ABSTRACT

MAHDAVI HEZAVEH, REZVAN. A Comprehensive Model of Software Configuration. (Under the direction of Dr. Laurie Williams).

Using feature toggles is a technique that allows developers to either include or exclude a block of code with a variable in a conditional statement. Software companies increasingly use feature toggles to facilitate continuous integration and continuous delivery. On the other hand, configuration options are programmatic techniques to include or exclude functionality in software easily. Software companies usually use configuration options to implement software product lines. Feature toggles and configuration options have similar definitions, and the border between their definitions is blurred. So, both concepts are considered *software configuration* in the literature.

Using feature toggles inappropriately may cause problems that can have a severe impact, such as code complexity, dead code, and system failure. For example, the erroneous repurposing of an old feature toggle caused Knight Capital Group, an American global financial services firm, to go bankrupt due to the implications of the resultant incorrect system behavior. Using configuration options also has challenges, such as testing multiple software variants in parallel. For example, the Linux kernel has over 15000 configuration options, and (considering all configuration options as boolean) it has up to 2^{15000} product variants for testing.

The research contributions on feature toggles and configuration options often focused on either feature toggles or configuration options. However, focusing on the similarities and differences between these two techniques can enable a more fruitful combined family of research. Moreover, a common terminology may enable meta-analysis, a more practical application of the research on feature toggles and configuration options, and prevent duplication of research effort.

Using a comprehensive model of software configuration can help researchers in a combined family of research on the software configuration.

We conduct four studies to (1) identify practices of using feature toggles by practitioners; (2) propose heuristics and metrics for structuring feature toggles in the code; (3) extend an existing model of software configuration to cover feature toggle and enable meta-analysis of a family of research; (4) application of the extended model of software configuration on related publications, perform a meta-analysis, and propose guidelines for researchers to document context variables in their software configuration research studies.

From these studies, we contribute the following: (1) 17 practices of using feature toggles by practitioners in 4 categories; (2) 7 heuristics to guide structuring feature toggles in the source code, and 12 metrics to support the principles embodied in these heuristics; (3) an extended model of software configuration to cover feature toggle concept as well as configuration option concept; (4) a comparison between feature toggle and configuration option related publications based on the application of the extended model of software configuration; (5) provide guidelines for researchers to use the extended model of software configuration to document context variables in their research studies; and (6) comprehensive definitions for feature toggles and configuration options based on their different characteristics in the application of the extended model of software configuration.

A Comprehensive Model of Software Configuration

by
Rezvan Mahdavi Hezaveh

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2023

APPROVED BY:

Dr. Christopher Parnin	Dr. Sarah Heckman
Dr. Julie Earp	Dr. Laurie Williams Chair of Advisory Committee

DEDICATION

To all the brave women and men in my home country, Iran, who are fighting for human rights. *Woman, Life, Freedom.*

BIOGRAPHY

Rezvan Mahdavi Hezaveh received a B.Sc. degree in Software Engineering from the Azad University of Karaj, Alborz, Iran, and a M.Sc. degree in Software Engineering from the Sharif University of Technology, Tehran, Iran. After spending 3 years in the industry, in August 2017, she started her work toward a Ph.D. in Computer Science at North Carolina State University, Raleigh, NC, under the guidance of Professor Laurie Williams. Her doctoral research interests mainly include empirical software engineering.

ACKNOWLEDGEMENTS

There are many, without whom this work would not have been possible, and I would like to express my gratitude. First and foremost, I would like to express my gratitude and appreciation to my advisor, Professor Laurie Williams, for her valuable guidance and encouragement throughout the research. I can never thank her enough for the support and encouragement that I received during my study. It was a great honor for me to know and work with her. I also would like to thank Professor Christopher Parnin, Professor Sarah Heckman, and Professor Julie Earp for spending their precious time reviewing my thesis and attending my exams.

Furthermore, I am so thankful for meeting and working with these great colleagues here at NC State University: Dr. Akond rahman, Dr. Nirav Ajmeri, Sarah Elder, Rayhanur Rahman, Nasif Imtiaz, Nusrat Zahan, and Setu Kumar Basak.

I would also like to thank my parents and sisters for their endless support and love. It was indeed their encouragement and patience that inspired me to come this far and be where I am now. Last but not least, I would like to thank Ali, my kind husband, who has always been there for me, for his wisdom, encouragement, and great soul.

TABLE OF CONTENTS

List of Tables . . . . . **viii**

List of Figures . . . . . **ix**

Chapter 1 Introduction **1**

Chapter 2 Background and Related Work **5**
2.1 Background 5
2.1.1 Continuous Integration(CI) and Continuous Delivery (CD) 5
2.1.2 Feature Toggles 6
2.1.3 Configuration Options 7
2.1.4 Grey Literature 7
2.1.5 Metrics 8
2.2 Related Work 8
2.2.1 Feature Toggles 8
2.2.2 Configuration options 10
2.2.3 Software Configurations 11
2.2.4 Frameworks for Structuring Families of Research 12

Chapter 3 Software Development with Feature Toggles **13**
3.1 Motivation 13
3.2 Research Methodology 14
3.2.1 Step One: Searching Initial Artifacts 14
3.2.2 Step Two: Identification of Practices and Categories 16
3.2.3 Step Three: Searching Company-specific Grey Literature 19
3.2.4 Step Four: Extraction of Practice Usage from Company-specific Artifacts 19
3.2.5 Step Five: Conducting Survey 20
3.2.6 Step Six: Analysis of Usage Frequency of Practices 20
3.3 Results 21
3.3.1 Feature Toggles Practices 21
3.3.2 Usage of Practices in Industry 33
3.4 Recommendations for Practitioners 40
3.5 Limitations 42
3.5.1 Finding artifacts 42
3.5.2 Identification of Practices and Categories 43
3.5.3 Extraction of Practice Usage from Company-specific Artifacts 43
3.5.4 Conducting Survey 43
3.6 Conclusion 44

Chapter 4 Heuristics and Metrics for Structuring Feature Toggles in Source Code 46
4.1 Motivation 46

4.2 Methodology . . . 47
4.2.1 Dataset–Repositories . . . 47
4.2.2 Phase One: Observational Study . . . 50
4.2.3 Phase Two: Survey and Case Study . . . 51
4.3 FT-Heuristics . . . 53
4.3.1 Shared Method to Check Value (26) . . . 53
4.3.2 Self-Descriptive Feature Toggles (19) . . . 54
4.3.3 Guidelines for Managing Feature Toggles (10) . . . 56
4.3.4 Use Feature Toggles Sparingly (53) . . . 56
4.3.5 Avoid Duplicate Code in Using Feature Toggles (57) . . . 57
4.3.6 Test Cases for Feature Toggles (17) . . . 58
4.3.7 Complete Removal of a Feature Toggle (21) . . . 58
4.4 FT-metrics . . . 59
4.5 Survey and Case Study . . . 61
4.5.1 Practitioners Agreement by Survey (SRQ_{PA}) . . . 62
4.5.2 Repository Inspection . . . 65
4.5.3 FT-heuristics and FT-metrics (SRQ_{HM}) . . . 65
4.6 Threats to Validity . . . 70
4.7 Lessons Learned . . . 71

Chapter 5 Comprehensive Model of Software Configuration (MSCv2) . . . 73
5.1 Motivation . . . 73
5.2 Methodology . . . 74
5.2.1 Meinicke et al.: Feature Toggles vs. Configuration Options . . . 74
5.2.2 Siegmund et al.: Model of Software Configuration (MSC) . . . 76
5.2.3 Phase One: Find Feature Toggle Management System Repositories . 76
5.2.4 Phase Two: Extending the Model of Software Configuration (MSCv2) 77
5.2.5 Phase Three: Application of Extended Model of Software Configuration (MSCv2) . . . 79
5.3 The Extended Model of Software Configuration (MSCv2) . . . 80
5.3.1 Stakeholder (S1, S2, S3) . . . 81
5.3.2 Stage (S1, S3) . . . 82
5.3.3 Binding Time (S1, S3) . . . 83
5.3.4 Type (S1, S3) . . . 83
5.3.5 Artifact (S1, S2, S3) . . . 84
5.3.6 Life Cycle (S1, S2, S3) . . . 85
5.3.7 Complexity (S1, S2, S3) . . . 87
5.3.8 Intent (S1, S2, S3) . . . 87
5.3.9 Activation Strategy (S3) . . . 89
5.4 Application of the Extended Model of Software Configuration (MSCv2) . . . 90
5.4.1 Similar Papers on Industry Practices . . . 90
5.4.2 Studies on Chrome repository . . . 91
5.4.3 Chrome Repository . . . 92

5.4.4 Takeaways . . . 94
5.5 Threats to Validity . . . 94
5.6 Conclusion . . . 95

Chapter 6 Meta-analysis of Software Configuration Related Publications . . . 96
6.1 Motivation . . . 96
6.2 Methodology . . . 98
6.2.1 Phase One: Search for Publications . . . 98
6.2.2 Phase Two: Application of MSCv2 on Selected Publications . . . 101
6.2.3 Phase Three: Meta-analysis of the publications . . . 102
6.3 Results . . . 103
6.3.1 All publications . . . 103
6.3.2 Feature toggle versus Configuration option . . . 105
6.3.3 High-level goals . . . 111
6.4 Discussion . . . 112
6.4.1 Research Gaps . . . 112
6.4.2 Definitions of Feature Toggle and Configuration Option . . . 113
6.4.3 Relations between dimensions and values . . . 114
6.5 Guidelines to Use MSCv2 . . . 115
6.6 Threats to validity . . . 116
6.7 Conclusion . . . 116

Chapter 7 Conclusion and Future Work . . . 117
7.1 Conclusion . . . 117
7.2 Future Work . . . 118

References . . . 120

APPENDICES . . . 136
Appendix A Survey questions for Chapter 3 . . . 137
Appendix B Preliminary Statistical Analysis for Chapter 4 . . . 140

LIST OF TABLES

Table 3.1 The quality assessment checklist of grey literature for software engineering adapted from (1). 16
Table 3.2 The level of confidence. The number in parenthesis of each level is the number of practices that fell into the category. 19
Table 3.3 Related practices. 34
Table 3.4 Feature toggle practices and their usage in the industry 35
Table 3.5 38 Companies and their usage of identified practices from company-specific artifacts 45

Table 4.1 Characteristics of the identified repositories. 49
Table 4.2 Hypothesized relationship between FT-heuristics and FT-metrics. . . 62
Table 4.3 Survey respondents' experience and frequency of using feature toggles. Underline indicates median. 63
Table 4.4 Observations from case study of 71 repositories for context metrics. . 67
Table 4.5 Observations from case study of 71 repositories for numeric metrics. Gray cells indicate the hypothetical relation between FT-heuristics and FT-metrics as mentioned in Table4.2. 68
Table 4.6 Observations from case study of 71 repositories for binary metrics. Gray cells indicate the hypothetical relation between FT-heuristics and FT-metrics as mentioned in Table4.2. 69

Table 5.1 Programming language of the identified repositories. 77

Table 6.1 Analyzed relevant publications 104

Table B.1 Best Subset Selection result for FT-metrics. ↑ indicates the predictor is in the overall best model with a positive coefficient. ↓ indicates the predictor is in the overall best model with a negative coefficient. 142

LIST OF FIGURES

Figure 2.1 An example of a feature toggle. 6
Figure 2.2 An example of configuration options (2) 7

Figure 3.1 The research methodology. 14
Figure 3.2 An example of using the open coding technique. 15
Figure 3.3 The lifecycle of a feature toggle, adapted from (3). 21
Figure 3.4 Number of artifacts found in Steps 1, 3 and 4 36
Figure 3.5 Frequency of using the subset of feature toggle practices with the Likert scale based on the survey 40

Figure 4.1 Research methodology outline. 48
Figure 4.2 Summary of the practitioner survey. 64

Figure 5.1 Research methodology outline. 75
Figure 5.2 Extended Model of Software Configuration (MSCv2). 81
Figure 5.3 Application of the MSCv2 on three publications on practitioners' practices: (a) (4), (b) (5), (c) (6) 91
Figure 5.4 Application of the MCSv2 to two research studies on feature toggles on Chrome: (a) (6), (b) (7) 92
Figure 5.5 Application of the MSCv2 on Chrome 93

Figure 6.1 Research methodology outline. 99
Figure 6.2 aggregated MSCv2 instances for all 57 publications 105
Figure 6.3 aggregated MSCv2 instances for 10 feature toggle-related publications 106
Figure 6.4 aggregated MSCv2 instances for 47 configuration option-related publications 107
Figure 6.5 p-values of Fisher exact test with Bonferroni correction 110

CHAPTER

1

INTRODUCTION

Developers guard blocks of code with a variable as a *feature toggle*[1] in conditional statements, and by changing the value of the variable, enable or disable that part of the code in the system's execution. The use of feature toggles is a technique often used in continuous integration (CI) and continuous delivery (CD) contexts that allow teams to incrementally integrate and test a new feature even when the feature is not ready to be released (6)(9). Developers also use feature toggles for other purposes, such as gradual rollout and experiments. However, feature toggles can turn into technical debt (10).

Configuration options are key-value pairs that allow software customization by including or excluding functionality in a software system (11). Using configuration options allows developers to implement software product lines (12; 11; 13).

Both feature toggles and configuration options are techniques practitioners use in the software development process. Meinicke et al. (11) state that feature toggles can be considered a use of configuration options for a new purpose, such as CI/CD, or configuration options can be seen as a subset of feature toggles. Despite feature toggles and configuration options being similar concepts, they have distinguishing characteristics, such as their users

[1] Feature toggles are also called feature flags, feature bits, feature flippers, and feature switches (8).

and lifetime. Whereas configuration options are used by end-users and can exist permanently, feature toggles are used by developers and are ideally removed from code (except for long-lived feature toggles). However, in reality, a large fraction of feature toggles stay in the code base forever (14).

The border between the definition of the feature toggle and the configuration option is blurred in the research community. As an example in (15), researchers defined feature toggles as *Software Product Line (SPL) configurations* that *developers* change and get values at *compile-time*. This definition has differences with definitions of feature toggles and has similarities with definitions of configuration options in other research publications. For instance, Rahman et.al (16) stated that feature toggles could be changed by *developers* at *run-time*, and Schubert et al. (17) state that *compile-time* configuration options are used in *SPLs* to encode a set of software products. Based on the blurred border between these two concepts, separating them from each other is not practical, and both concepts can consider categories of *software configuration* (11; 12). In the rest of this thesis, we use the term *software configuration* to refer to the joint concepts of feature toggles and configuration options.

A software development *practice* is an activity or step carried out to achieve a goal during software development. For example, unit testing is a practice for white-box testing of implementation code. The identification and categorization of feature toggle practices used by practitioners, and the way feature toggles are structured in the code may help software practitioners to use toggles more efficiently and to control the accumulation of technical debt. Software practitioners prefer to learn through the experiences of other software practitioners (18). As such, we obtained practice usage and guidelines on how to structure and manage feature toggles from practitioners' experiences and present these findings in Chapters 3 and 4.

Researchers perform research studies in both feature toggles and configuration options research areas. In addition, Meinicke et al. show that using these two techniques may cause similar problems, and there are similar solutions as well (11). So, if a research question is answered in the configuration option research area, the result may be applicable to feature toggles as well. However, researchers may miss this opportunity because they are unaware of the other technique, and solve the same question again. *Ignoring similarities and differences between the characteristics of software configuration in research studies can lead to duplication of effort, inefficiency in research, and unclear comparisons and generalizations.* Every family of research can benefit from a framework or common ontology to enable meta-analysis on them. Hence, researchers can benefit from a *model of software*

configuration to define context variables in their software configuration-related research studies clearly and to help other researchers to use their results and research design in other studies (12). We present our model in Chapters 5 and 6.

Thesis Statement: *Using a comprehensive model of software configuration can help researchers in a combined family of research on the software configuration.*

We evaluate the thesis statement by answering the following research questions. Within the body of this document, each research question has two or more sub-research questions:

RQ1: How do software practitioners use feature toggles in the industry?

RQ2: How are feature toggles structured in the code?

RQ3: How can we extend the existing Model of Software Configuration (MSC) to cover both the feature toggle and configuration option concepts?

RQ4: Can the application of the extended Model of Software Configuration be used to perform a meta-analysis of a family of research on software configurations?

In this thesis, we make the following contributions:

1. A list of practices used to support software development with feature toggles, and an analysis of the frequency of usage of feature toggle practices in the industry (Chapter 3);
2. Development of heuristics to guide the structuring of feature toggles (Chapter 4);
3. Identification of metrics to support the principles embodied in the heuristics (Chapter 4);
4. A case study analyzing practitioner adherence to the heuristics in open source software, and the relation of heuristics with metrics (Chapter 4);
5. An extended model of software configuration, and usage guidelines for researchers to document context variables in their research studies (Chapters 5 and 6);
6. Modeling, analysis, and systematic comparison of the software configuration related publications (Chapters 5 and 6);
7. Modeling the software configuration of Chrome as an industrial system (Chapters 5);
8. A set of challenges and research gaps related to software configurations (Chapters 6);
9. Comprehensive definitions for feature toggles and configuration options (Chapters 6);
10. Artifacts:
    - A database of GitHub repositories that use feature toggles in their development cycle, and examples of good and bad structuring of the feature toggles in these repositories (Chapter 4);
    - A list of open-source feature toggle management systems on GitHub (Chapters 5).

The rest of this thesis is organized as follows: in Chapter 2, we describe the background

and related works of the research area. In Chapter 3, we answer RQ1 by identifying practitioners' practices. In Chapter 4, we answer RQ2 by proposing heuristics and identifying metrics. In Chapter 5, we answer RQ3 by extending the existing Model of Software Configuration. And in Chapter 6, we answer RQ4 by conducting a meta-analysis of software configuration-related publications by applying the extended Model of Software Configuration to the publications. Chapter 7 includes the conclusion and future work.

CHAPTER

2

BACKGROUND AND RELATED WORK

This chapter describes the background and relevant related work.

2.1 Background

In this section, we first briefly define Continuous Integration (CI) and Continuous Delivery (CD). Next, we explain the feature toggle, configuration options, grey literature, and metrics.

2.1.1 Continuous Integration(CI) and Continuous Delivery (CD)

Companies must deliver valuable software rapidly to be competitive. This expectation leads companies to use Continuous Integration (CI) and Continuous Delivery (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 (19) in short intervals. CD is a practice for keeping the software in a state that can be released to a production environment anytime (20). CI/CD refers to a combination of these two practices and enables delivering code changes frequently. Using feature toggles is one of the techniques often used by software companies that practice CI/CD (9).

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

```

Figure 2.1: An example of a feature toggle.

2.1.2 Feature Toggles

Programming languages have long provided the language constructs to implement feature toggles. However, the first use of this language construct to support CI/CD was at Flickr in 2009 (21). Figure 2.1 is an example of a feature toggle. In this example, the dynamic choice of a search algorithm depends on the value of the useNewAlgorithm toggle. If the value of this toggle is true, then the new search algorithm is used; otherwise, the Search method calls the old search algorithm.

Feature toggles have been categorized into five types in software systems (6) and (22):

- *Release toggles*: Toggles are used to add new features in a trunk-based development context. In trunk-based development, all developers commit changes to one shared branch. Using release toggles in trunk-based development supports CI/CD for partially-completed features (22) (6).
- *Experiment toggles*: Toggles are used to perform experimentation on the software, such as is done by Microsoft (23) (24), to evaluate new feature changes and their influence on user-observable behavior (22).
- *Ops toggles*: Toggles are used to control the operational aspect of the system behavior. When a new feature is deployed, system operators can disable the feature quickly if it performs unexpectedly (22).
- *Permission toggles*: Toggles are used to provide the appropriate functionality to a user, for instance, special features for premium or paid users (22). Permission toggles are also called long-term business toggles in (6).
- *Development Toggles*: Toggles are used for enabling or disabling certain features to test and debug code (6).

