toggles can increase technical debt in the code. RMH: Include dev demographics

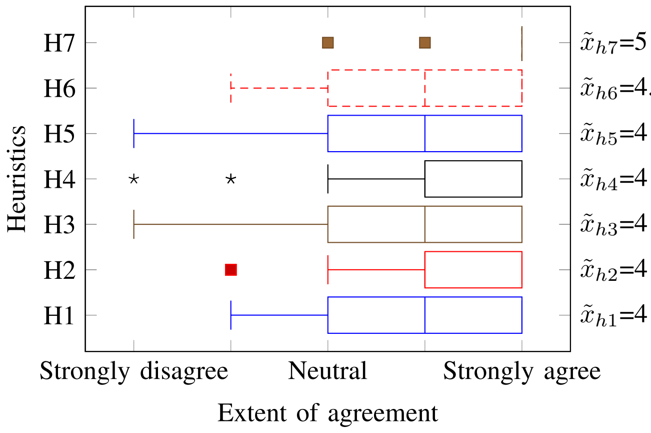

Fig. 2. Box plot summarizing the extent of agreement for each heuristic. $\tilde{x}$ indicates the median.

B. *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 IV, manually. An automatic approach to compute FT-metrics is relevant a future direction. We analyze the feature toggles in the last snapshot of the repository and do not consider metrics such as lifetime of the feature toggles in the repository.

For each repository, we also manually identify if the repository follows the heuristics proposed in Section V. Following are the criteria to mark if 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 for at least half of the feature toggles: (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 in the list of 123 selected commits.

We apply two tailed t-test [20] and Fisher's exact test [21], and compute Hedges' g [22], [23] and Pearson's r [24] in our analyses.

C. Heuristics and FT-metrics (SRQ$_{HM}$)

We now address SRQ$_{\text{HM}}$ on the relation between the adoption of the seven heuristics and the values of the twelve metrics. Table III 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. Bold values are better. Table III also shows the p-values (p) of two tailed t-test for context metrics and numeric metrics, and Fisher's exact test for binary metrics. In this table 'n' is # of repositories out of 22 which follow a heuristic, M1–M12 are FT-metrics, 'F' corresponds to repositories which follow a heuristic, and 'NF' corresponds to the repositories which do not follow that heuristic.
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. 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.
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) by definition. 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).

TABLE III
OBSERVATIONS FROM THE CASE STUDY FOR 22 REPOSITORIES.

			H1	H2	H3	H4	H5	H6	H7
Context	n (# repositories)	F	11	7	6	18	17	8	7
		NF	11	15	16	4	5	14	15
	# contributors	F	87	55	131	21	21	51	52
		NF	9	45	17	173	140	47	87
		p	0.09	0.82	0.19	0.26	0.26	0.93	0.63
	# 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
		p	0.20	0.72	0.51	0.46	0.51	0.21	0.85
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**
		p	0.88	<**0.01**	0.75	0.22	0.23	0.75	0.10
	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**
		p	0.50	0.41	0.32	0.43	0.40	0.40	0.34
	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
		p	0.63	0.91	0.17	0.13	0.17	0.79	0.37
	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
		p	0.47	0.87	0.09	0.08	0.06	0.38	0.66
	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**
		p	0.50	0.58	0.37	0.77	0.93	0.15	0.10
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**
		p	0.15	0.62	<**0.01**	1.00	0.59	0.62	0.59
	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**
		p	1.00	0.52	0.53	0.47	1.00	1.00	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
		p	0.64	<**0.01**	0.63	1.00	1.00	1.00	0.46
	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
		p	0.59	<**0.01**	1.00	1.00	1.00	1.00	0.19
	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
		p	1.00	0.35	**0.05**	<**0.01**	<**0.01**	0.34	1.00
	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
		p	1.00	**0.05**	**0.05**	1.00	1.00	1.00	**0.03**
	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
		p	0.18	1.00	0.14	0.60	0.31	<**0.01**	0.29

D. 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.

E. 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.

VII. THREATS TO VALIDITY

We identify the following 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, developing heuristics and identifying FT-metrics could be subjective to the first author's knowledge. To mitigate this threat, we asked second author to review the heuristics and FT-metrics and give feedback. We also conduct a survey to evaluate the proposed heuristics and FT-metrics with actual developers of GitHub repositories in our data set.