```
#ifdef HELLO
char * msg = "Hello World\n";
#endif
#ifdef BYE
char * msg = "Bye bye\n";
#endif

main(){
    printf(msg);
}

```

Figure 2.2: An example of configuration options (2)

2.1.3 Configuration Options

Configuration options are key-value pairs to include or exclude functionality in a software system or the setting parameters, such as buffer size (11). Practitioners use configuration options mostly to enable software customization in software product lines (12; 11; 13). Figure 2.2 is an example of using configuration options (2). In this example, the value of `msg` is assigned at compile-time based on the values of `HELLO` and `BYE` configuration options.

2.1.4 Grey Literature

Grey literature is defined as *"... literature that is not formally published in sources, such as books or journal articles"* (25). Software practitioners may share their experiences as grey literature which can be considered a valuable resource for researchers (26) and other practitioners. Academic publications reflect the state-of-the-art, and grey literature provides insight into state-of-the-practice in any research area. In practical research areas such as software engineering, combining state-of-the-art and state-of-the-practice is important to provide valuable results (27). In the area of feature toggles, a large number of grey literature artifacts exist, but only a small number of peer-reviewed papers have been published. As we will discuss in Step Two of the methodology in Chapter 3, we use the quality assessment checklist of grey literature for software engineering provided (1) to evaluate the quality of our grey literature artifacts.

2.1.5 Metrics

Chidamber et al. (28) developed and empirically validated six metrics (CK metrics) for object-oriented (OO) design. These metrics can be used to measure the OO software development process improvement. The main focus of developed metrics is to measure the complexity of the design of classes. The six metrics are: 1) Weighted Methods Per Class (WMC); 2) Depth of Inheritance Tree (DIT); 3) Number of Children (NOC); 4) Coupling between object classes (CBO); 5) Response For a Class (UFC); and 6)Lack of Cohesion in Methods (LCOM). Some of our metrics in Chapter 4 overlap with CK metrics.

2.2 Related Work

In this section, we describe related work to this thesis.

2.2.1 Feature Toggles

Rahman et al. (29) performed a qualitative grey literature study and conducted follow-up inquiries to study continuous deployment practices. They reported 11 continuous deployment practices used by 19 software companies. Using feature toggles is one of these 11 practices used by 13 of the 19 companies. In addition, at the Continuous Deployment Summit (9) 2015, researchers and practitioners from 10 companies shared their best practices and challenges. Parnin et al. (9) disseminated 10 best practices from the Summit, including using feature toggles to implement Dark Launches[1].

To understand the drawbacks, strengths, and costs of using feature toggles in practice, Rahman et al. (6) performed a thematic analysis of videos and blog posts created by release engineers. They provided six themes found in analyzed videos and blog posts, such as technical debt and combinatorial feature testing. To identify feature toggle practices in Chapter 3, we used videos and blog posts from (6) and additional peer-reviewed papers and grey literature artifacts, including more videos and blog posts. Rahman et al. (6) also performed a quantitative analysis of feature toggle usage across 39 releases of Google Chrome from 2010 to 2015. They mined a spreadsheet used by Google developers for feature toggle maintenance. Release toggles should be short-lived toggles but Rahman et al. observed that 53% of the release toggles exist for more than 10 releases in Chrome. They classified

[1] *Dark launching* is a practice in which code is incrementally deployed into production but remains invisible to users (9).

unused but existing release toggles in the code as technical debt. Rahman et al. reported three purposes for using feature toggles: rapid release, trunk-based development, and A/B testing. They found that using feature toggles makes control over deployed functionality flexible, but can introduce technical debt and require additional maintenance effort by developers.

Rahman et al. (7) extracted four architectural representations of Google Chrome: 1) conceptual architecture; 2) concrete architecture; 3) browser reference architecture; and 4) feature toggle architecture. Using the extracted feature toggle architecture, developers can find out which feature affects which module and which module is affected by which feature. The goal of their study was to show how developers can get a new viewpoint into the feature architecture of the system using the extracted feature toggle architecture. Their result raise awareness of the impact of using feature toggles on the modular architecture of the system.

Ramanathan et al. (30) developed an automated code refactoring tool to delete code related to old and unused feature toggles in Uber repositories. Their tool analyzes the abstract syntax tree of the code, generates a diff (differential revision which contains code modifications, code reviewers, and summary of the change) on the GitHub repository for unused feature toggles, and assigns a task including the generated diff to the developer who creates the feature toggle to accept or reject the code changes. Over 1.5 years of using the tool, the tool generated diffs for 1381 feature toggles, and 65% of them were accepted by assigned developers without any changes.

Meinicke et al. (14) proposed a semi-automated approach to detect feature toggles in open-source repositories by analyzing the repositories' commit messages. They found 100 GitHub repositories that use feature toggles via keyword search in commits. Meinicke et al. analyzed some aspects of feature toggle usage in these identified repositories, such as the relationship between having short-lived toggles and having a toggle owner.

Hoyos et al. (31) analyzed the source code of 12 Python repositories that use feature toggles and surveyed 61 practitioners to shed light on the removal of feature toggles in practice. They found that 75% of feature toggles in analyzed Python repositories are removed within 49 weeks after creation. They also found that not all feature toggles are removed from code, and some long-lived toggles are not designed to live that long. Practitioners who participated in their survey reported removing feature toggles when the feature stabilized in the production, when they have scheduled clean-up audits, when the A/B test is done, or when they refactor the code.

Prutchi et al.(32) studied the effect of adopting feature toggles on the frequency and

complexity of branch merges, and a number of the defects and their fix time in 949 open source repositories. They observed a decrease in the merge effort and an increase in defect fix time. However, they did not confirm this increase is because of adopting feature toggles.

2.2.2 Configuration options

Sayagh et al. (4) aimed to understand the process required by practitioners to aggregate configuration options in a software system, the challenges they face, and the best practices they could follow. To achieve their goal, the authors did 14 interviews with software engineering experts, conducted a survey on Java software engineers, and did a literature review of academic papers in the area of configuration options. They identified 9 configuration management activities, 21 configuration challenges, and 24 expert recommendations. One of the reported challenges, the increasing complexity of the code by adding configuration options, is the same as the challenges of using feature toggles (11).

Meinicke et al. (33) analyzed 8 small- to medium-sized configurable systems' traces to identify interactions among configuration options on their control flow and data. Exponentially growing space for configuration options in these systems can affect the quality assurance of the systems. They showed that configuration interaction in these systems is lower than combinatorial complexity in theory.

Zhang et al. (15) studied 1,178 configuration-related commits of four open-source cloud system repositories. They analyzed the evolution of configuration design and implementation in these systems. Zhang et al.'s goal was to understand developers' practices to revise the design and the implementation of configurations from code changes in response to misconfigurations. They studied commits from four cloud system repositories over 2.5 years and focused on the revision of misconfigurations. They studied code changes to help reduce misconfigurations in cloud systems and developed a taxonomy for cloud systems' configuration design and implementation evolution.

Liebig et al. (34) analyzed 40 open-source software projects that use C preprocessors (cpp) to implement variable software. Using cpp is a popular approach to implementing configuration options. Liebig et al. introduce several metrics to measure cpp usage in terms of comprehension and refactoring, such as Lines of Feature Code (LOF) and Granularity (GRAN). Based on their results, Liebig et al. suggested alternative implementation techniques. Although some of our metrics overlap with Liebig et al.'s metrics, Liebig et al. focused on projects written in C. In Chapter 4, we do not filter projects based on programming languages. Our findings are language-independent.

To quantify the challenges of testing and debugging highly-configurable applications, Jin et al. (35) analyzed one industrial and two open-source applications. They found that configuration traceability is necessary, analysis tools need to cross the programming language barrier, and a way is needed to capture the active configuration when a system fails to reproduce and debug the failing test case.

Kim et al. (36) proposed SPLat, an approach to prune irrelevant configurations to reduce combinatorial combinations in testing configurable systems. This approach dynamically determines configurations during test execution by monitoring accesses to configuration variables. The authors experimentally showed that SPLat reduces the total test execution in many cases.

2.2.3 Software Configurations

Meinicke et al. (11) explored the differences and commonalities between feature toggles and configuration options by conducting nine semi-structured interviews with feature toggle experts. During the interviews, the authors asked practitioners about the existing literature on feature toggles. Then, they discussed with the interviewees the configuration options topics and asked them if they saw common challenges and solutions. The researcher found that although feature toggles and configuration options are similar concepts, they have distinguishing characteristics and requirements. The goal of the usage and challenges of each of the techniques are distinct. The researcher identified 10 themes for the differences, such as their users and their lifetime. Feature toggles are used by developers, but configuration options are used by end-users. The feature toggles will be removed from code ideally, but configuration options can exist permanently.

To provide software configuration terminology for research studies and help practitioners to find possible challenges, Siegmund et al. (12) derived a model (MSC) consisting of eight dimensions for software configuration with 47 values in total. They used a mixed-methods approach. In the first step, they interviewed 11 practitioners, such as developers, team leads, and senior software engineers. From the collected data from interviews, they created an initial model of configuration. Their initial model includes seven dimensions with 32 values. In the second step, they analyzed two related academic publications to their study to validate their results and complete their model. As a result, they found one additional dimension and 15 new values. In the last step, to verify the applicability of their developed model and find gaps in the area, they applied their model to 16 academic publications that involve configuration options. Some of their dimensions are overlapped with

themes in Meinicke et al. (11).

The two discussed related work in this subsection (11; 12) are the basis of our studies in Chapters 5 and 6.

2.2.4 Frameworks for Structuring Families of Research

Williams et al. (37; 38) proposed an evaluation framework for researchers and practitioners to express concretely the Extreme Programming (XP) practices that the organization has chosen to adopt and modify, as well as the outcomes of those practices. Williams et al. showed the necessity of using this framework to enable a meta-analysis for a combining family of case studies on XP, and categorize empirical studies into a body of knowledge.

Morrison (39) proposed a Security Practice Evaluation Framework (SP-EF) to collect software development context factors, practice adherence, and security outcomes. They showed using SP-EF, researchers and practitioners can compare security practices across publications and projects and evaluate security practices.

Runeson et al. (40) explained that using a theoretical framework to design case study research in software engineering makes the context of the research clear, helps researchers to conduct the study, and build upon the results.

A family of research can benefit from a framework or common terminology to enable meta-analysis. Hence, following fellow researchers (37; 38; 39; 40), in Chapters 5 and 6, we propose an extended model of software configuration, guidelines for researchers to document context variable in their software configuration research studies, and do a meta-analysis on software configuration family of research.

CHAPTER

3

SOFTWARE DEVELOPMENT WITH FEATURE TOGGLES

3.1 Motivation

In 2012, developers in Knight Capital Group, an American global financial services firm, updated their automated, high-speed, algorithmic router which inadvertently repurposed a feature toggle, activating functionality which had been unused for 8 years. Within 2 minutes, developers realized the deployed code behaved incorrectly but took 45 minutes to stop the system. During that time, Knight Capital lost nearly 400 million dollars, which caused the group to go bankrupt (41). As illustrated, *using feature toggles without following good practices can be detrimental to an organization.*

The goal of this research study is to aid software practitioners in the use of practices to support software development with feature toggles through an empirical study of feature toggle practice usage by practitioners. Software practitioners prefer to learn through the experiences of other software practitioners (18). As such, our study obtains practice usage from practitioners.

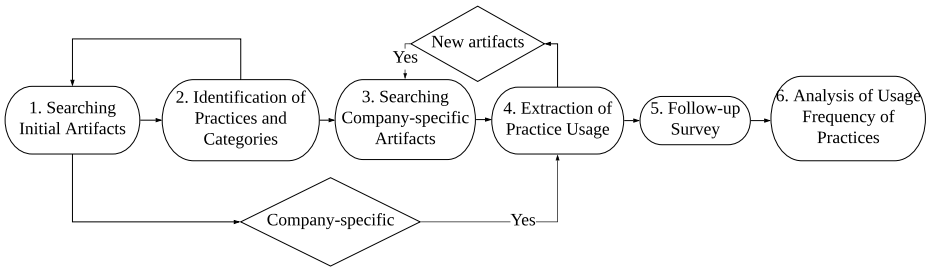


Figure 3.1: The research methodology.

We state the following research question with two sub-research questions:

RQ1 How do software practitioners use feature toggles in industry?

RQ1.1 (Identification): What are the feature toggle practices that software practitioners use?

RQ1.2 (Frequency): How frequently are feature toggle practices used?

3.2 Research Methodology

We describe the steps of our methodology to answer the research questions. Our methodology has six steps, as shown in Figure 3.1. We started by searching for an initial set of peer-reviewed papers and artifacts from the grey literature related to our study scope in Step One. In Step Two, we identified and categorized practices found in this literature, analyze the quality of grey literature artifacts, and calculate the level of confidence for identified practices. In Steps Three and Four, we iteratively searched for grey literature related to specific companies and extract the usage of identified practices. Then, we sent a survey to practitioners in Step Five. Finally, in Step Six, we analyzed the usage frequency of practices and compare practices, if possible. Each of these steps from Figure 3.1 will be explained in detail in the following sub-sections.

3.2.1 Step One: Searching Initial Artifacts

The first step in our research methodology in Figure 3.1 is to use a keyword search in the Google search engine to identify grey literature and in Google Scholar to find peer-reviewed papers. We used the following search terms: ‘feature toggle’; ‘feature flag’; ‘feature switch’;

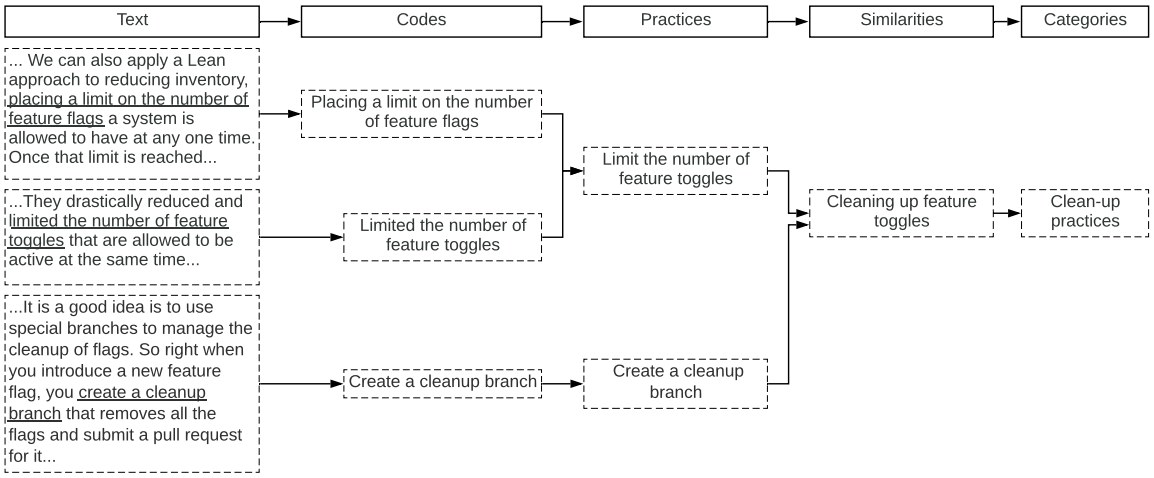


Figure 3.2: An example of using the open coding technique.

'feature flipper'; and 'feature bit'. These search terms were obtained from Fowler's blog post (8).

Selecting the related peer-reviewed papers and grey literature artifacts is done by two researchers. For grey literature, reviewing all the results of each search in the Google search engine was infeasible. The most relevant links are provided earlier in the Google search engine. We reviewed the first 10 pages of the results of each search to select the most relevant links. To determine if a link is related to the scope of our research, we read the article by looking for the search term and read 2–3 sentences before and after the search term in the text. For videos, we kept all the links and analyzed them in Step Two of the methodology. In the relevant grey literature artifacts, we used a snowballing approach (42). We clicked on links and the references to other feature toggle resources found in the artifact, and we read the papers, articles, or watched the videos. For peer-reviewed papers, we checked the titles of search results and kept those that were related to our topic. We used the snowballing approach for peer-reviewed papers. We checked the references and selected the related ones. *In the rest of the chapter, we use the term "artifacts" to refer to the set of "peer-reviewed papers and grey literature artifacts".* The time of the publishing of all collected artifacts is before June 2019. Some collected artifacts in Step One were company-specific artifacts that were often written by a release manager or developer, referencing feature toggle usage at a specified company. We used these company-specific artifacts in Step Two and Step Four.

3.2.2 Step Two: Identification of Practices and Categories

The grey literature and peer-reviewed papers found in Step One were used to identify feature toggle practices. We analyzed the artifacts using an open coding technique, a technique to analyze textual data by coding (i.e. labeling) concepts and identifying categories based on the similarity and dissimilarity of codes (43). First, we took notes from the videos. Then, we coded the suggested recommendations, experiences, and implementation details about using feature toggles mentioned by practitioners in the textual artifacts and the notes of videos. The coding of the artifacts was done by two researchers.

After the identification of practices, we observed similarities and dissimilarities between practices. We put practices with similarities into one category based on an open coding technique and found four categories. We give an example of using open coding with a sample of our data in Figure 3.2. In this figure, three paragraphs from three artifacts are shown and codes are assigned to them. The codes of the two first paragraphs pointed to the same concept so we grouped them as "Limit the number of feature toggles". The last code is changed to "Create a cleanup branch" practice. The similarity between these two extracted practices is pointing to cleaning up feature toggles, so the two practices are grouped as "Clean-up practices". The result of this step is the answer to the first sub-research question (RQ1.1-Identification). Step One and Step Two of the methodology were done iteratively. The snowball approach was terminated when we did not identify any new practices and did not find any new artifacts.

After identification of practices from artifacts, we specified a "Level of confidence" for each practice which was used to quantify our confidence in the quality, correctness, and importance of the identified practices. We used the quality assessment checklist of grey literature for software engineering provided in (1) and shown in Table 3.1. Following their example, each of the 20 questions is assigned a score of 1, 0.5, or 0, so the highest score would be 20. We assigned a score of 20 to peer-reviewed papers.

Table 3.1: The quality assessment checklist of grey literature for software engineering adapted from (1).

Criteria	Questions	Notes
Authority of the producer	Is the publishing organization reputable?	The authorship is attributed to an organization.
	Is an individual author associated with a reputable	The authorship is attributed to an individual author(s).

Table 3.1 (continued).

	Has the author published other works in the field?	Having other grey literature in the area.
	Does the author have expertise in the area?	Having experience in the area.
Methodology	Does the source have a clearly stated aim?	Having clear related subject.
	Does the source have a stated methodology?	Having a structured flow for discussion.
	Is the source supported by authoritative, contemporary references?	Having any references.
	Are any limits clearly stated?	Pointing to at least one related limitation.
	Does the work cover a specific question?	Covering the concept of feature toggles.
	Does the work refer to a particular population or case?	Related to feature toggles.
Objectivity	Does the work seem to be balanced in the presentation?	Discussing the subject from different views.
	Is the statement in the sources as objective as possible?	Including enough evidence.
	Is there vested interest?	Unbiased to any organization's tool.
	Are the conclusions supported by the data?	Having a reasonable conclusion.
Date	Does the item have a clearly stated date?	Including date.
Position w.r.t. related sources	Have key-related grey literature or formal sources been linked to/discussed?	Having at least one of the key grey artifacts of the area.
Novelty	Does it enrich or add something unique to the research?	Including any valuable data for the research.

Table 3.1 (continued).

	Does it strengthen or refute a current position?	Supporting any valuable data for the research.
Impact	Normalize the Number of backlinks, Number of social media shares into a single metric	For backlinks: `https://www.seoreviewtools.com/valuable-backlinks-checker/` , For social media share: `http://www.sharedcount.com/`
Outlet type	• 1st tier grey literature (measure=1): High credibility: Books, magazines, theses, government reports, white papers • 2nd tier grey literature(measure=0.5): Moderate credibility: Annual reports, news articles, presentations, videos, Q/A sites (such as StackOverflow), Wiki articles • 3rd tier grey literature(measure=0): Low credibility: Blogs, emails, tweets	Assign score based on the type of artifact.

To specify the level of confidence, we combined two factors for each practice as shown in Table 3.2: (1) The average quality of the artifacts that the practice is mentioned in; and (2) The number of artifacts that point to the practice. We defined four levels of confidence: High, Moderate-High quantity, Moderate-High quality, and Low. In Table 3.2, the range for the average quality score of artifacts is between 0 and 20. We divided this range into 2 equal width ranges. The number of artifacts analyzed in this step is 66, and we divide the range of 0 to 66 into 2 equal width ranges. The numbers in front of each level of confidence are the number of practices that fell into the category. Moderate-High quantity is used when the practice is mentioned in more than half of the artifacts but the average quality of the artifacts is lower than the selected threshold which is 10. Moderate-High quality is used

Table 3.2: The level of confidence. The number in parenthesis of each level is the number of practices that fell into the category.

Number of artifacts	Average quality score of artifacts	
	[0,10)	[10,20]
[0,33)	Low (0)	Moderate-High quality (16)
[33,66]	Moderate-High quantity (0)	High (1)

when the practice is mentioned in less than half of the artifacts but the average quality of the artifacts is more than the threshold. We will refer to this table in Section 3.3.1.

3.2.3 Step Three: Searching Company-specific Grey Literature

Some artifacts collected in Step One were company-specific artifacts. Additionally, some artifacts contained a list of companies that use feature toggles. From these artifacts, we obtained a list of companies that use feature toggles in their development cycle. Additional searches were conducted to collect more artifacts related to feature toggles from these specific companies. We used the search strings in the following format: "[company name] [feature toggle term]" where company name represents the name of the company; and feature toggle term is a search term for "feature toggle," as defined in Step One. For each combination of company name and feature toggle term, a search string was applied to collect as many artifacts as possible. These strings were searched by using both the Google search engine and search feature found within a company's blog. We looked at the first 10 pages of the Google search engine result and all of the results in the company's blogs. If a company uses a feature toggle management system named by an artifact, we also used that system's name instead of the "feature toggle term" in a search string. For example, Facebook uses Gatekeeper for feature toggle management (44). We used Gatekeeper instead of "feature toggle term" as well as search terms for feature toggle in the search for Facebook.

3.2.4 Step Four: Extraction of Practice Usage from Company-specific Artifacts

We analyzed the company-specific grey literature artifacts collected in Step One and Step Three to determine which practices identified in Step Two are used by the companies, as mentioned in the artifacts. If a practice was not clearly mentioned, a second person

analyzed the artifact, and then we made a decision if the company used the practice or not.

Step Three and Step Four in Figure 3.1 were performed iteratively and repeatedly if new artifacts for a company were found in Step Four.

3.2.5 Step Five: Conducting Survey

After extracting practice usage by the companies, we observed that our results were not complete. For instance, some of the identified feature toggle practices were not mentioned in any of the company-specific artifacts. Besides, we were interested to know about the status of usage of feature toggles in the industry. So, we conducted a survey to obtain more information about feature toggle practice usage.

Contact information of company employees was gathered by collecting social media accounts and email addresses of named individuals associated with company-specific artifacts found in Steps One and Three. We also found contact information of managers/developers in companies that we knew were using feature toggles based upon Step One, even though we did not find company-specific artifacts for them in Step Three. We requested each practitioner to complete the survey. We contacted the practitioners by email where email was available and by social media if email addresses were not found. We sent the survey to 45 practitioners and got 20 responses for a response rate to the survey of 44%.

The survey has 11 questions and is presented in the Appendix A. On average, each practitioner needed approximately 5 minutes to answer all questions. We used Likert scale options (45) for the 12 practices for which Likert scale options can be used. We provided five options in the survey for each practice to specify how much the survey respondents use the practice: Always, Mostly, About half of the time, Rarely, and Never. For the remaining 5 practices, we provided practice-specific answer options.

3.2.6 Step Six: Analysis of Usage Frequency of Practices

We analyzed the information from Step Four (analyzing company-specific artifacts) and Step Five (survey) to find the frequency of usage of each identified practice in the industry to answer RQ1.2. We integrated the result of Step Four and Step Five and report the frequency of usage of feature toggle practices. In addition, we reviewed all the artifacts (including initial artifacts and company-specific artifacts) and record any comparison made between practices in artifacts in this step.

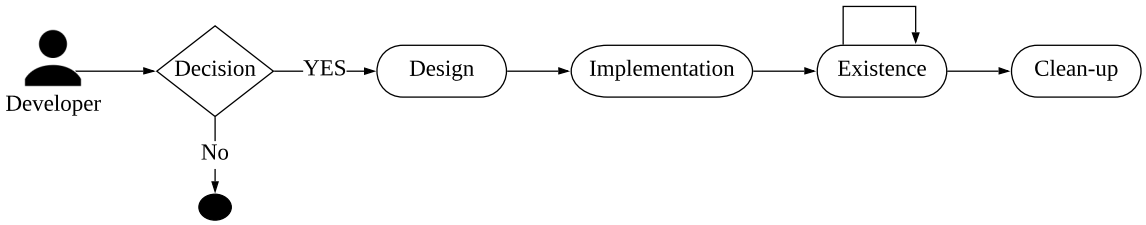


Figure 3.3: The lifecycle of a feature toggle, adapted from (3).

3.3 Results

Based on (3), we propose the lifecycle of a feature toggle as shown in Figure 3.3. The first phase is **Decision** when the development team decides if the usage of a feature toggle is necessary for their situation. When the development team decides to use a feature toggle, the second phase is **Design** in which the details of the feature toggle are determined, such as the type of the toggle, the possible values of the toggle, and/or the name of the toggle. The third phase is **Implementation** which includes adding the designed feature toggle to the code. The fourth phase is **Existence**, in which the toggle exists in the code and/or is updated. The fifth phase is **Clean-up** in which the toggle is removed from the code. For each identified practice, in Section 3.3.1, we will specify the lifecycle phases covered by the practice.

In the rest of this section, we present the results of the research methodology. Section 3.3.1 provides the answer to RQ1.1-Identification, and Section 3.3.2 provides the answer to RQ1.2-Frequency.

3.3.1 Feature Toggles Practices

We found 66 artifacts including 10 peer-reviewed papers, 41 blog posts and online articles, and 15 videos in Step One. Of the 10 peer-reviewed papers, 9 papers (6; 44; 46; 47; 48; 49; 50; 51; 7) identify practices and are discussed in this subsection. The remaining paper (29), as discussed in the Related Work (Section 2.2), listed 11 companies that use feature toggles in their development cycle (29). We used this list of companies in Step Three of our methodology to find company-specific artifacts that are used in Section 3.3.2. From these 66 artifacts, we identified and categorized 17 practices in Step Two. We found four categories of practices: Management practices (6), Initialization practices (3), Implementation practices

(3), and Clean-up practices (5). We describe the 17 practices in their categories in the rest of this section.

The level of confidence of each identified practice is calculated as we described in Step Two of the methodology in Section 3.2. The range of scores for quality assessment of grey literature artifacts is from 9.5 to 17. The number of practices that fell into each level of confidence is shown in Table 3.2. "Use management systems" is the only practice in **High** level, all the rest practices are in **Moderate-High quality** level.

Some of the practices are related to other practices. For example, following one of the practices may help following another practice. We determined the relation between practices based on the effects of using them. Related practices are summarized in Table 3.3.

For each practice, we explain the practice using the following structure:

1. Description: The explanation of the practice.
2. Goal: The goal of following the practice.
3. Examples: The example(s) from the 38 companies listed in Table 3.5 and practitioner's experiences about following or not following the practice.
4. Covered phases in the lifecycle: The phase(s) in which the practice would be used, as shown in Figure 3.3.
5. Comparative practices: The list of practices that could be compared with the practice (because of the same goal). Practices in the Clean-up category are the only practices that could be compared to each other. So, practices in other categories do not have this bullet in their structure.
6. Generalizability: A categorization of whether the practice is generally to software engineering (SE); common with configuration options (C) as identified in (4) and discussed in Section 2.2; or feature toggle-specific (F).

Management Practices (6)

Management practices are practices that are performed by development team members to make decisions about how to use feature toggles.

Mg1: Use management systems:

Description: Management systems help companies to create, use, and change the value of feature toggles. Using feature toggle management systems helps to overcome technical debt and manage the added complexity (52). Feature toggle management systems can have a dashboard that helps team members to see the list of feature toggles and their

current values. Team members can add new feature toggles or change the values of the toggles if they have permission. Management systems are connected to the code, and the changes impact the running system immediately. Management systems can be open-source or closed-source. Organizations may create their own feature toggle management system.

Goal: To manage creating and updating feature toggles in a centralized system.

Examples: Facebook uses the Gatekeeper toggle management system (44). Alternatively, companies can use third-party management systems, such as LaunchDarkly[1] and Split[2]. As an example, Behalf[3] and CircleCI[4] use the LaunchDarkly feature toggle management system. Envoy[5] uses Split's feature toggle management system.

Covered phases in lifecycle: Design, Implementation, Existence, and Clean-up.

Generalizability: (C). Sayagh et al. (4) recommend the adoption of existing configuration frameworks.

Mg2: Document feature toggle's metadata:

Description: Through the documentation of feature toggles, practitioners record the feature toggle's information e.g. the owner of the feature toggle; the current status (to remove untriaged, keep, removed); the time of its creation; and any notes.

Goal: To enable practitioners to have access to the feature toggles' metadata at any time.

Examples: Google has a spreadsheet with a list of feature toggles, the owner, toggles' status, and notes about toggles that are used in the Chromium project (53). However, this spreadsheet has only changed three times in 2018 and three times in 2019. The developers may have moved on to use a new tool for documenting feature toggles' metadata.

Covered phases in lifecycle: Implementation, Existence, and Clean-up.

Generalizability: (SE(54)/C). Sayagh et al. (4) listed "Comprehension of Options and Values" and "Configuration Knowledge Sharing" as two configuration option activities. The concept behind these two activities is similar to documenting the feature toggle's

[1] `https://launchdarkly.com/`
[2] `https://www.split.io/`
[3] `https://www.behalf.com/`
[4] `https://circleci.com/`
[5] `https://envoy.com/`

metadata.

Mg3: Log changes:

Description: Tracking changes that are made on feature toggles. By logging, the information of who changes which toggle and when is recorded (55).

Goal: To document traceability of actions for creating, updating, and deleting toggles and their values.

Examples: Split's feature toggle management system has the ability to log changes of the feature toggles (56).

Covered phases in lifecycle: Implementation, Existence, and Clean-up.

Generalizability: (SE (54)/C). In (4) having "Right Granularity of Execution Logs" is one of the recommendations, so logging changes are recommended for configuration options.

Mg4: Determine applicability of feature toggle:

Description: Before the design and implementation of a feature toggle, the development team should determine if a feature toggle should be used. Using feature toggles adds more decision points to the code which adds more complexity to the code and requires attention to remove toggles when the initial use is completed.

Goal: To make an explicit decision on the creation of a toggle that may reduce the number of feature toggles in a codebase.

Examples: Different companies have different approaches to making decisions. For example, all new features in GoPro have feature toggles[6]. However, practitioners in Finn.no, the largest online marketplace in Norway, avoid using feature toggles if they do not need the toggle (57).

Covered phases in lifecycle: Decision.

Generalizability: (F).

Mg5: Give access to team members:

[6] https://bit.ly/2ISi1ye

Description: Through this practice, permission to change values of feature toggles is granted to team members in addition to developers using the feature toggle management system.

Goal: To prevent feature toggle management bottleneck.

Examples: If all team members, such as Q&A team members, have access to feature toggles, they can change a toggle status in case of a problem (55). For instance, Instagram gives access to its feature management system to the product managers and sales team (58).

Covered phases in lifecycle: Design, Implementation, and Existence.

Generalizability: (SE).

Mg6: Group the feature toggles:

Description: Grouping similar feature toggles.

Goal: To enable the assignment of access to groups of feature toggles to teams or team members (55); to simplify management of dependent toggles (3); or to turn on or off feature toggles related to a part of the system at the same time.

Examples: Practitioners in GoPro have a two-level toggle hierarchy: simple feature toggles and higher-level feature toggles(59).

Covered phases in lifecycle: Design, Implementation, and Existence.

Generalizability: (SE).

Initialization Practices (3)

Initialization practices are used to make decisions about the design of the feature toggle before its creation.

I1: Set up the default values:

Description: Default values for each feature toggle are set in case assigned values are not found or do not exist.

Goal: To mitigate unwanted behavior of the feature toggle.

Examples: At Lyris, feature toggles without values are automatically turned off (60).

Covered phases in lifecycle: Design and Implementation.

Generalizability: (SE/C). Sayagh et al. (4) mentioned that the selection of the "right" default value for configuration options is a challenge.

I2: Use naming convention:

Description: Having predefined naming rules for feature toggles.

Goal: To establish naming conventions, particularly to make the intention of the toggle self-documented.

Examples: Having the naming convention has several benefits. First, understanding the purpose of using the toggle is useful (60), i.e. if the owner of the code is changed, the new owner can understand the usage of the toggle easily if the name of the toggle reflects its usage. An example of Lyris toggles is "ct.enable_flex_cache_inspector' and the purpose of using the toggle is clear based on its name (60). Second, the use of a naming convention reduces the likelihood of multiple toggles with the same names even by different teams by following naming conventions (55). Third, adding the type of the toggle as a prefix in its name can help with the management of the toggles (61). For instance, if the feature toggle is a short-lived toggle, like release toggles, the developer will get a signal from the name of the toggle that the first intention of using the toggle was a short-term use and will plan to remove it. For instance, In InVision, long-lived toggles have the "OPERATIONS-" prefix (62). Developers in this company also add the JIRA ticket number to the name of the feature toggle to make the purpose of using the toggle and the responsible team to remove the toggle clear. If a "RAIN-123-release-the-kraken" is the name of the toggle, it is clear that the toggle is related to JIRA ticket RAIN-123, and the responsible team for clean-up the feature toggle is the Rainbow team (63).

Covered phases in lifecycle: Design and Implementation.

Generalizability: (SE(64)/C). O(4) one of the challenges of using configuration options is "meaningless Option Names". The authors listed the "Explicit Option Naming Convention" as a recommendation to overcome the challenge which is similar to use naming convention practice.

I3: Determine the type of the toggle:

Description: With this practice, the type of toggle is specified using the toggle types mentioned in Section 2.1. The implementation and management of each type of the five

toggle types are different.

Goal: To aid in quality management of a toggle's implementation and to enable the plan to remove the toggle on time based on the type of the toggle.

Examples: The author in (65) points to naming short-lived toggles with the prefix of "temp-" in their name. The identification of short-lived toggles can be useful in limiting the number of toggles (66).

Covered phases in lifecycle: Design and Implementation.

Generalizability: (F).

Implementation Practices (3)

Implementation practices are related to implementation details of feature toggles.

Im1: Type of assigned values:

Description: Companies use three different ways to assign values to toggles. One way is to assign a string to feature toggles. The second way to assign values is to assign Boolean values to feature toggles. When the value is true, the toggle is enabled (67). The third way is to assign multivariate values, such as when the toggle captures user experiences.

Goal: To help practitioners to choose an appropriate type of the values for feature toggles in their system.

Examples: For instance, one of the feature toggles in Google Chrome project is kDisable-Flash3d[] = "disable-flash-3d". If the value of the feature toggle is set then the toggle is enabled (6). As another example, Rollout[7] provides multivariate toggles, for instance, a toggle can accept "Red", "Blue", and "Yellow" as its value.

Covered phases in lifecycle: Design and Implementation.

Generalizability: (C). Sayagh et al. (4) provide the "Using Simple Option Types" recommendation which mentioned string and Boolean types for configuration options. This recommendation has partially overlapped with the current practice.

Im2: Ways of accessing the values:

[7]https://rollout.io/

Description: We identified three ways development teams access the values of feature toggles. First, the feature toggles could be primitive variables, hard-coded into the program. Second, toggles could be objects and the object has a method to determine the value of the toggle (e.g. myToggle.isActive()). Third, toggles could be accessed through a manager object. Managers map key/value pairs to return the value.

Goal: To help practitioners to choose an appropriate way of accessing the values in their system.

Examples: Figure 2.1 is an example of direct access to the values. We found implemented libraries in GitHub which use a method from toggle object, such as rollout[8]. LaunchDarkly is an example of using manager objects to access feature toggle values.

Based on the experience of the practitioner in (68), having a class of toggles and checking the value of the toggle using isEnable() function of the class is better than checking the primitive variable of a string name of the toggle. Having feature toggle objects helps to refactor toggles same as the other parts of the code and find every usage of it easily. The practitioner (69) also points to the fact that using objects of feature toggles is better than strings because of getting compile error on all places a feature toggle is used after removing the toggle. The same comparison is mentioned in (70): "Toggles should be real things (objects) not just a loosely typed string. This helps with removing the toggle after use: 1) Can perform a 'find uses' of the Toggle class to see where it's used, and 2) Can just delete the Toggle class and see where build fails."

Covered phases in lifecycle: Design and Implementation.

Generalizability: (SE (64)).

Im3: Store type:

Description: The list of feature toggles and their values can be stored in one of two ways: file storage and database storage. In file storage, the values of feature toggles are stored in one or multiple configuration files(6). In database storage, the values of feature toggles are stored in databases, such as Redis (67) or SQL (70). In addition, some companies "use a third-party service" to fetch values of the feature toggle. If they use a feature management system, they fetch the values of feature toggles from the management system.

[8]https://github.com/fetlife/rollout

Goals: To store the values of feature toggle in an appropriate way, based on the advantages and disadvantages of each store type.

Examples: Based on the article (22), when practitioners use configuration files to save the feature toggle values, they may need to redeploy the application after each value update to get the reflection of the updated value. In addition, when a system is on a large scale, it is hard to manage feature toggles using configuration files and it is hard to make sure the consistency of configuration files on different servers. So the practitioner recommends using some sort of database to store the value of feature toggles.

The practitioner in (67) is also made the same comparison in his article: " A more dynamic approach is to store the feature configuration in either an ephemeral or permanent storage, like Redis or your database, respectively. Assuming your code continuously checks feature flags at runtime, all it takes is changing a value in the central configuration service and it can have an immediate effect on the running application without requiring a restart."

The article (71) points to the same comparison: " Release and Experiment toggles are likely to be set at deployment time, so from a running application perspective they are *static* settings ... However, Ops and Permission toggles are *dynamic* and need to be configurable at run time, so you might want to store them in a database of some sort." The same concept is mentioned in (3) and (69) as well.

Dropbox uses configuration files and database together(72). They explain the reason for using both ways. Because of having a large number of production servers, they prefer using a database instead of using configuration files. However, this may create a huge number of fetches against the database and the database will be the single point of the failure even if they have a caching system. So, they come up with a combination of both ways. A JSON file is shared between all the production servers and contains the value of feature toggles. If this JSON file is not accessible for any reason, the feature toggle management system has the ability to access the database directly. This example shows the advantages and disadvantages of using each way of storing the values of feature toggles.

Covered phases in lifecycle: Design and Implementation.

Generalizability: (C). "Managing Storage Medium" activity form (4) is similar to this practice. Moreover, one of the challenges in (4) is "Storage media Impact Performance" which is really similar to the practitioners' experiences we provided for this practice.

Clean-up Practices (5)

Following the clean-up practices helps practitioners to remove their feature toggles on time and manage the complexity of using feature toggles.

C1: Add expiration date: This practice is followed using one of the following three processes:

- C1.1: Time bombs:

  Description: If the toggle exists after its expiration date, a test fails or the application does not start, which causes a developer to remove the toggle (22), (73), (66). The expiration date is the latest possible date in which the developers should remove the toggle from the code. Using this practice forces practitioner to remove a toggle by the determined expiration date.

  Goal: To remove unused toggles.

  Examples: We did not find any specific examples in company-specific artifacts. However, the practitioner in (66) says: "(Time bomb) is very extreme and I wouldn't recommend doing (it). I think a lot of organizations would not be comfortable with doing that, it does force a lot of other things to be good."

  Covered phases in lifecycle: Design, Implementation, and Clean-up. The expiration date is identified during the design phase, the time bomb is added in the Implementation phase and the feature toggle is removed in the Clean-up phase.

  Comparative practices: All the practices in this category (C1-C5).

  Generalizability: (F).

- C1.2: Automatic reminders:

  Description: Add automatic reminders to remind developers of the deadline for removing feature toggles (73). Using this practice helps practitioners to remember to remove a toggle by the determined deadline.

  Goal: To remove unused toggles.

  Examples: Slack has an archival system. When developers want to add a new feature toggle, they have to specify the date they plan to delete the toggle. If the toggle is not deleted by the specified date, the developer will get an alert[9].

[9] https://bit.ly/2W4hQUk

Covered phases in lifecycle: Design, Implementation, and Clean-up.

Comparative practices: All the practices in this category (C1-C5).

Generalizability: (SE).

- C1.3: Use cards/tasks/stories for removing toggles:

  Description: Add tasks/stories/cards for removing toggles to a Kanban board (or any other tool that the team uses) (73) or to the developer's task backlog (22), (66). Using this practice reminds practitioners of the task of removing a toggle at the expiration date when the purpose of using the toggle is done.

  Goal: To remove unused toggles.

  Examples: Developers at Lyris create user stories for removing toggles (60). However, the practitioner in (66) says about his experience of using this practice:" I think it can help but it's kind of the bare minimum, .. I have spent a lot of time with clients where the clean-up ticket is just at the very top of the next Sprint's backlog for like six months. It's always like yeah we really should do that the next week and it will be done next week, but it's always next week." Practitioner in (63) has the same experience with this technique: " Teams are supposed to create additional "clean-up" tasks in JIRA for their feature flags such that we don't lose track of them. The reality, however, is far less sanitary. Our feature flags tend to pile up and we have to occasionally have a "purge" of flags that no longer seem relevant."