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.

VIII. 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) 7 heuristics to structure feature toggles, and (2) 12 metrics, which we refer as FT-metrics, to support the effect of incorporating heuristics. We evaluate our developed heuristics and proposed metrics via conducting a survey and a case study on 22 repositories. The survey respondents, who are developers associated with using and managing feature toggles in the analyzed repositories, are agree with the correctness of proposed heuristics and the effect of using them on FT-metrics. Our case study's 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.

The FT-metrics can be validated by 47 criteria to validate software metrics extracted by Meneely et al. [46] in the future works. An automated tool to recommending heuristics and compute FT-metrics 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] M. T. Rahman, L.-P. Querel, P. C. Rigby, and B. Adams, "Feature toggles: Practitioner practices and a case study," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*. Austin: ACM, 2016, pp. 201–211.

[2] C. Parnin, E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor *et al.*, "The top 10 adages in continuous deployment," *IEEE Software*, vol. 34, no. 3, pp. 86–95, 2017.

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

[4] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering (TSE)*, vol. 20, no. 6, pp. 476–493, 1994.

[5] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Boston: Pearson Education, 2010.

[6] M. Fowler, "Continuous delivery," [Online]. Available: https://martinfowler.com/bliki/ContinuousDelivery.html, Accessed 6 August 2019,, 2013.

[7] R. Harmes, "Flipping out," [Online]. Available: http://code.flickr.net/2009/12/02/flipping-out/, Accessed 6 August 2019.

[8] R. Mahdavi-Hezaveh, J. Dremann, and L. Williams, "Feature toggle driven development: Practices used by practitioners," *arXiv preprint arXiv:1907.06157*, pp. 1–25, Jul. 2019.

[9] P. Hodgson, "Feature toggles (aka feature flags)," [Online]. Available: https://martinfowler.com/articles/feature-toggles.html, Accessed 6 August 2019.

[10] B. Hodges, "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.

[11] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne, "Controlled experiments on the web: Survey and practical guide," *Data mining and knowledge discovery*, vol. 18, no. 1, pp. 140–181, 2009.

[12] A. A. U. Rahman, E. Helms, L. Williams, and C. Parnin, "Synthesizing continuous deployment practices used in software development," in *Proceedings of the IEEE Agile Conference*. IEEE, 2015, pp. 1–10.

[13] M. K. Ramanathan, L. Clapp, R. Barik, and M. Sridharan, "Piranha: Reducing feature flag debt at uber," in *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE)*. Seoul: ACM, jul 2020, pp. 1–10.

[14] J. Meinicke, J. Hoyos, B. Vasilescu, and C. Kästner, "Capture the feature flag: Detecting feature flags in open-source," in *International Conference on Mining Software Repositories*, 2020.

[15] J. Meinicke, C.-P. Wong, B. Vasilescu, and C. Kaestner, "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)*, 2020, to appear.

[16] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo, "Software configuration engineering in practice: Interviews, survey, and systematic literature review," *IEEE Transactions on Software Engineering (TSE)*, 2018.

[17] J. Meinicke, C.-P. Wong, C. Kästner, T. Thüm, and G. Saake, "On essential configuration complexity: Measuring interactions in highly-configurable systems," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 483–494.

[18] R. Likert, "A technique for the measurement of attitudes," *Archives of Psychology*, 1932.

[19] A. Danial, "Cloc: Count lines of code," [Online]. Available: http://cloc.sourceforge.net/, Accessed 24 July 2019.

[20] M. Hollander and D. A. Wolfe, *Nonparametric Statistical Methods*. New York: Wiley, 1999.

[21] R. A. Fisher, "Statistical methods for research workers," in *Breakthroughs in statistics*. Springer, 1992, pp. 66–70.

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

[23] L. V. Hedges and I. Olkin, *Statistical Methods for Meta-Analysis*. Orlando: Academic Press, Inc., 2014.