  Covered phases in lifecycle: Design, Implementation, and Clean-up.

  Comparative practices: All the practices in this category (C1-C5).

  Generalizability: (SE (74)).

C2: Track unused toggles:

Description: With this practice, dead code and unused feature toggles are removed. Based on the logging system or using documentation, the status of toggles could be monitored. Developers can use this data to find when the toggle is safe to remove (55). When a toggle is always on or always off, it should be removed.

Goal: To remove unused toggles.

Examples: DropBox has a static analyzer tool with a service specifically for feature toggles. The static analyzer tool sends emails to feature toggle owners about removing the toggles which are not in use anymore (72).

Covered phases in lifecycle: Implementation and Clean-up.

Comparative practices: All the practices in this category (C1-C5).

Generalizability: (F).

C3: Limit the number of feature toggles:

Description: Using this practice the number of alive feature toggles at a time is limited to control the number of toggles. An alive feature toggle is a toggle that exists in the code whether it is on or off. By this limitation, practitioners have to remove an unused toggle to be able to add a new toggle if the number of existing toggles meets the limitation (22), (66), (47).

Goal: To remove unused toggles.

Examples: We did not find any specific example of using this practice in company-specific artifacts. The practitioner in (66) says this practice is his favorite practice for removing feature toggles.

Covered phases in lifecycle: Implementation and Clean-up.

Comparative practices: All the practices in this category (C1-C5).

Generalizability: (F). Sayagh et al. (4) point to minimizing the number of configuration options in the system as one of the recommendations. This recommendation partially covers the practice of limiting the number of feature toggles. In limiting the feature toggles, developers have to remove a toggle if the limitation is reached but removing is not mentioned in the recommendation of configuration options. So, we specify the practice as a feature toggle specific practice.

C4: Create a cleanup branch:

Description: This is the practice of creating a branch to delete the toggle and submitting a pull request for the branch at the same time as adding a new feature toggle (61), (65). Using this practice prevents forgetting the deletion of the feature toggle.

Goal: To remove unused toggles.

Examples: We did not find any example of using this practice in company-specific artifacts. The author of the (65) says that this practice works pretty well in their team: “The advantage to managing cleanup this way is that you do the work to remove the flag when

all of the context is fresh in your mind. At this point, you know all the pieces that get touched by the change, and it is easier to be sure you don't forget something .. This is certainly not the only way to handle this issue, but it seems to work pretty well for our team.".

Covered phases in lifecycle: Implementation and Clean-up.

Comparative practices: All the practices in this category (C1-C5).

Generalizability: (SE).

C5: Change a feature toggle to a configuration setting:

Description: This is the practice of keeping feature toggles in the code with changed functionality. The feature toggle can be changed to admin or user configuration settings. This technique is used when the development team decides to keep more than one variant of the feature toggle in the code.

Goal: To remove unused feature toggles.

Examples: Suppose a feature toggle is used for running experiments to see which color is better for the "buy" button in an e-commerce application. The experimental results show that the users are happiest when they can control the color of the button. Instead of deleting the feature toggle, it will be changed to a user configuration setting (75).

Covered phases in lifecycle: Implementation and Clean-up (C1-C5).

Comparative practices: All the practices in this category.

Generalizability: (F).

3.3.2 Usage of Practices in Industry

In Step One, 26 artifacts were company-specific artifacts. In Step Three and Step Four, we found 43 additional company-specific artifacts. In total, 69 company-specific artifacts from 38 companies were collected. The overlap between initial artifacts and company-specific artifacts is shown in Figure 3.4. In Step Four, we analyzed these 69 company-specific artifacts to find which companies use the identified practices. The practices used by each company are shown in Table 3.5. This table is useful for practitioners because software practitioners prefer to learn through the experiences of other software practitioners (18), as we mentioned before.

Table 3.3: Related practices.

		Management						Initialization			Implementation			Clean-up				
		Use management systems	Document feature toggle's metadata	Log changes	Determine applicability of feature toggle	Give access to team members	Group the feature toggles	Set up the default values	Use naming convention	Determine the type of the toggle	Type of assigned values	Ways of accessing the values	Store type	Add expiration date	Track unused toggles	Limit the number of feature toggles	Create a cleanup branch	Change a feature toggle to a configuration setting
Management	Use management systems	–	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		
	Document feature toggle's metadata	✓	–															
	Log changes	✓		–											✓			
	Determine applicability of feature toggle				–													
	Give access to team members	✓				–												
	Group the feature toggles	✓					–											
Initialization	Set up the default values	✓						–										
	Use naming convention				✓				–									
	Determine the type of the toggle	✓								–						✓		
Implementation	Type of assigned values	✓									–							
	Ways of accessing the values	✓										–						
	Store type	✓											–					
Clean-up	Add expiration date	✓												–				
	Track unused toggles	✓	✓	✓											–			
	Limit the number of feature toggles									✓						–		
	Create a cleanup branch																–	
	Change a feature toggle to a configuration setting																	–

Table 3.4: Feature toggle practices and their usage in the industry

Category (mean of frequencies based on artifacts, mean of frequencies based on survey)	Practice	Frequency from artifacts (38 companies)	Frequency from the survey (20 companies)
Management (42%, 63%)	Use management systems	32 (84%)	20 (100%)
	Document feature toggle's metadata	25 (66%)	7 (35%)
	Log changes	21 (55%)	12 (60%)
	Determine applicability of feature toggle	8 (21%)	16 (80%)
	Give access to team members	7 (18%)	14 (70%)
	Group the feature toggles	2 (5%)	7 (35%)
Initialization (25%, 72%)	Set up the default values	22 (58%)	17 (85%)
	Use naming convention	5 (13%)	14 (70%)
	Determine the type of the toggle	1 (3%)	12 (60%)
Implementation (66%, 100%)	Type of assigned values (string, boolean, multivariate, more than one)	32 (1 (3%), 7 (18%), 5 (13%), 19 (50%))	20 (1 (5%), 6 (30%), 2 (10%), 11 (55%))
	Ways of accessing the values (primitive variable, objects, managers, more than one)	28 (0 (0%), 0 (0%), 28 (74%), 0 (0%))	20 (3 (15%), 5 (25%), 3 (15%), 9 (45%))
	Store type (file, database, both, third party service)	15 (9 (24%), 4 (11%), 2 (5%), -)	20 (3 (15%), 6 (30%), 6 (30%), 4 (20%))
Clean-up (3%, 39%)	Add expiration date (Time bombs, Automatic reminders, Use cards/tasks/stories for removing toggles)	6 (0 (0%), 1 (3%), 5 (13%))	14 (1 (5%), 4 (20%), 9 (45%))
	Track unused toggles	1 (2%)	9 (45%)
	Limit the number of feature toggles	0 (0%)	10 (50%)
	Create a cleanup branch	0 (0%)	4 (20%)
	Change a feature toggle to a configuration setting	0 (0%)	2 (10%)

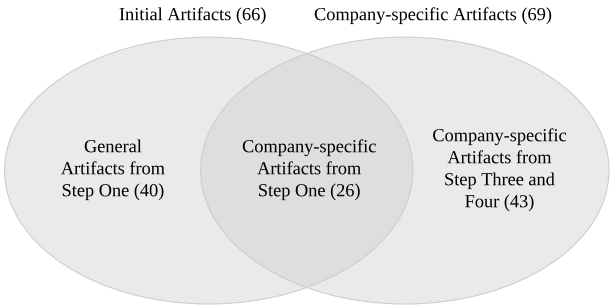


Figure 3.4: Number of artifacts found in Steps 1, 3 and 4

In Step Five, we conducted a survey to gather additional information about the usage of feature toggles practices in the industry. We had company-specific artifacts of 38 companies. Of these 38 companies, we sent out the survey to 36 companies for which we had the contact information for release engineers and/or developers. In addition to these companies, we identified a list of 20 companies that use feature toggles in their companies. These companies are mentioned in artifacts as examples of companies that are using feature toggles but we cannot find company-specific artifacts related to their practice usage. We found contact information for release engineers and/or developers in 9 of these companies and sent the link to the survey to them. In total, we sent the survey to 45 companies. We got 20 responses for a response rate to the survey of 44%. These 20 responses are from at least 17 companies. We cannot compute the exact number because three respondents did not identify their company names.

As mentioned in Section 3.2, we used a Likert scale (45) with five options for 12 of the 17 practices for which Likert scale options can be used. In our analysis, we grouped *Always*, *Mostly*, and *About half of the time* responses and assumed the companies that selected these options use the practice. We also grouped *Rarely* and *Never* and assumed the company does not use the practice if the respondents selected one of these two options. The detailed result of the survey responses on the 12 questions with Likert scale options is shown in Figure 3.5. For the 5 remaining practices which include all 3 implementation practices and 2 management practices, survey respondents chose from a list of provided answers or to add text to an open-ended "other" response. For example, for the "Use management systems" (Mg1) respondents could choose among open or closed source systems, chose they did not use a management system, or add any answer to the "other" response. The survey questions are listed in the Appendix A of the dissertation.

We use results from analyzing company-specific artifacts and survey responses to answer RQ1.2. The result of the analysis of company-specific artifacts is shown in the third column and the survey result is shown in the last column of Table 3.4. The frequency of usage of each practice in both company-specific artifacts and survey results is shown in this table.

Based on the survey responses, the companies in which the survey respondents work have been using feature toggles for an average of 4.8 years. Among 20 respondents, 19 respondents use toggles to have gradual rollouts. Nineteen respondents use toggles to support CI of partially-completed features, and 17 respondents use toggles to perform A/B testing. Fifteen respondents use toggles to have dark launches.

In the following subsection, we go through each of the categories and highlight the main findings on the usage of practices. The analysis is based on the survey responses and company-specific artifacts for each category of practices.

Management Practices

The most used practice is "Use management systems" (Mg1) based on both company-specific artifacts and survey responses. However, in comparison to configuration options, Sayagh et al. (4) show that developers do not tend to use existing configuration frameworks.

For the "Determine the applicability of feature toggle" (Mg4) practice, four survey respondents stated that the feature toggle is always added when a new feature is added or any feature is changed. They do not have any decision-making process for using feature toggles. In companies where a feature toggle is added for each new feature, there will eventually be a large number of feature toggles so management and deletion of the toggles are more critical to prevent increased code complexity and dead code. The "Log changes" (Mg3) practice enables practitioners to follow the "Track unused toggles" (C2) practice from the clean-up category as shown in Table 3.3. If a company logs every change made on feature toggles, tracking unnecessary toggles will be easy.

Initialization Practices

The most used practice in the initialization category based on both company-specific artifacts and survey responses is "Set up the default values" (I1) based on Table 3.4. The "Use naming conventions" (I2) and "Determine the type of the toggle" (I3) are next in the rankings. The usage ranking of the practices in this category is the same in both company-specific artifacts and survey responses.

"Determine the type of the toggle" (I3) is a practice that helps practitioners to use the "Limit the number of feature toggles" (C3) practice in the clean-up category more efficiently, as shown in Table 3.3. If the type of the toggles is pre-determined, the practitioners have a list of short-lived toggles as a suggested list of toggles to remove. Instead of checking all of the toggles, the short-lived toggles could be checked for removal.

Implementation Practices

As shown in the first column of Table 3.4, the mean usage frequencies of implementation practices is 66% based on company-specific artifacts and 100% based on the survey. This category of practices is the most used practices in the industry based on our results. When a company uses feature toggles, the development team implements the code of the feature toggle including a mechanism to store the values of the toggle, select the type of the assigned value, and determine how to access the value.

For "Ways of accessing the values" (Im2), the experiences from practitioners in (68), (69) and (70) as we mentioned in **Examples** of the practice in Section 3.3.1 reflect the popularity of using objects among other ways. This can justify the survey's result, the most used way is objects (25%). All the survey respondents which use more than one way (45%) use objects as one of the ways.

For "Store type" (Im3), using a configuration file is more popular than using databases in the company-specific artifacts; but the survey's responses indicate that databases and a combination of configuration files and databases are most used. We allowed respondents to add their own answers to this question, and three respondents mentioned *using a third-party service*, such as to get values from LaunchDarkly servers. However, we did not identify this option in analyzing company-specific artifacts. We added this new store type to Table 3.4 for the survey responses. As we mentioned the comparisons made between using configuration files and databases in (22), (67), (71), (3) and (69) in the **Examples** part of "Store Type" (Im3) practice, using databases gives the development team more flexibility but using configuration files is faster. These comparisons from the practitioners' point of view can justify different results for the practice based on company-specific artifacts and the survey. Companies may start with configuration files and after realizing the disadvantages of it, switch to using a database or a combination of configuration files and databases to have a fast updating of values.

The difference between usage frequencies of practices based on the artifacts and based on the survey's responses shows that companies may change the implementation details

of feature toggles over time and based on their experiences. Company-specific artifacts mentioned the feature toggles implementation details at the time of publishing the artifacts, but the survey's responses reflect the current implementation details.

Clean-up Practices

Based on the company-specific artifacts and survey responses, the practices of the clean-up category are the least used category of practices. The mean usage frequency of clean-up practices is 3% based on company-specific artifacts and 39% based on the survey which is the lowest frequency category.

We did not find any comparison between "track unused toggles" (C2) and "Change a feature toggle to a configuration setting" (C5) with the rest of the practices in this category. Based on the comparisons done by the practitioner for the rest of the practices as we mentioned in **Examples** of each practice in Section 3.3.1, "Use cards/tasks/stories for removing toggles" (C1.3) seems to be not very useful and "Time bombs" (C1.1) imposes pressure to organizations. However, "Limit the number of feature toggles" (C3) and "Create a cleanup branch" (C4) are good to follow based on the practitioners' experiences. Companies may use more than one of these practices for cleaning up the feature toggles, as we observed in survey responses.

We have both company-specific artifacts and survey results from twelve companies shown in Table 3.5. We are able to compare what we found in company-specific artifacts with their responses to the survey for these twelve companies. For eight companies, there are discrepancies in survey responses and company-specific artifacts about the type of assigned values to feature toggles and ways of accessing these values in Implementation practices. There are no discrepancies found for Clean-up practices, the survey responses and observations from company-specific artifacts are aligned. Three of the companies do not have documentation for their feature toggles anymore, based on survey responses. It could be because of the management systems they have which provide documentation facilities. One of the companies followed "Determine the applicability of feature toggles" (Mg4) based on the company-specific artifacts. However, they add a feature toggle for any new feature now, based on their response to the survey.

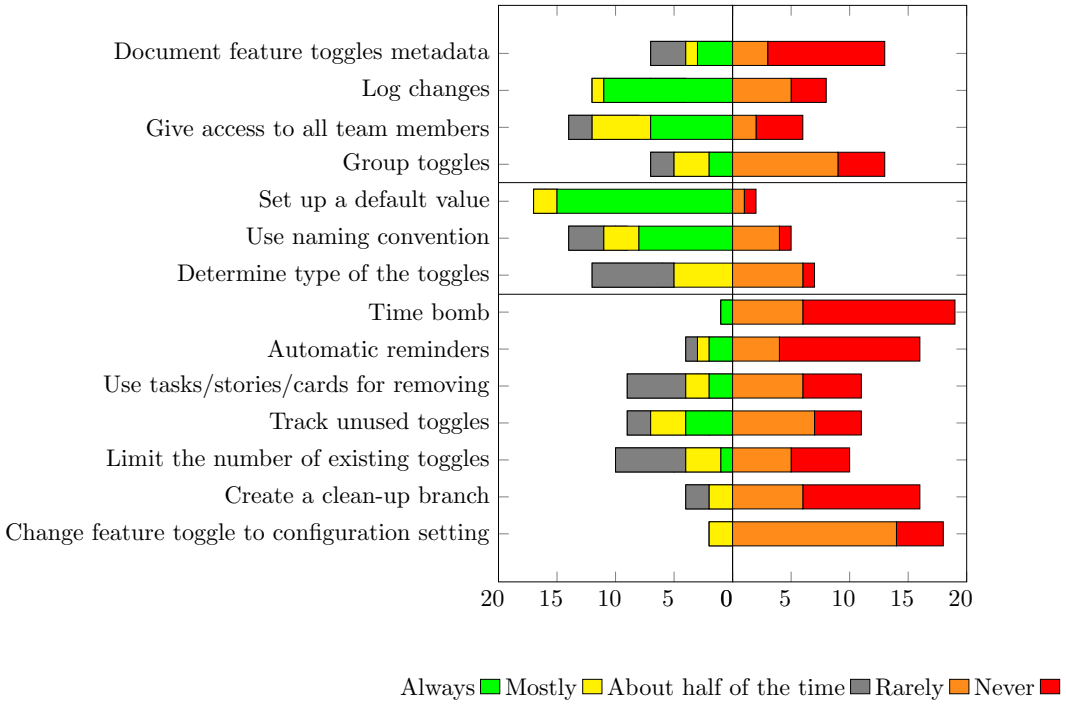


Figure 3.5: Frequency of using the subset of feature toggle practices with the Likert scale based on the survey

3.4 Recommendations for Practitioners

Acknowledging that teams will not adopt all 17 feature toggle practices, we provide our prioritization for the use of these practices. We base our recommendation on the frequency of usage of the practice by other practitioners, the practitioners' experiences, and the substantiated benefits of the use of similar practices in other areas of software engineering.

Management Practices

We recommend the use of three of the five management practices:

- The **"Use management systems" (Mg1)** practice is used to manage the added complexity and technical debt of adding feature toggles through a separate system. This practice is the only practice that fell into the **High** level of confidence in Table 3.2. In addition, as shown in Table 3.4, all survey respondents use this practice in their development cycle, and this practice is also the most used practice based on company-specific artifacts. Moreover, this practice is related to twelve practices, as shown in Table 3.3, indicating

that using a management system can also help in following other practices.

- The **"Determine applicability of feature toggle" (Mg4)** practice is the only practice that covers the **Decision** phase of the feature toggle lifecycle, as shown in Figure 3.3. Also, based on the survey results in Table 3.4, this practice is the second most used practice in this category.
- The **"Log changes" (Mg3)** practice is a common practice in the software engineering area. Besides, following this practice help to follow "Track unused toggles" (C2) based on the related practices shown in Table 3.3.

Initialization Practices

We recommend one of the three initialization practices:

- The **"Set up the default values" (I1)** helps to mitigate unwanted behavior of the feature toggles that can have severe impacts on the software system. Moreover, based on the frequency of usage of practices in Table 3.4, I1 is the most used initialization practice based on company-specific artifacts and survey respondents.

Implementation Practices

The Implementation category has three practices. Each of the practices provides practitioners with more than one option to choose from for implementing feature toggles. In the **Examples** of the practices in Section 3.3.1, the advantages and disadvantages of different options are provided, which could guide the practitioners in selecting appropriate options based on their project context. Our recommendations are as follows:

- We recommend the use of Boolean values in the **"Type of assigned values" (Im1)**. In Table 3.4, using Boolean values is the most used type based on both company-specific artifacts and survey responses. All the survey responses that use more than one type use Boolean values as one of the types.
- For the **"Ways of accessing the values" (Im2)** practice, we recommend having objects of feature toggles with a method to determine the value of the toggles. Having objects of feature toggles is more manageable based on the practitioner's experience, as mentioned in Section 3.3.1. Based on the survey's result shown in Table 3.4, using objects is the most

used way, and all the survey respondents who use more than one way use objects as one of the ways.

- **Strong Recommendation:** For **"Store type" (Im3)**, we suggest the use of a combination of configuration files and databases. Based on the provided examples for this practice in Section 3.3.1, using databases gives the development team more flexibility but using configuration files is faster. The combination of them brings the advantages of both ways.

Clean-up Practices

The common goal of all practices in the Clean-up category practices is to remove feature toggles when the purpose of using the toggles is accomplished. A practitioner can choose to use multiple of these Clean-up practices (C1-C5). We recommend two of the five practices:

- Based on the **Examples** provided for practices in this category, we strongly recommend following **"Limit the number of feature toggles" (C3)** and/or **"Create a cleanup branch" (C4)** for removing feature toggles. The experiences of practitioners show that these practices work well for development processes. For "Create a cleanup branch" (C4), the advantage is that planning to remove the toggle is done when the context of using the feature toggle is fresh in the developer's mind, so it is easier to plan for it.

Besides all of our suggestions, the context of the project and the development team's culture can affect choosing each one of these practices. Project managers should consider the context when making decisions about following each practice.

3.5 Limitations

In this section, the limitations of this research study are discussed.

3.5.1 Finding artifacts

In Step One, we used a keyword search based on five keywords to find grey literature and peer-reviewed papers. We selected artifacts that were related to the use of feature toggles in software development by reviewing the first 10 pages of the search results. We also followed links and references to other artifacts in selected artifacts. We may have missed artifacts because of using the limited keywords and reviewing the first 10 pages of search results.

In addition, we did not follow a systematic literature review search process to find related peer-reviewed papers. So, we may have missed peer-reviewed papers.

In Step Three, we searched for company-specific artifacts based on the companies found in the initial artifacts. Data from companies who have not shared their results on the Internet are not included in our study.

3.5.2 Identification of Practices and Categories

In Step Two, we did not use any automatic technique or tools to identify practices. We may have missed some practices which were mentioned implicitly in artifacts.

Another limitation is the lack of specific examples by companies of using feature toggle practices. Practitioners mentioned most of the practices with no concrete example.

In addition, testing practices are not identified and mentioned in the list of practices. Testing of the system which has feature toggles has different aspects, such as unit testing of feature toggles, testing all combinations of feature toggles enabling and disabling, and testing dependent feature toggles. Another study should be conducted to cover testing concerns and practices when a development team uses feature toggles.

3.5.3 Extraction of Practice Usage from Company-specific Artifacts

In Step Four, we reviewed company-specific artifacts to extract feature toggle practices usage. If the practice was not mentioned in the artifacts, we cannot conclude that the company does not use the practice. To overcome this limitation, we conducted a survey to gather more information about the usage of feature toggle practices in companies.

3.5.4 Conducting Survey

In Step Five, we found contact information of individuals associated with company-specific artifacts or who were release managers or developers of the companies. The contact information for some of the individuals could not be found or was old and out of date. To overcome this limitation, we found contact information of current development team members, such as release managers or developers of the companies, using company websites or social media pages, such as LinkedIn. Additionally, the small sample size of the survey was a limitation.

3.6 Conclusion

Feature toggles are a technique often used by companies who practice CI/CD to integrate partially-completed features into the code, conduct a gradual rollout, and/or perform experiments. However, the development practices used by these organizations have not been enumerated in prior research. We performed a qualitative analysis of 99 artifacts from grey literature and 10 peer-reviewed papers. We identified 17 feature toggle practices in four categories: Management practices (6), Initialization practices (3), Implementation practices (3), and Clean-up practices (5). We also quantified the frequency of usage of these identified practices in the industry by analyzing company-specific artifacts and conducting a survey.

The most popular practice in each category is consistent across the company-specific artifacts and survey responses. We observed that all of the survey's respondents "Use a management system" (Mg1) to create and manage feature toggles in their code. "Document feature toggle's metadata" (Mg2), "Log changes" (Mg3), and "Set up the default values" (I1) are three additional highly-used practices in the industry based on company-specific artifacts. The least used category of practices is Clean-up practices, even though cleaning up the feature toggles helps with managing the added complexity to the code and removing dead code. Inattention to removing feature toggles can cause severe problems, such as what happened to Knight Capital Group.

Table 3.5: 38 Companies and their usage of identified practices from company-specific artifacts

	Management						Initialization			Implementation			Clean-up				
Company	Use management systems	Document feature toggle's metadata	Log changes	Give access to team members	Determine applicability of feature toggle	Group the feature toggles	Set up the default values	Use naming convention	Determine the type of the toggle	Type of assigned values	Ways of accessing the values	Store type	Track unused toggles	Add expiration date	Change a feature toggle to a configuration setting	Limit the number of feature toggles	Create a cleanup branch
Airbnb	✓	✓	✓				✓				✓						
Apiary	✓	✓	✓	✓			✓			✓	✓						
AppDirect	✓	✓	✓				✓			✓	✓						
Behalf	✓	✓	✓				✓			✓	✓						
CircleCI	✓	✓	✓	✓			✓			✓	✓						
Checkr	✓		✓							✓	✓	✓					
commercetools	✓	✓	✓		✓					✓				✓			
Domain	✓	✓					✓			✓							
DropBox	✓		✓							✓	✓	✓	✓				
Envoy	✓	✓	✓	✓			✓			✓	✓						
Etsy	✓					✓	✓			✓	✓	✓					
Facebook	✓	✓								✓	✓	✓					
FINN.no	✓	✓	✓		✓		✓			✓	✓						
Flickr		✓								✓	✓	✓					
GoPro	✓	✓	✓			✓	✓			✓	✓						
Google Chrome		✓					✓			✓	✓	✓					
IBM								✓		✓		✓					
Instagram	✓	✓	✓	✓						✓	✓						
InVision	✓	✓	✓		✓		✓	✓	✓	✓	✓			✓			
Librato	✓									✓	✓	✓					
Lyris					✓		✓	✓				✓		✓			
Main Street Hub	✓	✓	✓				✓			✓	✓						
Microsoft	✓	✓			✓		✓			✓	✓	✓					
Outbrain					✓							✓					
Pinterest	✓	✓	✓				✓				✓	✓					
Rally Software		✓												✓			
Reddit	✓						✓	✓		✓	✓	✓					
Slack	✓													✓			
Soluto	✓							✓		✓							
Surfline	✓	✓	✓	✓			✓			✓	✓						
ThoughtWorks	✓			✓						✓	✓	✓		✓			
thredUP	✓	✓	✓		✓		✓			✓	✓						
Travis-CI	✓									✓	✓						
Twilio	✓	✓	✓				✓			✓	✓						
Upserve	✓	✓	✓		✓		✓			✓	✓						
Visma	✓									✓							
WePay	✓	✓	✓				✓			✓	✓						
Wix	✓	✓	✓	✓						✓		✓					
Total (38)	32	25	21	7	8	2	22	5	1	32	28	15	2	6	0	0	0

CHAPTER

4

HEURISTICS AND METRICS FOR STRUCTURING FEATURE TOGGLES IN SOURCE CODE

4.1 Motivation

Developers may follow certain structures to incorporate feature toggles in their code. As an example, checking the value of a feature toggle could be done through different structures. One structure is to check the value of *all* feature toggles through one method. As an alternative, each feature toggle could have its own specific method to check the value of that toggle. So, even a simple task of checking the value of feature toggles could be structured in more than one way. *Feature toggles structured incorrectly could result in technical debt* (76; 10). Thus, arises a need for guidelines on how to structure and manage toggles. *The goal of this research is to aid software practitioners in structuring feature toggles in the codebase by proposing and evaluating a set of heuristics and corresponding complexity, comprehensibility, and maintainability metrics based upon an empirical study of open source repositories.*

Software practitioners prefer to learn through the experiences of other practitioners (18). To address the goal, we systematically study feature toggle usage in open-source software repositories, and

1. develop heuristics, (FT-heuristics), to guide the structuring of feature toggles; and
2. propose metrics, (FT-metrics), to support the heuristics.

Accordingly, we state the following research question with three sub-research questions:
RQ2 How feature toggles are structured in the code?

RQ2.1 (heuristics): What heuristics can be used to guide the structuring of feature toggles in a codebase?

RQ2.2 (metrics): What metrics can be used to measure the effect of incorporating proposed heuristics in the codebase?

RQ2.3 (survey and case study): To what extent do the practitioners incorporate the heuristics, and how the metrics are related to those heuristics?

4.2 Methodology

Figure 4.1 summarizes our two-phase method to develop the FT-heuristics and FT-metrics and the resulting dataset. Phase 1 addresses RQ2.1 and RQ2.2, and Phase 2 addresses RQ2.3.

4.2.1 Dataset–Repositories

We analyzed GitHub repositories that incorporate feature toggles. First, similar to Meinicke et al. (14), we searched GitHub repositories for the keyword "feature toggle". Our search date was 23-May-2019. GitHub categorizes search results into 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 was the most appropriate for our study. GitHub search skips forks by default, so the commits of the forked repositories are excluded automatically. Including forks could

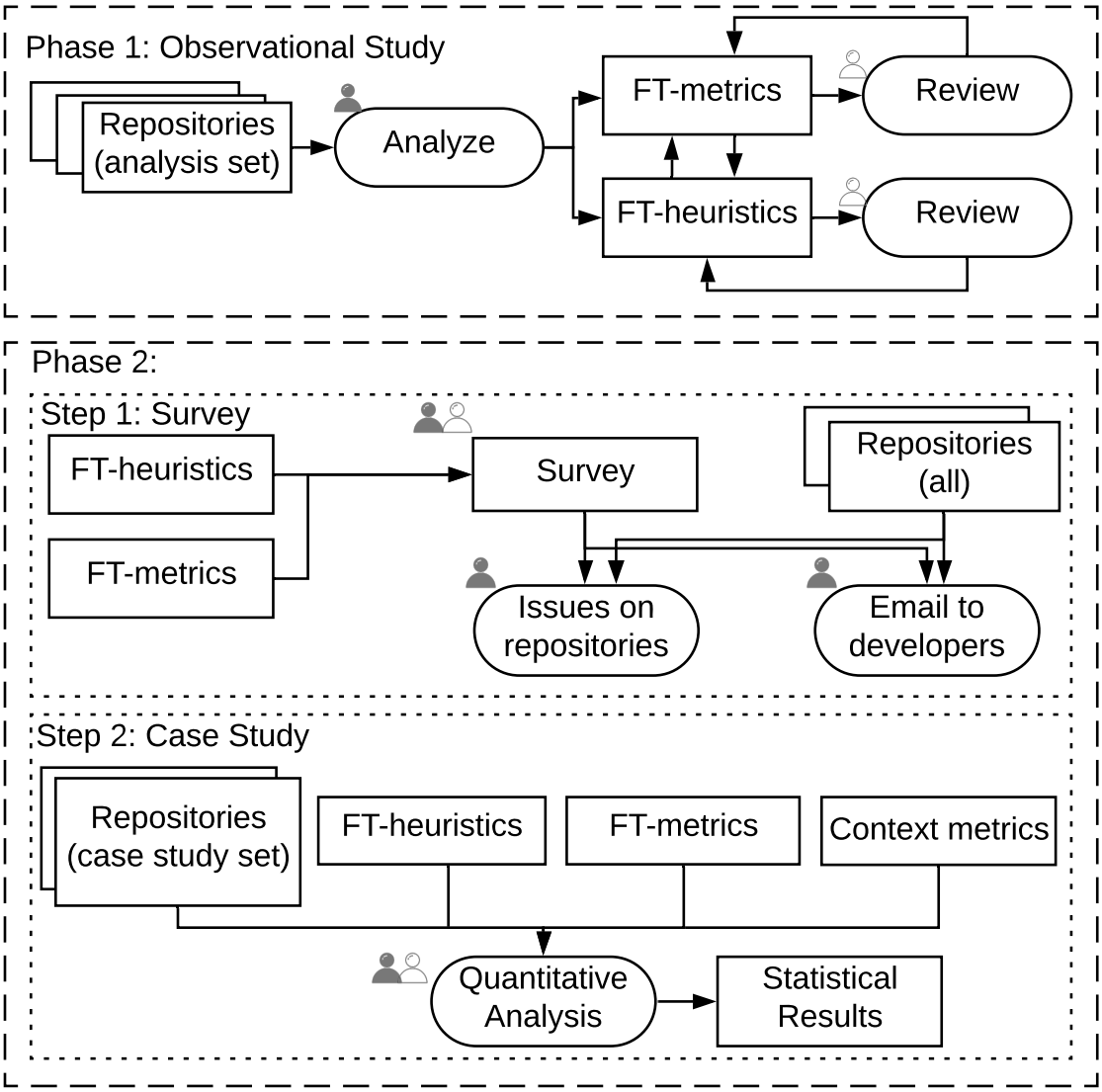


Figure 4.1: Research methodology outline.

skew the results. Search results in “Repositories” and “Code” categories listed repositories of feature toggle management systems which are not the focus of this work. Results under “Issues” can also be considered as a source to find repositories that use feature toggles. For this study, we used results in the “Commits” category but future work could consider using “Issues”.

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. On inspection, we found the first 400 search results were most likely relevant to feature toggles. We manually examined each of the 400 commits including commit messages, the files which were changed, and the changes which were made to the code; as well as the issues, pull requests, and documentation in each repository to find details about incorporating feature toggles.

We identified 110 relevant commits after excluding commits that met at least one exclusion criteria:

1. Commit URLs that no longer exist;
2. Commits related to toggle/button input controls in the user interface (UI) of an application. For example, the commit message is "add toggle-all feature" and the change is "added a checkbox to check all the options in UI". However, wrapping a UI element with a feature toggle is not excluded;
3. Commits in repositories of feature toggle management systems that are not in the scope of this study; and
4. Commits from the repositories in which we cannot distinguish feature toggles from configuration options. For example, if a toggle is defined in a way that end-users can change its value, we exclude the repository.

The selected 110 commits belonged to 80 repositories that use feature toggles in their development process. We list the language and lines of code (LOC) of these repositories in Table 4.1.[1]

Table 4.1: Characteristics of the identified repositories.

	Language	# repositories	LOC range
Top Five	TypeScript	17	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		80	170 to 3,547,167

[1]Data availability: The data including commits, repositories, and case study details are posted here: `https://sites.google.com/view/feature-toggles-dataset/`.

We randomly selected 60 repositories as the *analysis set* which was analyzed to develop the FT-heuristics and to identify the FT-metrics. *The case study set* consisting of all 80 repositories (60 repositories from the analysis set and 20 additional repositories) to find the relation between the FT-heuristics and the FT-metrics.

4.2.2 Phase One: Observational Study

Figure 4.1 (top block) outlines the steps in Phase 1. To address RQ2.1 and RQ2.2, we manually developed FT-heuristic and FT-metrics iteratively and concurrently through an observational study of the repositories in the analysis set. We looked for structural patterns and developed a mental model. The FT-heuristics help in the actionability of FT-metrics. The manual analysis for each repository contained the following steps:

1. Starting with the GitHub 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 and the feature toggle class. If the configuration file was not found in the 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, which contains a list of toggles.
2. We searched for the usage of all feature toggles identified in Step 1 in the code and commit history;
3. We inspected issues, pull requests, documentation, and comments to find information about incorporating toggles.
4. We recorded the details of structures of incorporating feature toggles observed in Steps 2 and 3, including definition details (e.g. file of definition, meta-data, names); usage details (e.g. checking the value of toggle); removal details (e.g. removed lines of codes, changed files).

We develop the FT-heuristics to structure feature toggles using notes recorded in Step 4 of the above process. Following the open coding technique (43), we assign codes to observational notes from Step 4 and define an FT-heuristic for each category of codes based on our developed mental model. For example, one part of the notes for each repository is about how they check the values of feature toggles with methods. We observe two codes in that note “Check all the values with one method” and “Check the values with specific methods for each toggle”. We observed less complexity and less maintainability effort in repositories that use a shared method for checking feature toggle values. So, we define Heuristic 1 (SharedMethod) based on these codes and our observations.

The FT-heuristics embody the actionable recommendations based on the current state of using feature toggles in analyzed repositories. In addition, based on the notes, FT-heuristics, and considering relevant literature on metrics such as CK metrics (28) and metrics to measure variability in software product lines using C preprocessors to implement configuration options (34), we identified FT-metrics to support FT-heuristics iteratively.

During this iterative process, we had the list of the metrics from the literature (28) and (34). Although we did not have pre-defined criteria, we discussed the applicability of each one of the metrics to observed details of incorporating feature toggles in repositories iteratively by the two researchers. For example, metrics "Depth of Inheritance Tree (DIT)" and "Number of Children (NOC)" from CK metrics have no connection with feature toggles based on researchers' observations. But, the concept behind metric "Lines of Feature Code (LOF)" from variability metrics is used in our "Feature toggle lines of code (M9)" metric. The discussions between the two researchers help to reduce the subjective selection bias of the metrics. In the end, the metrics with no effect by incorporating feature toggles were removed from the list. Note that, not all metrics came from literature. We also added other metrics to the final list during this iterative process based on our observations, such as the presence of guidelines and the presence of duplicate code. Then, we categorized FT-metrics based on their effect and existing categories in literature (4) in three categories: *complexity*, *comprehensibility*, and *maintainability* using card sorting technique (77). Card sorting technique for categorizing FT-metrics is done by two researchers. We pre-define these three categories and discuss the correct category for each FT-metrics. In this discussion, we rely on our observations from analysis repositories in this study and our experience as software engineers. When both researchers agree on a category for a FT-metric, we move forward to the next metric.

The first researcher performed Phase 1 in consultation with the second researcher, who critically examined the definition and examples of the FT-heuristics, examined the definitions and measurements of the FT-metrics and gave feedback in all steps. Sections 4.3 and 4.4 present the results of Phase 1.

4.2.3 Phase Two: Survey and Case Study

The bottom block of Figure 4.1 outlines the two steps in Phase 2, which we follow to address RQ2.3.

Step 1 (Survey): To evaluate the acceptance of our FT-heuristics by practitioners of all repositories in our dataset, we conducted a survey. In the survey, we asked to what extent practi-

tioners agree that the FT-heuristics can be used to guide practitioners on how to structure and use feature toggles to reduce technical debt. We used the Likert scale (45) to specify the level of agreement: Strongly disagree to Strongly agree. [2] To distribute the survey, first, we submitted *issues* in the repositories that allowed us to submit an issue. Second, we sent emails to practitioners when an email address of a feature toggles' contributor was available.

Step 2 (Case Study): To find the relation between FT-heuristics and FT-metrics, we analyzed the default branch of each GitHub repository in the case study set.

First, using the commit by which we identified the repository, we

1. found the list of toggles in the repository;
2. measured each of the identified FT-metrics; and
3. checked whether the repository follows each of the FT-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 the *cloc* (78) to calculate lines of code and identify the language. Using the GitHub profile of each repository, we identified the number of contributors for that repository. The number of toggles was identified via manual inspection of the repository.

Next, to examine the relationship between the FT-heuristics and the FT-metrics, for each heuristic, we separated repositories into two groups based on whether they follow (F) or do not follow (NF) that heuristic.

To compare the differences between the metric values yielded by the following (F) and not following (NF) groups of repositories, we define a measure named *Improvement* in Section 4.5.3. Improvement is computed as the percentage improvement in a FT-metric by following a FT-heuristic. Using the improvement measure, we discuss the trends in case study repositories.

In Appendix B, we report preliminary statistical analyses on case study data. This analysis includes regression analysis using the *Best Subset Selection* (79) (linear regression models for numeric metrics and logistic regression models for binary metrics) for each FT-metric.

[2] Published data includes the survey questionnaire.

The first researcher collected data, and the first and second researchers performed statistical analysis of the results. We report the result of the survey; and the observed relation between FT-heuristics, FT-metrics, and context metrics in Section 4.5.

4.3 FT-Heuristics

We now describe FT-heuristics which we derive by analyzing 60 "analysis set" repositories in Phase 1 of our method. For each FT-heuristic, we provide examples found in the repositories which follow or not-follow that heuristic. Our naming convention for the example is H (for Heuristic), followed by the heuristic number, followed by the subscript FE if the example is the following example, and NFE if it is a not-following example. The number at the end of the subscript is a counter of examples for that heuristic. For instance, $H1_{FE1}$ means the first following example for Heuristic 1. The number in the parenthesis in front of each FT-heuristic name (subsection title) is the number of the repositories that follow the heuristic. Note that a low number for a heuristic does not necessarily correspond to low importance for that heuristic. A low number could be a sign of a related bad smell. For instance, we are aware that not having test cases is a bad smell but only 17 repositories follow H6 (Testing).

The derived FT-Heuristics are similar to general software engineering best practices. However, we find that these are not followed by developers in all repositories in the analysis set. So, increasing the awareness of the developers about the impact of following these heuristics is important.

4.3.1 Shared Method to Check Value (26)

Heuristic 1 (SharedMethod). Using one *shared method* to check the value of all feature toggles, instead of having a method to check the value for each feature toggle, 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()` (5). The lower the number of feature toggle value checking methods, the lower the code complexity. Having fewer feature toggle value checking methods also decreases

1. the number of files and the lines of code that need to be modified to implement and maintain a feature toggle; and

2. the probability of the presence of dead codes when deleting feature toggles because an associated feature toggle value checking method does not need to be deleted.

$H1_{FE1}$: In Listing 4.1, the name of the toggle is passed to the method and the value of the toggle is checked in the list of feature toggles (80). 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 need 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 4.1: One shared IsEnabled value checking method (80).

$H1_{NFE2}$: The code in Listing 4.2 shows each feature toggle having its own function to check its value (81). 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 code that should be changed is larger compared to $H1_{FE1}$.

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

Listing 4.2: IsEnabled function for one toggle (81).

4.3.2 Self-Descriptive Feature Toggles (19)

Heuristic 2 (SelfDescriptive). Using intention-revealing names for toggles, adding a description field in the configuration file as a meta-attribute for each feature toggle, and including comments when using the toggles can improve comprehensibility.

Having self-descriptive code is a known practice in software development (64; 82). Self-descriptive code improves understanding and reduces code maintenance effort. Also, Sayagh et al. (83) suggests having self-descriptive configuration options. Feature toggles may remain in the codebase for a while and should be treated similarly to the implementation code. For example, adding comments is a way to make code understandable.

H2$_{FE1}$: As Listing 4.3 shows, each toggle in the configuration file of the CFS-Frontend repository of the UK Education and Skills Funding Agency (84) has an intention-reveling name and a description that makes the purpose of the toggle clear.

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

Listing 4.3: A feature toggle with description (84).

H2$_{FE2}$: A repository of Automattic (85) uses intention-revealing names and comments 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."*

H2$_{FE3}$: In the configuration file of a Salesforce repository (86), the feature toggles are grouped in two groups: *long term toggles* and *short term toggles*. The following is the description developers provided 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 to 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 the comment is *"add a new toggle here if you expect it to just be a short-term thing, i.e. we'll use it to control the 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.

In a survey study (5), practitioners suggested "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. Later, if developers need to limit the number of feature toggles in the code, they have a list of short-lived toggles which can potentially be removed first from the code.

H2$_{NFE4}$: Developers of HMCTS (87) named a feature toggle `FEATURE_TOGGLE_520` which doesn't convey its purpose.[3]

[3]This feature toggle is now removed from the code. The link to the removing commit is `https://bit.ly/34tcjOk`

4.3.3 Guidelines for Managing Feature Toggles (10)

Heuristic 3 (Guidelines). Providing *guidelines* for adding or removing feature toggles can improve comprehensibility and maintainability.

Management of feature toggles, including adding and removing toggles, is a challenge for developers and project managers. If feature toggles are added arbitrarily, a large number of toggles may end up in the code after a while. Developers should know when to add a feature toggle (For every new feature? For every huge change?) and when to delete a feature toggle (In a month? After publishing a new release?). Hence including guidelines for adding and removing toggles is important. Dead code may be introduced if the developers do not know when or how to remove a toggle correctly.

An example of feature toggle management is using pull requests to remove a toggle (5). We observed that developers use issues and pull requests to add and remove toggles. Development teams may use other project management systems, such as a Kanban board, or wiki pages (4) to manage toggles, however, these are not tightly integrated with the code base and may be missed by developers.

$H3_{FE1}$: In a repository of The Guardian (88), developers use pull requests to delete a feature toggle. Developers can use *"feature toggle in:title"* as a search string in the list of issues and pull requests of repositories to find those related to toggles.

$H3_{NFE2}$: In a repository from the Australian Department of Veterans' Affairs (89), the guideline for adding feature toggles is provided in the `README` file. Although the developers have guidelines for adding toggles, they do not have guidelines for removing toggles.

4.3.4 Use Feature Toggles Sparingly (53)

Heuristic 4 (UseSparingly). Using a feature toggle in *as few locations as possible* in the code can 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 development effort and the possibility of creating dead code. The more number of paths in the code, the higher the code complexity. Note that, the focus in this FT-heuristic is not to minimize the "number" of the feature toggles, but the count of files that a toggle is used in them is better to be as low as possible.

$H4_{FE1}$: 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 could be checked or used in the rest of the code (6). Listing 4.4 shows an example of using feature toggles to set the value of another variable. Instead of individually checking the three conditions in the example, only the `canFork` variable is checked in the rest of the file (90). In Listing 4.4, 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 4.4: Use feature toggles to set value to a new variable.

H4$_{\text{NFE2}}$: One way to store the value of a toggle is using configuration files, but in some repositories, more than one configuration file exists for the same set of feature toggles. The Multiplication Tables Check (MTC) project of the UK Department of Education (91) has 14 files for feature toggles. To remove or edit a feature toggle, a developer must remove or edit the toggle in all of the 14 configuration files. Missing any of these files 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 complexity and decreases the maintainability of the code.

4.3.5 Avoid Duplicate Code in Using Feature Toggles (57)

Heuristic 5 (AvoidDuplicate). When a feature toggle that wraps the same code is used more than once in the same file, *creating a method containing the feature toggle with the wrapped piece of code* can improve maintainability.

Duplicate code is a code smell (92). When the consequent fragment of code wrapped by a feature toggle's conditional if-statement appears more than once in the same file; that counts as duplicate code, adds complexity to the code, and may create dead code. However, the *extract method* refactoring pattern (92) could be used with feature toggle's consequent fragment of code to avoid duplicate code and subsequent repercussions

H5$_{\text{NFE1}}$: In Salesforce's refocus repository (93), the code in Listing 4.5 is a fragment of code wrapped with a feature toggle. This block of code appears twice in the same file. The *extract method* refactoring pattern could be used to prevent duplicate code.