[24] D. Freedman, R. Pisani, and R. Purves, *Statistics (4th Edition*. New York: W. W. Norton & Company, 2007.

[25] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering (TSE)*, vol. 2, no. 4, pp. 308–320, 1976.

[26] S. SA, "Metric definitions," [Online]. Available: https://docs.sonarqube.org/latest/user-guide/metric-definitions/, Accessed 27 July 2019.

[27] N. B. Dale, C. Weems, and M. R. Headington, *Introduction to Java and Software Design: Swing Update*. Jones & Bartlett Learning, 2003.

[28] G. Penny and A. A. Takang, *Software Maintenance: Concepts and Practice*. World Scientific, 2003.

[29] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Proceedings of the 6th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 2007, pp. 30–39.

[30] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[31] NAV, "pleiepengesoknad," [Online]. Available: https://github.com/FeatureToggleStudy/pleiepengesoknad/blob/master/src/app/utils/featureToggleUtils.ts#L7 Accessed 27 February 2020.

[32] Education and S. F. A. Team, "Cfs-backend," [Online]. Available: https://github.com/SkillsFundingAgency/CFS-Backend/blob/d4461bda36e3d785909350233f833594984823c3/Debugging/CalculateFunding.DebugAllocationModel/FeatureToggles.cs Accessed 21 October 2019.

[33] ——, "Cfs-frontend," [Online]. Available: https://github.com/featuretogglestudy/CFS-Frontend/blob/39961217b8aabd665c71b108903d87014a41582c/DevOps/frontend-azure.dfe.json#L237 Accessed 27 February 2020.

[34] Automattic, "wp-calypso," [Online]. Available: https://github.com/Automattic/wp-calypso/blob/8d1bf0b0146fc341288059c765e8a3bf8c8bb7ef/client/me/purchases/manage-purchase/purchase-meta.jsx Accessed 21 October 2019.

[35] Salesforce, "refocus," [Online]. Available: https://github.com/salesforce/refocus/blob/18116cb7df73c0db70af3c6115342ccf93db6534/config/toggles.js Accessed 21 October 2019.

[36] T. Guardian, "grid," [Online]. Available: https://github.com/guardian/grid Accessed 27 February 2020.

[37] D. of Veterans' Affairs, "myservice-prototype," [Online]. Available: https://github.com/AusDVA/myservice-prototype Accessed 9 October 2019.

[38] P. Network, "recogito2-workspace-frontend," [Online]. Available: https://github.com/pelagios/recogito2-workspace-frontend/blob/367723732cfa74c4f38e542b88c0a4491789cc04/src/profile/Profile.jsx Accessed 9 October 2019.

[39] D. for Education, "Mtc," [Online]. Available: https://github.com/DFEAGILEDEVOPS/MTC/tree/0eb2d765b6683c90c852ba21c225742f07f050b9/admin/config Accessed 9 October 2019.

[40] SatoshiPay, "solar," [Online]. Available: https://github.com/satoshipay/solar/blob/122830c16ba1fcbf82f4b388f92ea0483444c938/src/platform/settings.ts Accessed 9 October 2019.

[41] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[42] Salesforce, "refocus," [Online]. Available: https://github.com/salesforce/refocus/blob/18116cb7df73c0db70af3c6115342ccf93db6534/jobQueue/jobWrapper.js Accessed 9 October 2019.

[43] D. for Education, "Mtc," [Online]. Available: https://github.com/DFEAGILEDEVOPS/MTC/commit/0eb2d765b6683c90c852ba21c225742f07f050b9#diff-8633026cf78840f2cb5a5b32fe1aa00f Accessed 11 October 2019.

[44] "Knightmare: A devops cautionary tale," [Online]. Available: https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/ Accessed 24 April 2019.

[45] D. for Education, "Mtc," [Online]. Available: https://github.com/DFEAGILEDEVOPS/MTC/commit/0eb2d765b6683c90c852ba21c225742f07f050b9 Accessed 27 February 2020.

[46] A. Meneely, B. Smith, and L. Williams, "Validating software metrics: A spectrum of philosophies," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 4, pp. 1–28, 2013.

APPENDIX