```
1 if (featureToggles.isFeatureEnabled('enableWorkerActivityLogs') &&
    jobResultObj && logObject) {
```

```
2   mapJobResultsToLogObject(jobResultObj, logObject);
3   if (featureToggles.isFeatureEnabled('enableQueueStatsActivityLog'
     )) {
4     queueTimeActivityLogs.update(jobResultObj.recordCount,
     jobResultObj.queueTime); }
5   activityLogUtil.printActivityLogString(logObject, 'worker');
6 }
```

Listing 4.5: Code block that appears twice in the same file (93).

4.3.6 Test Cases for Feature Toggles (17)

Heuristic 6 (Testing). Including *test cases* for each feature toggle can improve maintainability.

Software testing is a recommended activity for assessing the quality and correctness of the code (94). Feature toggles can determine the logic flow and behavior of a product, so must be correct and of high quality. Automated test cases of feature toggles can be used as regressions tests which enhances maintainability. Test cases should remain in the code if the development team decides to make the feature permanent.

$H6_{FE1}$: In the Multiplication Tables Check (MTC) project of the UK Department of Education (95), when developers decided to make a feature wrapped in a feature toggle permanent, they removed unit tests for a disabled toggle and kept the tests for an enabled toggle with changed names.

4.3.7 Complete Removal of a Feature Toggle (21)

Heuristic 7 (CompleteRemoval). Ensuring *complete removal* of a feature toggle by removing it from source code files, configuration files, and test cases can improve maintainability.

Developers should remove feature toggles when the purpose of using the toggles is accomplished. They should remove the code related to the feature toggle from all files in the source code, including configuration files and test files. Incomplete removal can cause problems such as dead code (41). One solution to ensure complete removal of a feature toggle is to have an implementation that throws a compilation error when a feature toggle exists in the code after being removed from the configuration files.

$H7_{FE1}$: In this commit (96), the developers of the Multiplication Tables Check (MTC) project from UK Department for Education completely removed a "prepareCheckMessaging" feature toggle from the configuration files, code, and test cases.

4.4 FT-metrics

We now describe the FT-metrics we identify to support the FT-heuristics. Following the steps in Section 4.2.2, first, we manually analyze the 60 repositories in the analysis set. We identify 12 metrics based on our observations of structuring feature toggles in the repositories. Based on their effect on the code base, using the card sorting technique and existing categorization in the literature (4), we group these metrics into three categories: *Complexity*, *Comprehensibility*, and *Maintainability*.

Below, we list the 12 FT-metrics. The type of each metric, binary or numeric, is mentioned after the metric's name. For binary metrics (M3–M6, M10–M12), in this study, we consider the presence or absence of the metric in the repositories. A higher level of granularity may be obtained via an automated tool, as discussed in Section 4.7. FT-metrics are as follows:

Complexity is the degree to which a system has a design or implementation that is difficult to understand and verify (97).

- *M1: Number of added paths in code (Numeric)* is computed using McCabe's Cyclomatic Complexity (98). McCabe's Cyclomatic Complexity counts the number of paths in code based on the number of decision points. Incorporating a feature toggle 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, 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. 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 (28) which is computed as the number of methods in a class. To compute this metric, we count the number of feature toggle value checking methods in the feature toggle class manually. We assume all the methods have equal complexities, so the weight for all is 1.0.

Comprehensibility is the degree to which a system is understandable to the developers.

- *M3: Presence of guidelines (Binary)* helps developers know the processes of adding and removing a feature toggle. The absence of guidelines may cause problems such as the creation of a dead code after a toggle removal. Guidelines may be provided as a document in repositories or as comments in feature toggles' configuration files.
- *M4: Intention-revealing names (Binary)* for variables and methods is a known practice in coding (64). A feature toggle's name and related methods should tell the reader what value the toggle holds and what task the code wrapped by it accomplishes. M4 is a subjective metric.
- *M5: Use of comments (Binary)* as human-readable notes that support the source code is a coding practice (82) that helps developers understand the purpose and behavior of the feature toggles.
- *M6: Use of description (Binary)* for each feature toggle can be used to clarify a toggle's purpose. The description could be added as an attribute to the feature toggle class. Listing 4.3 in Section 4.3 is an example of including descriptions. Unlike comments which are not available everywhere, object attributes are accessible throughout the codebase.

Maintainability is the ease with which a software system can be modified to correct faults, improve performance or other attributes, or adapt to a changing environment (97).

- *M7: Number of files (Numeric)* which contain a feature toggle, including configuration, code, and test files. The higher the number of files that need to be changed to support feature toggles, the higher the probability to make a mistake. We count the number of files for each feature toggle and then average it for each 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 toggle used in two files. In a configuration file, a toggle is mentioned 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.
- *M9: Feature toggle lines of code (Numeric)* which are directly associated with a feature toggle when the toggle is added or removed from a repository. In general, the number of lines of code is a metric to measure maintainability in software systems (99). 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 not feature toggle usage and enclosed code) and then average it on the number of toggles in each repository.

- *M10: Presence of duplicate code (Binary)* is a code smell (100). Duplicate code is a problem of repeating the same block of the code. We consider a code fragment that contains checking the value of a feature toggle in a conditional statement and the piece of code wrapped by the toggle as duplicate code. In case of updating or removing the toggle, all 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 that is not used in any execution path (101). 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. In this study, dead code is considered "present" if we found a feature toggle definition or test cases for a toggle but that toggle is not used in the code anymore.
- *M12: Presence of test cases (Binary)* for feature toggles is a metric to measure whether feature toggles are tested. Feature toggles should be tested similarly to implementation code. We consider two types of test cases:

  1. checking the values of the feature toggles; and
  2. checking the behavior of the code based on the value of the feature toggle.

  If the repository has any type of test cases for the majority of the feature toggles in the code base, we record "yes" for this metric.

Table 4.2 shows hypothesized relations between FT-heuristics and FT-metrics. The hypothesized relations are determined based on observations in Phase 1 of the methodology and iterative discussions between the two researchers.

4.5 Survey and Case Study

We now explain Phase 2 of the method in Figure 4.1 and Section 4.2.3. We propose the following sub-research questions to investigate RQ2.3 on evaluating FT-heuristics and FT-metrics.

SRQ_{PA} (Practitioners' agreement): To what extent do practitioners agree that the FT-heuristics can be used to guide practitioners on how to structure and use feature toggles to reduce technical debt?

Table 4.2: Hypothesized relationship between FT-heuristics and FT-metrics.

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 (LOC)	✔				✔		✔
	M10 (Duplicate)					✔		
	M11 (Dead)	✔		✔	✔			✔
	M12 (Test cases)						✔	

H1 to H7 are list of FT-heuristics: SharedMethod (H1), SelfDescriptive (H2), Guidelines (H3), UseSparingly (H4), AvoidDuplicate (H5), Testing (H6), CompleteRemoval (H7)

SRQ_{HM} (Heuristics and metrics): What is the relation between the adoption of FT-heuristics and values of FT-metrics?

To address these proposed sub-research questions, we conduct a survey of practitioners from 80 repositories, and a case study with all 80 repositories as the case study set.

4.5.1 Practitioners Agreement by Survey (SRQ_{PA})

To evaluate the acceptance of FT-heuristics by practitioners, we asked the practitioners about the difficulty in managing feature toggles and the extent of their agreement that the FT-heuristics can be used to guide practitioners on how to structure and use feature toggles to reduce technical debt. We used a five-point *Likert* scale for difficulty: Not at all difficult (1) to Very difficult (5); for agreement: Strongly disagree (1) to Strongly agree (5). We included a N/A option too.

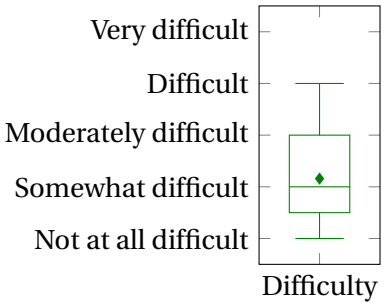
Table 4.3: Survey respondents' experience and frequency of using feature toggles. Underline indicates median.

Experience	# practitioners	Usage frequency	# practitioners
1–3 years	5	Rarely in projects	6
3–5 years	7	Half of the projects	6
5–8 years	3	Most of the projects	5
8+	0	All of the projects	2
N/A	5	N/A	1

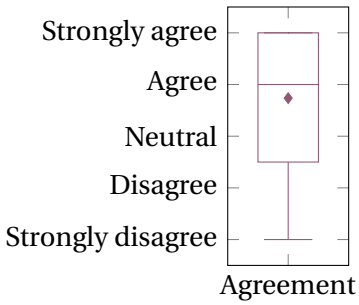
To reach out to practitioners and elicit their responses to the survey questionnaire, we first submitted 72 issues in 80 repositories in our dataset. The settings of the other 8 repositories did not allow us to submit an issue. We received 8 responses via issues. Next, we sent 57 emails to practitioners of 45 repositories associated with feature toggles' commits and changed files, and received 12 responses. For 35 repositories, the email addresses of feature toggles' contributors were not available. Table 4.3 shows the 20 survey respondents' experience and frequency of using feature toggles.

The 20 practitioners (survey respondents) have 3–5 years (median) of experience in using feature toggles. They use feature toggles in half of their projects (median). As summarized in Figure 4.2, the practitioners perceive managing feature toggles to be somewhat difficult and agree that feature toggles increase technical debt. Of the 20 survey respondents, 19 practitioners *strongly agree* and one practitioner *agree*s with CompleteRemoval (H7), stressing the importance of complete removal of feature toggle and how that can improve maintainability. The medians of the agreement to all FT-heuristics are over 4 indicating *Agree*, and the individual means of the agreement to FT-heuristics are all at least 3.7, which is close to *Agree*. The respondents agree that following each of the seven FT-heuristics could guide practitioners on how to structure and use feature toggles to reduce technical debt.

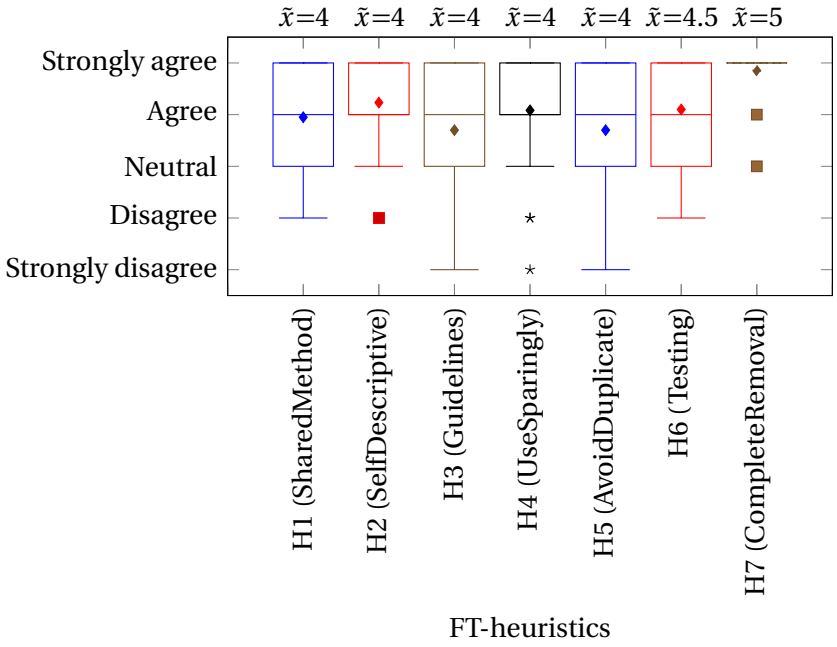
We also asked practitioners for their suggestions to reduce the technical debt of using feature toggles. Only one of the practitioners answered. The practitioner suggested considering the count of developers who interact with each toggle and the last time each changed the code in the path of using the feature toggle. This is similar to the spreadsheet



(a) Difficulty of feature toggle management.



(b) Feature toggle usage increases technical debt.



(c) Extent of agreement for FT-heuristic. $\tilde{x}$ is median; ♦ is mean.

Figure 4.2: Summary of the practitioner survey.

that Chrome's developers use to record the owner of feature toggles (6). Future works could consider including this metric and corresponding heuristic.

Although we use a survey to evaluate the acceptance of FT-heuristics by practitioners, the main focus of Phase 2 of the methodology (Survey and Case Study) is on the case study to find the relation between the adoption of FT-heuristics and the values of FT-metrics.

4.5.2 Repository Inspection

We inspect each repository in the case study set and identify the toggles in its master branch. From 80 repositories in the case study set, 9 repositories have no feature toggle in their master branch. So 71 repositories are used for case study analysis. For each toggle, we manually compute FT-metrics, as described in Section 4.4, and the context metrics, as described in Section 4.2.3 in the last snapshot of each repository. We also manually identify if each repository follows the FT-heuristics using the following criteria and hypothesized dependent FT-metrics for each heuristic:

SharedMethod (H1): if the values of all toggles are checked using a shared value checking method; not applicable to repositories that check primitive values of toggles in conditional statements.

SelfDescriptive (H2): if the repository has at least two of the following three FT-metrics for the majority of the toggles:

1. intention-revealing names;
2. use of comments; or
3. and use of description.

Guidelines (H3): if the repository has guidelines to manage feature toggles.

UseSparingly (H4): if, based on an expert's (first researcher's) subjective judgment a feature toggle could be used in fewer files or locations.

AvoidDuplicate (H5): if the repository does not have a feature toggle duplicate code based on an expert's (first researcher) subjective judgment.

Testing (H6): if the feature toggles have associated test cases.

CompleteRemoval (H7): if there is no trace of toggles in the codebase for which there are associated commit message(s) referring to toggle removal.

We also checked the usage of available feature toggle management packages in these repositories. Only 13 of those use feature toggle management packages. In the remaining 67 repositories, the contributors implement their own feature toggle management approach.

4.5.3 FT-heuristics and FT-metrics (SRQ_{HM})

We now address SRQ_{HM} on the relation between the adoption of FT-heuristics and the values of the FT-metrics.

Tables 4.4, 4.5, and 4.6 summarize our results. In Table 4.4, # of repositories, # of contributors, # of feature toggles are context metrics, in Table 4.5, M1, M2, M7–M9 are numeric

FT-metrics, and in Table 4.6, M3–M6 and M10–M12 are binary FT-metrics. In all three tables 'F' is for repositories that follow a heuristic and 'NF' is for the repositories which do not follow that heuristic. For numeric metrics, the values for 'F' and 'NF' in Table 4.5 are the normalized average μ (except for M2 which is an absolute number). For example, the average number of files (M7) for the repositories that follow SharedMethod (H1) is 4.3 versus 4.2 for the repositories that do not follow H1. For binary metrics in Table 4.6, the values are the fraction of the repositories that have the metric in 'F' repositories or 'NF' repositories. For instance, 40% of the repositories that follow SharedMethod (H1) have test cases (M12) and 20% of the repositories that do not follow H1 have M12.

In Tables 4.5 and 4.6, the gray cells show the hypothetical relation between FT-heuristics and FT-metrics. The hypothesized relations are determined based on observations in Phase 1 of the methodology and iterative discussions between two researchers as shown in Table 4.2. The results of the case study may support these relations or show new relations.

In the following paragraphs, we analyze the results for each FT-heuristic based on Table 4.4, Table 4.5, and Table 4.6. The percentage improvements are calculated by Equation 4.1:

$$\text{Improvements} = \frac{\text{F} - \text{NF}}{\text{NF}} \times 100 \tag{4.1}$$

Improvements can be positive or negative. For example, for the FT-metric Guidelines (M3) in SharedMethod (H1) FT-heuristic, the improvement is 303.9%, i.e., the existence of guidelines in repositories that follow H1 is 303.9% more than those that do not follow H1. For the same FT-heuristic (H1), the improvement value for the number of the value-checking method (M2) is –93.7% which shows the number of the value checking methods in repositories that follow H1 is 93.7% lower, which is reasonable. The improvements are shown in Tables 4.4, 4.5, and 4.6 as Im.

SharedMethod (H1): We observe that repositories which follow H1 ($n = 26$) have more contributors and feature toggles compared to those which do not follow H1 ($n = 45$). In repositories that follow H1, from our hypothesized relationships, the number of value checking methods (M2) is 93.7% lower; but the number of files (M7), lines of code (M9), and presence of dead code (M11) do not have a significant difference compared to repositories not following H1. We unexpectedly observe the metric presence of guidelines (M3) is improved 303.9%, and the presence of test cases (M12) is improved 147.3% for repositories that follow H1. As we mentioned 13 repositories use feature toggle management packages.

Table 4.4: Observations from case study of 71 repositories for context metrics.

Context Metrics			H1(SharedMethod)	H2(SelfDescriptive)	H3(Guidelines)	H4(UseSparingly)	H5(AvoidDuplicate)	H6(Testing)	H7(CompleteRemoval)
n (# repositories)		F	26.0	19.0	10.0	53.0	57.0	17.0	21.0
		NF	45.0	52.0	61.0	18.0	14.0	54.0	13.0
# contributors		F	40.3	39.1	73.5	19.7	21.6	24.4	24.6
		NF	20.3	23.4	20.1	50.8	52.2	28.6	53.9
		Im	98.5	67.1	266.0	−61.2	−58.7	−14.7	−54.4
# feature toggles		F	19.5	20.1	27.7	12.0	10.9	7.0	14.8
		NF	9.5	10.7	10.8	16.8	22.6	15.2	26.2
		Im	105.0	87.5	156.0	−28.6	−52.0	−53.8	−43.5

Of these 13 repositories, 10 repositories have feature toggles in their master branch. From these 10 repositories, 8 repositories follow H1. This shows that having a shared method to check the value of feature toggle is an accepted heuristic for feature toggle management package providers.

SelfDescriptive (H2): We notice that the repositories that follow H2 ($n = 19$) have more contributors and more feature toggles compared to repositories without self-descriptive feature toggles ($n = 52$). From hypothesized relationships, the presence of comments (M5) and presence of descriptions (M6) are higher and improved by more than 3,000% in repositories that follow H2. In addition, we observe that these repositories have 41.4% less duplicate code (M10), and 31.6% less dead code (M11). Since having the intention-revealing names is one of the known coding practices (64), we observe that M4 is the most existed metric in all the repositories.

Guidelines (H3): Similar to SharedMethod (H1), repositories that follow H3 ($n = 10$) have a 266% more contributors and 156.0% more feature toggles. The metric presence of guidelines (M3) is hypothesized metric for H3 and it is better and 5,390% more in repositories that follow the heuristic. Another hypothesized metric is dead code (M11) which we observe that it is 92.6% lower in repositories that do not follow H3. Among non-hypothesized relationships, in repositories that follow H3, the metrics presence of duplicate code (M10)

Table 4.5: Observations from case study of 71 repositories for numeric metrics. Gray cells indicate the hypothetical relation between FT-heuristics and FT-metrics as mentioned in Table4.2.

			H1(SharedMethod)	H2(SelfDescriptive)	H3(Guidelines)	H4(UseSparingly)	H5(AvoidDuplicate)	H6(Testing)	H7(CompleteRemoval)
Numeric Metrics	M1 (Paths)	F	1.4	1.7	1.7	1.3	1.3	1.9	2.0
		NF	1.5	1.4	1.4	2.0	2.2	1.4	1.0
		Im	−8.7	19.2	21.0	−33.4	−41.8	38.6	111.4
	M2 (Methods)	F	1.1	9.6	1.0	6.3	5.1	2.9	2.2
		NF	17.0	3.7	6.6	3.1	6.8	7.0	5.6
		Im	−93.7	156.6	−84.8	102.4	−25.1	−58.0	−60.3
	M7 (Files)	F	4.3	4.9	5.2	3.4	4.0	5.0	5.2
		NF	4.2	4.0	4.1	6.9	5.1	4.0	4.8
		Im	3.5	20.7	28.2	−51.1	−20.2	26.3	7.7
	M8 (Locations)	F	5.7	6.0	7.3	5.0	5.5	8.1	7.3
		NF	5.9	5.7	5.6	8.1	7.1	5.1	5.6
		Im	−2.5	4.4	31.6	−37.8	−23.2	60.6	31.7
	M9 (LOC)	F	7.1	5.5	4.7	5.8	5.9	13.7	10.8
		NF	5.7	6.5	6.5	7.4	7.5	3.8	4.8
		Im	24.7	−14.3	−26.6	−21.4	−20.9	257.8	127.4

is on average 232.7% higher . So, we observe that mere having documented guidelines or using issues or pull requests for structuring feature toggles does not necessarily prevent the occurrence of dead code and duplicate code resulting from feature toggle usage.

UseSparingly (H4): In contrast to SharedMethod (H1) and Guidelines (H3), H4 is followed more in repositories ($n = 53$) with a lower number of contributors and a lower number of the feature toggles. From hypothesized relationships, the repositories which follow H4 have 51.1% lower number of files (M7), and 37.8% lower number of locations (M8). For the rest of the metrics, nothing specific is observed except for the presence of duplicate code (M10) which is 61.8% lower by following H4.

AvoidDuplicate (H5): We observe that repositories that follow H5 ($n = 57$) on average have

Table 4.6: Observations from case study of 71 repositories for binary metrics. Gray cells indicate the hypothetical relation between FT-heuristics and FT-metrics as mentioned in Table4.2.

			H1(SharedMethod)	H2(SelfDescriptive)	H3(Guidelines)	H4(UseSparingly)	H5(AvoidDuplicate)	H6(Testing)	H7(CompleteRemoval)
Binary Metrics	M3 (Guidelines)	F	0.3	0.2	0.9	0.2	0.1	0.2	0.2
		NF	0.1	0.1	0.0	0.1	0.3	0.1	0.3
		Im	303.9	17.3	5390.0	35.9	−63.2	36.1	−38.1
	M4 (Intention)	F	0.9	1.0	1.0	1.0	1.0	1.0	1.0
		NF	1.0	0.9	1.0	0.9	0.9	0.9	1.0
		Im	−5.6	6.1	5.2	10.4	3.9	5.9	0.0
	M5 (Comments)	F	0.2	0.6	0.3	0.2	0.1	0.1	0.2
		NF	0.1	0.0	0.1	0.2	0.2	0.2	0.0
		Im	44.2	–	128.8	−9.4	−34.5	−29.4	–
	M6 (Description)	F	0.2	0.6	0.3	0.2	0.2	0.1	0.2
		NF	0.2	0.0	0.2	0.2	0.1	0.2	0.1
		Im	48.4	3184.2	83.0	−23.6	194.7	−73.5	209.5
	M10 (Duplicate)	F	0.2	0.2	0.6	0.2	0.1	0.3	0.3
		NF	0.2	0.3	0.2	0.4	1.0	0.2	0.4
		Im	−5.6	−41.4	232.7	−61.8	−94.7	−32.4	−13.3
	M11 (Dead)	F	0.4	0.3	0.6	0.3	0.4	0.3	0.1
		NF	0.3	0.4	0.3	0.4	0.4	0.4	0.9
		Im	15.4	−31.6	92.6	−12.7	−1.8	−20.6	−89.7
	M12 (Test cases)	F	0.4	0.1	0.4	0.3	0.2	1.0	0.4
		NF	0.2	0.3	0.2	0.2	0.3	0.0	0.3
		Im	147.3	−63.5	87.7	10.4	−20.2	–	23.8

fewer contributors and a lower number of feature toggles compared to repositories that do not follow H5 ($n = 14$). We note that in repositories that follow H5, from hypothesized relationships, the presence of duplicate code (M10) is 94.7% less. In addition, in these repositories, the number of paths (M1) is 41.8% less. However, the presence of guidelines (M3) is 63.2% lower compared to repositories that do not follow H5.

Testing (H6): We notice that repositories ($n = 17$) with a lower number of feature toggles follow H6. H6 has one hypothesized relationship with FT-metric presence of test cases (M12), which is obviously better in repositories that follow H6. We observe that the number of locations (M8) is 60.6% more and lines of code (M9) are 257.8% more for repositories that follow H6 compared to those that do not follow H6 . Larger values for M8 and M9 are reasonable as having test cases increase the number of the locations and lines of code related to feature toggles.

CompleteRemoval (H7): We observe that repositories with a larger number of contributors and a larger number of feature toggles do not follow the H7. From hypothesized relationships, repositories that follow H7 have less dead code (M11) by 89.7%. Among non-hypothesized relationships, the use of comments (M5) is higher (non of the not following repositories use comments), the number of paths (M1) is 111.4% higher, and the number of value checking methods (M2) is 60.3% lower .

4.6 Threats to Validity

Internal validity. We searched GitHub for the keyword “feature toggle” and checked only the first 400 search results. We may have missed repositories that use feature toggles in their development process. Developing FT-heuristics and identifying FT-metrics could be subjective to the first researcher’s knowledge. To mitigate this threat, the second researcher critically reviewed the FT-heuristics and FT-metrics and give feedback. We also conducted a survey to evaluate the findings with practitioners of GitHub repositories in our dataset. Although FT-heuristics and FT-metrics cover the lifecycle of a feature toggle from design to clean-up and our process was iterative and involved more than one person, we do not claim the completeness of our findings. Future works could find more heuristics and metrics.
In the case study, we only discuss the trends in improvement in FT-metrics when following FT-heuristics. In Appendix B, we report on our findings from preliminary statistical analyses of the case study data. The statistical findings could be strengthened with replication on a larger dataset.

External validity. We use open source repositories from GitHub for our study. Including repositories from other organizations, such as proprietary organizations, may change the results of our study. 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 stronger evidence of generalization.

Construct validity. Incorrect classification of a code snippet in the 80 GitHub repositories as a feature toggle could render our results invalid. To mitigate this threat, the first researcher critically examined each feature toggle implementation in consultation with the second researcher.

4.7 Lessons Learned

Our survey respondents—practitioners who routinely use feature toggles, agree with (1) the difficulty of managing feature toggles; (2) the increase of the technical debt by using toggles; and (3) the FT-heuristics can be used to guide practitioners on how to structure and use feature toggles to reduce technical debt.

Based on our case study and survey observations, we note:

SharedMethod (H1). Using *shared method* is more common in repositories with guidelines and test cases compared to those without guidelines and test cases. However, we did not observe any meaningful difference in the number of files, lines of code, and dead code when following this heuristic compared to not following it.

SelfDescriptive (H2). Having *self-descriptive feature toggles* helps in preventing duplicate code and dead code in a repository. We did not hypothesize the relation between self-descriptive feature toggles and metrics duplicate code and dead code but we find these metrics to be lower with repositories having self-descriptive feature toggles.

Guidelines (H3). Providing *guidelines* to manage feature toggles may not necessarily reduce duplicate and dead code. Practitioners may mandate guidelines by other means such as code review.

UseSparingly (H4). Using *toggles sparingly* is the second most followed FT-heuristic in our case study set. Doing so reduces duplicate code, as a non-hypothesized metric.

AvoidDuplicate (H5). Avoiding *duplicate code* in using feature toggles is the most followed FT-heuristic in our case study set. This shows that developers are aware of the negative effects of having duplicate code for feature toggles as other parts of the code.

Testing (H6). Although surveyed practitioners agree on having *Test cases* for feature toggles, this heuristic is second least followed in our case study. Finding the effect of the lack of test cases for feature toggles is a direction for future studies.

CompleteRemoval (H7). *Complete removal* of a feature toggle received the highest agreement by survey respondents. Following the FT-heuristic on ensuring complete removal of feature toggles reduces the presence of the dead code.

As a result, we suggest practitioners create self-descriptive feature toggles (H2), use feature toggles sparingly (H4), avoid duplicate code in using feature toggles (H5), and ensure complete removal of a feature toggle (H7). Practitioners may follow FT-heuristics without measuring FT-metrics. However, we strongly suggest practitioners consider four FT-metrics: number of feature toggle value checking methods (M2), presence of guidelines (M3), presence of duplicate code (M10), and presence of test cases (M12).

CHAPTER

5

COMPREHENSIVE MODEL OF SOFTWARE CONFIGURATION (MSCV2)

5.1 Motivation

Every family of research can benefit from a framework or common ontology to enable meta-analysis on them. Researchers show the necessity of having a common ontology to exchange and keep organized knowledge to reuse in software engineering research previously (102; 103). To provide this common ontology, researchers proposed evaluation frameworks in different software engineering scopes such as extreme programming (XP) (37; 38) and software security practices (39) to enable comparing case studies and results in related publications. Morrison (39) states that evaluation frameworks can help organizing empirical studies to a body of knowledge that supports repeating the studies and future project planning. Runenson et al. (40) explains that using a theoretical framework to design case study research in software engineering makes the context of the research clear, and helps researchers to conduct the study and other researchers to review the results. Hence, researchers can benefit from a *model of software configuration* to define the configuration

in their research studies clearly, and help other researchers to use their results in other studies (12).

To provide terminology for software configuration research, Siegmund et al. (12) derived a Model of Software Configuration (MSC) consisting of eight dimensions (characteristics), with more than one value attached to each dimension , by performing interviews and analyzing academic publications. To derive their MSC, Siegmund et al. (12) focused on the configuration options concept more than feature toggles, and all the analyzed publications are related to configuration options. In another study, Meinicke et al. (11) explored similarities and differences between feature toggles and configuration options by conducting semi-structured interviews with feature toggle experts and considered configuration options literature. We use these two publications as the basis of our research study.

The goal of this research study is to aid researchers in conducting a family of research on software configuration by extending an existing model of software configuration that provides terminology and context for research studies. We will call the *extended Model of Software Configuration* as MSCv2 in the rest of this dissertation. Accordingly, we state the following research question with two sub-research questions:

RQ3: Can we extend the existing Model of Software Configuration to cover feature toggle concept as well as configuration option?

RQ3.1: What dimensions and values need to be added to extend the MSC to MSCv2 to cover feature toggles as well as configuration options?

RQ3.2: Can the MSCv2 be used to model software configuration in research publications and in industrial systems?

5.2 Methodology

Figure 5.1 describes our methodology to answer our research questions. Our methodology consists of three phases. In this section, we explain each phase in detail [1].

5.2.1 Meinicke et al.: Feature Toggles vs. Configuration Options

To better understand feature toggles and specify their relation to configuration options, Meinicke et al. (11) conducted nine semi-structured interviews with practitioners who

[1]Data availability: The data including the list of feature toggle management system repositories, links to the selected repositories, list of related academic research publications, and extracted codes used for this research study is available at `https://figshare.com/s/16881bbd5614795e266f`

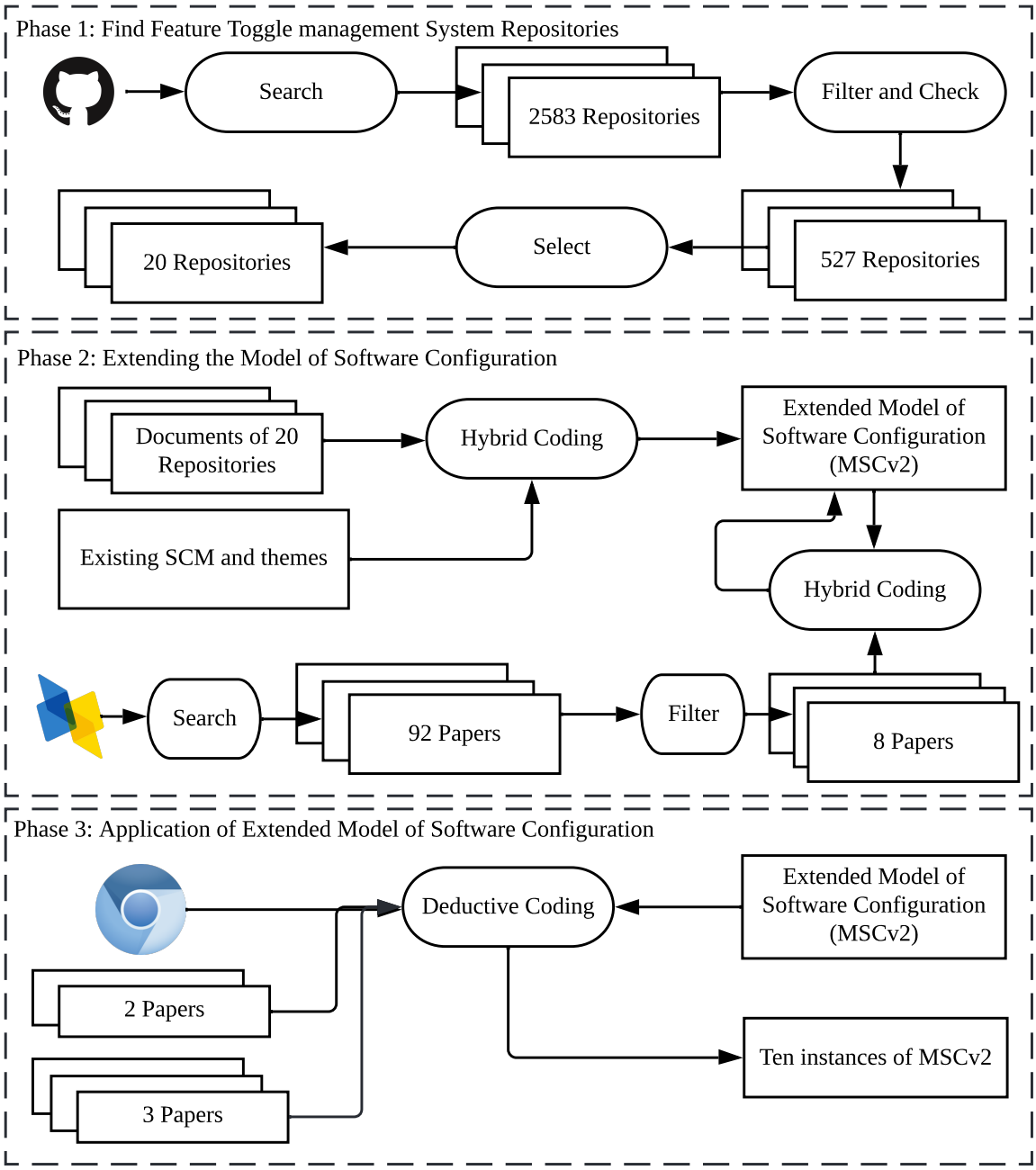


Figure 5.1: Research methodology outline.

were feature toggle experts. By analyzing the interview scripts, they identified ten themes that explain the differences between configuration options and feature toggles. They also reflected that “depending on one’s perspective, configuration options can be seen as a

special subset of feature toggles or one can see feature toggles as using configuration options for a new purpose." Hence, we consider feature toggles and configuration options to be part of the same branch of knowledge. We use Meinicke et al.'s themes as a basis for our research study.

5.2.2 Siegmund et al.: Model of Software Configuration (MSC)

To provide software configuration terminology for research studies and help practitioners to find possible challenges, Siegmund et al. (12) derived the MSC that consists of eight dimensions for software configuration with 47 values in total. *Dimensions* are high-order topics related to configurations and one or more *values* can be assigned to each dimension. They used a mixed-methods approach. First, they interviewed 11 practitioners, such as developers, team leads, and senior software engineers. From the collected interviews' data, they created an initial model of configuration. Their initial model includes seven dimensions with 32 values. Then, they analyzed two related publications to their study to validate their results and complete their model. As a result, they found one additional dimension and 15 new values. To verify the applicability of their developed model and find gaps in the area, they applied their model to 16 *configuration option*-related publications.

We use MSC as a basis for our research study and extend it explicitly to support both feature toggle and configuration options which we refer to jointly as *software configuration*.

5.2.3 Phase One: Find Feature Toggle Management System Repositories

A feature toggle management system is a platform that helps practitioners to define, implement, integrate, and manage feature toggles in their code. Feature toggle management systems often provide services, such as APIs, an admin UI, data storage for toggles, and auditing logs. To find open source feature toggle management system repositories, on 14th April 2021, we searched GitHub using the search API with the following keywords (feature toggles are also called feature flags, feature switches, and feature flips in literature (8)): "feature toggle" (1,065 repositories), "feature flag" (1,045 repositories), "feature switch" (355 repositories), and "feature flip" (118 repositories). We used these keywords because feature toggles are also called feature flags, feature switches, and feature flips in literature (8). In total, we retrieved 2,583 repositories. To filter the list of feature toggle management repositories, we follow the steps below (the numbers in parenthesis are the number of the repositories after each step):

- First, we removed duplicated repositories (2,387 repositories);
- We removed repositories with zero stars (910 repositories);
- We removed repositories with no update in the last five years, after 2015-12-31 (765 repositories);
- We manually read the name and description of all 765 repositories, and removed unrelated repositories (548 repositories);
- Then, we removed repositories that were archived by the owners (527 repositories).

In the end, we ended up with 527 feature toggle management repositories on GitHub. Table 5.1 shows the programming languages of these repositories.

Table 5.1: Programming language of the identified repositories.

	Language	# repositories
Top Five	JavaScript	110
	PHP	62
	TypeScript	61
	C#	52
	Ruby	49
Other languages		193
Total		527

5.2.4 Phase Two: Extending the Model of Software Configuration (MSCv2)

In Phase 2, we extend the MSC by focusing on feature toggle-related resources including the documentation from the feature toggle management system repositories, and related academic publications.

To select a subset of Phase 1 results for document analysis, we focused on the top 5 programming languages with greatest number of repositories shown in Table 5.1. We sorted repositories based on their stars. Then we selected the first, second, third, and fourth-ranked repositories for each one of the 5 programming languages, and analyzed their documentation as discussed below. We did not find any new values from the fourth-ranked repositories for each programming language, so we stopped. We analyzed 20 repositories in total (104; 105; 106; 107; 108; 109; 110; 111; 112; 113; 114; 115; 116; 117; 118; 119; 120; 121;

122; 123).

To extend the MSC, two researchers analyzed the available resources from each one of the 20 selected repositories including the README files, documentation, videos, and websites.

We used a hybrid coding approach including *deductive coding* and *inductive coding*. Coding is a technique for qualitative analysis of the text by assigning words or short phrases as codes to parts of the text (43). Deductive coding is an approach that codes are developed ahead to guide the coding process, and inductive coding is an approach that codes are developed while analyzing the text (124). In our hybrid coding approach: (1) We used Siegmund et al.'s dimensions, and Meinicke et al.'s themes as predefined codes (deductive coding), (2) We assigned the predefined codes to the text of documentation for each repository, (3) At the same time, we assigned new codes to any parts of the text that was not covered by predefined codes (inductive coding), (4) In step 2 and 3 we recorded values for each code (codes are dimensions or themes, and each one has possible values), (5) At the end, we selected most relevant new codes from step 3 that explain high-level aspects of feature toggles, and added them to the existing codes as new dimensions, and create MSCv2. Two researchers performed steps 1 to 5 separately, then they discussed their findings at the end of the analysis and resolved disagreements in their codes. During this process, the first researcher found one possible new dimension, and the second researcher found two possible new dimensions. After discussion, the researchers agreed on one of the new dimensions and discard the other two. The Cohen's Kappa (125) agreement scores on values of existing dimensions are 0.77, 0.58, 0.45, 0.28, 0.43, 0.47, 0.8, 0.33 in order for dimensions listed in Section5.3. All the scores are between "fair agreement" and "substantial agreement". During this analysis, we found one new dimension and 18 new values.

After analyzing the repositories' documentation, we use MSCv2 to model software configuration of relevant feature toggle academic publications to evaluate the comprehensiveness of the model and to find new dimensions and values. To find feature toggle relevant publications, we searched DBLP (Digital Bibliography & Library Project) on 22nd March 2022 with the following keywords: "feature toggle" (9 with 2 duplicates), "feature flag" (13), "feature switch" (75), and "feature flip" (6). Two researchers read the peer-reviewed publications titles, and select the relevant ones to our research study. For any publication with unclear title, we read the abstract of the publication to find out if it is relevant to feature toggles in the software development area or not. In total, we found nine relevant publications. All of these nine publications are explained in Chapter 2. One publication is one of the basic studies that we used to extend the model (11) and is explained in Section 5.2.1.

Two researchers, (1) Read each one of the eight publication, (2) Performed deductive coding on them with the list of codes (dimensions) they found in documentation analysis, (3) Recorded any values mentioned for dimensions, and (4) Performed inductive coding if there is any new aspect that was not covered by predefined codes. Two researchers discussed their results and resolved disagreements. The Cohen's Kappa agreement scores (125) for values of dimensions are 0.77, 0.40, 0.22, 0.82, 0.31, 0.33, 0.41, 0.51, and 0.06 in order for dimensions listed in Section 5.3. All the scores are between "slight agreement" and "substantial agreement". During this analysis, we did not find any new dimension, but six new values added to the MSCv2. Our process showed that the software configuration can be modeled systematically in feature toggle-related publications, and other researchers can compare the modeled configuration with the configuration in their research.

5.2.5 Phase Three: Application of Extended Model of Software Configuration (MSCv2)

Researchers and practitioners can follow the steps explained in the following subsections 5.2.5 and 5.2.5 to apply MSCv2 to their research studies' or industrial systems' design and evaluation. *Application* means creating an instance of MSCv2 with appropriate dimensions and values.

Academia

We analyzed two sets of academic publications in this step. The first set includes two feature toggle publications on Chrome data (6; 7), and the second set includes three publications asking similar research questions about practitioners' practices in using software configuration, (6; 5; 4) two of them focused on feature toggles and one of them focused on configuration options.

For all five publications, two researchers, (1) Read each one of the five publications, (2) extracted parts of the text explaining characteristics of their configuration in their research scope including the definition of feature toggle in the literature and in industrial systems they used, (3) Performed deductive coding on the extracted text with the list of dimensions and values found in Phase 2, and (4) visualized the result for better understanding.

After modeling the configuration using MSCv2, we compare the models in each set of publications, and discuss our observations in Section 5.4.

Industry

To show the applicability of MSCv2 in practice, we modeled configurations as it is defined in Chrome repository. Two researchers, (1) Searched *docs* directory of Chrome repository by using the following keywords: “feature toggle”, “feature flag”, “feature switches”, “feature flips”, and “configuration options”, (2) Found the file[2] explaining five categories of configurations in their repository: prefs, flags, settings, features, switches, (3) Performed deductive coding on the definitions on this file with the list of dimensions and values found in Phase 2, and assign values for each dimension for each category, (4) Follow links in the file to other files, and found more documentation for each category, (5) Performed steps (3) and (4) until no new value found for dimensions, and (6) visualized the five models for better understanding. After modeling the five configuration, we discuss our observations in Section 5.4.

5.3 The Extended Model of Software Configuration (MSCv2)

Based on Phase 2 of our methodology, we extended the MSC that covers both feature toggles and configuration options concepts. Other than dimensions in (12) and themes in (11), our analysis of feature toggles revealed one new dimension with 14 values. We also identified 10 additional values for the existing dimensions from basis research studies. Figure 5.2 [3] shows the MSCv2. The blue items in this figure are the new dimensions and new values that we have added to Siegmund’s model of configuration (answer to RQ3.1).

In the following subsections, we describe the dimensions in MSCv2.For each dimension, first we define the dimension (*Definition*). The definition can come from (12), (11), or from our analysis. Next, we explain the possible values based on two basic studies and our analysis of documents from feature toggle management systems’ repositories (*Model Extension*). Then, we explain the result of the application of MSCv2 to feature toggle-related publications to find missed dimensions and values (*Model Completion*). Lastly, we summarize each dimension in a box. If the dimension or values came from (12), we marked them with a S1; from (11) with a S2; and from our observations and analysis with a S3 identifiers.

[2]https://github.com/chromium/chromium/blob/main/docs/configuration.md

[3]In Chapter 6, we found “Documentation” and “(UML) Models” as new Artifact values, “Platform” as a new Activation strategy value, and “Dependency” as a new Complexity value when the level of dependency is not clear. We show these values as blue and italic in this figure.

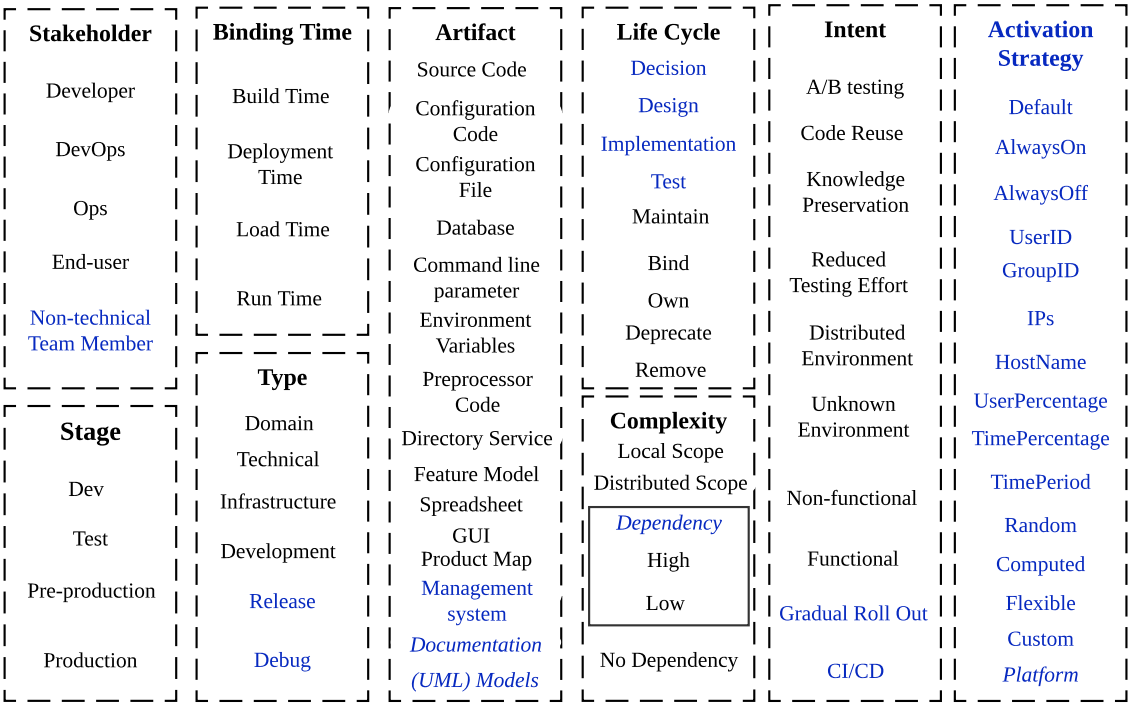


Figure 5.2: Extended Model of Software Configuration (MSCv2).

5.3.1 Stakeholder (S1, S2, S3)

Definition. Based on Siegmund et al. (12), the stakeholder is anyone "who deals with configuration". Meinicke et al. (11) provides better definition as the person who is making configuration decisions. We select the second definition for this dimension.

Model Extension. In the Siegmund et al.'s definition, *Developer*, *DevOps*, *Ops*, and *End-user* can change the software configurations of the system. In their study, users can configure a system based on their needs. However, based on our analysis of repositories' documents, *End-user* is not a type of stakeholder in feature toggle usage. We have observed *Developer*, *DevOps*, and *Ops* as stakeholders in the analyzed documentation, but never observed the possibility for users to change the value of the toggles or manage them. The different purposes of using feature toggles and configuration options can explain this observation. Configuration options allow *End-user* to change the system based on their needs. In addition, Meinicke et al. (11) mentioned that one of the key differences between feature toggles and configuration options is the person who makes configuration decisions. They mentioned that feature toggles are controlled by *Developers* and *Ops*, and configuration options

are controlled by *End-users* most of the time. However, they found that feature toggles are sometimes exposed to *End-users* in experimental usage of the toggles. They observed that this may cause testing and removing challenges for the development team. One example of exposing feature toggles to users is chrome://flags/ where users of Chrome can enable or disable features themselves. This is one of the reasons that the distinction between feature toggles and configuration options is blurred. So, using the MSCv2 can help researchers and practitioners to understand the configuration space clearly, without relying just on their names. Moreover, while we analyze the documents for stakeholders, other than existing values, we have identified a new value: *Non-technical Team Member* (106), which points to give permission to non-technical team members to be able to release changes while using feature toggles.

Model Completion. We have observed all the values for this dimension in analyzed publications. About the new identified value of *Non-technical Team Members*, one of the practices Mahdavi-Hezaveh et al. (5) identified in the feature toggle usage practices used by practitioners is "Give access to team members". Based on practitioners' experiences giving permission to Q&A team members, sales team members, and product managers helps to prevent management bottleneck of feature toggles (5).

Stakeholder (S1, S2, S3). A stakeholder is a person who is making configuration decisions. The values for this dimension are: Developer (S1, S2, S3), DevOps (S1, S3), Ops (S1, S2, S3), End-user (S1, S2), and Non-technical Team Member (S3).

5.3.2 Stage (S1, S3)

Definition. Based on (12) stage "describes that configuration happens in different stages of the development process".

Model Extension. Siegmund et al. identified four values as stages: Dev, Test, Pre-Production, Production.

In our analysis of GitHub repositories, we observed all four stages for the feature toggle usage. Using release toggles for dark launches, and trunk-based development points to *Production* stage, and using them for gradual roll out points to *Pre-production* stage. Ops toggles and permission toggles are used in *Production* stage. Development toggles are used in *Dev*, and experiment toggles are used in *Test* stage.

Model Completion. Our analysis of academic publications confirms our observation in GitHub repositories' documentation.

Stage (S1, S3). Stage dimension describes what configuration happens in what stages of the development process. The values for this dimension are: Dev (S1, S3), Test (S1, S3), Pre-Production (S1, S3), Production (S1, S3)

5.3.3 Binding Time (S1, S3)

Definition. Based on (12), binding time points to the event of binding a value to a software configuration.

Model Extension. Siegmund et al. identified four binding times: Build Time, Deployment Time, Load Time, and Run Time. *Build Time* configurations are used in form of build scripts, *Deployment Time* configurations are used as configuration files, and *Load Time* and *Run Time* configurations are used in form of properties files, databases, command-line arguments, and user interfaces. The binding time of software configurations in a system will affect the implementation and design of the configuration. Based on our analysis of GitHub repositories' documentation, *Run time* is the most used (but not the only) binding time for feature toggles. In contrast, we have not observed any *Build time* as the binding time for feature toggles.

Model Completion. In the analyzed publications, we have the same observation about the binding time of feature toggles. Authors of all publications except one (14) mentioned *Run time* as the binding time for feature toggles.

Binding Time (S1, S3). Binding time is the time of binding a value to a software configuration. The values for this dimension are: Build time (S1), Deployment time (S1, S3), Load Time (S1, S3), Run Time (S1, S3).

5.3.4 Type (S1, S3)

Definition. Siegmund et al. (12) define type of configuration as "what should be configured".

Model Extension. The values Siegmund et al. identified for this dimension are: Domain, Technical, Infrastructure, and Development. *Domain* configurations are used to change software based on the user's requirements. However, *Technical* configurations that contains *Infrastructure* and *Development* types focus on environment of the system and its deployment process. *Infrastructure* configuration is used to adjust software to related hardware and software, such as database system and used ports. *Development* configuration refers to setting-up development tools such as IDEs, build tools, building and testing process, and automating the deployment process.

In the analyzed GitHub repositories documentations, we observe *Domain* type (in

all repositories) and *Infrastructure* type (in (111)), but no mentioning of *Development* configuration type. We also observed the need to define new values for *Type* dimension to cover all types of feature toggles provided by these management systems, such as release toggles to enable trunk-based development (106).

Model Completion. Based on literature in the analyzed academic publications, feature toggles have 5 types:

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 for premium users, and
5. *development toggle* to turn on or off developmental features to test and debug code.

Considering the definitions of mentioned feature toggles' types, experiment toggles and permission toggles can be considered as *Domain* configurations. Ops toggles can be a kind of *Development* configuration. But none of the provided configuration types cover release toggles and development toggles. We also have observed the need for a new value for release toggles while we analyzed feature toggle management systems' documentation. Hence, we define two new values for Type:

1. *Release* to cover release toggles as configurations that hide incomplete implementation of a function from user.
2. *Debug* to cover development toggles that use to test and debug code.

Type (S1, S3). Type is the dimension of what is configured in the system. The values for this dimension are: Domain (S1, S3), Technical (S1, S3), Infrastructure (S1, S3), Development (S1), Release (S3), and Debug (S3).

5.3.5 Artifact (S1, S2, S3)

Definition. Configuration Artifacts are development artifacts that contain the configuration (12).

Model Extension. Based on Siegmund et al.'s analysis the 12 artifacts are: Source Code, Configuration Code, Configuration File, Database, Command line parameter, Environment Variable, Preprocessor Code, Directory Service, Feature Model, Spreadsheet, Product Map, GUI. *Configuration File* points to files such as properties files and .ini files which are easy

to understand and change. *Configuration Code* points to more complex artifacts such as build scripts, container descriptions (such as Docker files), and automation scripts (such as Ansible playbook or Jenkins files).

Meinicke et al. has three themes named *documentation*, *constraints*, and *dependencies* that are related to artifact dimension. They found that configuration options are documented in *Feature Models*, because they have constraints and are highly dependent on each other. However, their interviewees emphasize that constraints on feature toggles are none or few, dependencies between feature toggles are at most nesting or grouping, and configuration knowledge is spread across *Source Files*, and *Configuration Files*.

Based on our GitHub repositories' analysis, we have not observed any of the following artifact types: *Configuration Code*, *Preprocessor Code*, *Directory Service*, *Feature Model*, *Spreadsheet*, and *Product Map*. As we already mentioned, some of them are specific to configuration options such as *Feature Model*. However, some of them can be a kind of feature toggle artifact such as *Spreadsheet*. Chrome developers are using a spreadsheet to record the name, the feature set file name, the owner of the toggle, the status of the toggle, related bugs, and comments on the purpose of using the toggle (126).

Model Completion. In the analyzed research studies, feature toggle management systems are often used by practitioners to implement and manage feature toggles in their software code to control the added complexity and technical debt of adding feature toggles (5; 30; 14). Feature toggle management system is a platform that help practitioners to define, implement, integrate, and manage feature toggles to their code. Feature toggle management systems often provides services such as APIs, a UI, data storage, and auditing logs. We add this artifact as a value to this dimension.

Artifacts (S1, S2, S3). Configuration Artifacts are development artifacts that contain the configuration. The values for this dimension are: Source Code (S1, S2, S3), Configuration Code (S1), Configuration File (S1, S2, S3), Database (S1, S3), Command line parameter (S1, S3), Environment Variable (S1, S3), Preprocessor Code (S1), Directory Service (S1), Feature Model (S1, S2), Spreadsheet (S1), Product Map (S1), GUI (S1, S3), Management system (S3).

5.3.6 Life Cycle (S1, S2, S3)

Definition. The life cycle of a configuration is defined as various aspects of its lifetime from creation to removal (12).

Model Extension. Siegmund et al. (12) identified six values for the life cycle of a configu-

ration: Create, Maintain, Bind, Own, Deprecate, and Remove.

Meinicke et al. (11) identified *removal of configurations* as one of the themes. They mentioned that removal is more important and more frequent when practitioners use feature toggles, because of their short-term usage goals. On the other hand, configuration options tend to be permanent in the code, so the practitioners do not remove them from the code. Meinicke et al. also emphasized the importance of documenting *owner* of the feature toggles in the documentation. Testing software configurations is one of the themes in Meinicke et al. study. They stated that testing is an important task when having feature toggles and configuration options. However, the strategies may differ. Configuration options have interaction with each other so the testing of their combinations should be done, but feature toggles have rare interactions, and testing them separately is enough based on practitioners' experiences. Hence, *test* should be the new value for life cycle dimension.

While analysing GitHub repositories' documentations, we observe all the life cycle values including the new value of *test*. *Own*, *Deprecate*, and *Remove* are less frequent, and none of the packages talk about all life cycle values.

Model Completion. Mahdavi-Hezaveh et al. (5) provided a lifecycle for feature toggle that contains Decision, Design, Implementation, Existence, and Clean-up. Siegmund et al. mentioned that the life cycle of a configuration option starts when a new optional feature is planned (12). The decision, design, and implementation steps in the feature toggle life cycle are covered by the *Create* value for the life cycle dimension. Existence is covered by *Bind*, *Maintain* and *Own*. The clean-up step in feature toggle life cycle is covered by *Deprecate*, and *Remove*. So, the life cycle of feature toggles is covered by the values for this dimension identified by Siegmund et al.. However, we suggest to split *Create* to *Decision*, *Design*, and *Implementation*. The reason is that decision making and designing feature toggles are very important steps in using feature toggles that help to control the added complexity to the code and plan to remove the toggles ahead of time (5).

Rahman et al. (6), and Mahdavi-Hezaveh et al. (5) mentioned that one way to remove feature toggles from code is to make the toggle a permanent configuration option in the code. So, not all the feature toggles have *Deprecate*, and *Remove* in their life cycle, and feature toggles can turn into configuration options at the end of their life time.

Another observation from academic publications is the importance of testing feature toggles (6).

Life Cycle (S1, S2, S3). The life cycle of a configuration is defined as various aspects of its lifetime from creation to removal. The values are: Decision (S1, S3), Design (S1, S3), Implementation (S1, S3), Test (S2, S3), Maintain (S1, S3), Bind (S1, S3), Own (S1, S2, S3), Deprecate (S1, S2, S3), and Remove (S1, S2, S3).

5.3.7 Complexity (S1, S2, S3)

Definition. We follow the definition of Siegmund et al. for complexity. Siegmund et al. (12) consider two aspects of using software configuration to define the complexity: scope and dependency. The scope depends on the impact of the configuration on one or more modules with two possible values: *Local scope* and *Distributed scope*. For dependency, based on the degree of dependency of a configuration to other configurations with three possible values: *High dependency*, *Low dependency*, and *No dependency*.

Model Extension. From the themes in Meinicke et al.'s (11) study, *complexity*, *dependencies*, *feature traceability*, and *feature interactions* fell into this dimension. They reported that practitioners try to keep the scope of feature toggle's impact as *Local Scope* in single modules. The practitioners also stated that dependencies among feature toggles are rare and interactions between them are not important, but configuration options have many dependencies often in hierarchical groups. They also found that the complexity of using configuration options is high, but for feature toggles depend on the number of them.

From the analyzed GitHub repositories' documentation, we have observed *Local* (in all) and *Distributed* (104) scopes. The practitioners did not discuss the dependency between feature toggles in their documentation because the dependency is introduced while using them in the code. So, all three levels of dependency are possible.

Model Completion. We observed all values of this dimension in analyzed publications.

Complexity (S1, S2, S3). Complexity is the scope and dependency between software configurations. The values for this dimension are: Local scope (S1, S2, S3), Distributed scope (S1, S3), High dependency (S1, S2), Low dependency (S1, S2), and No dependency (S1, S2).

5.3.8 Intent (S1, S2, S3)

Definition. Siegmud et al. (12) identified a dimension named *intent*. The authors defined *intent* as the purpose of using software configuration. Meinicke et al. identified *goal* as a theme that is defined same as *intent*. We use their definitions for *intent* dimension.

Model Extension. From our analysis of GitHub repositories' documentations, we observed that feature toggles are used for A/B testing (110), gradual roll out (canary release) (106), turning on and off a feature (104), targeted release (104), implementing trunk-based development (prevent long-lived branches) (106), release new feature faster (106), and use as kill switches (106).

Siegmund et al. identified the following values as goals of using software configuration: A/B Testing, Code Reuse, Knowledge Preservation, Reduced Testing Effort, Distributed Environment, Unknown Environment, Non-Functional, and Functional. From these identified values, *A/B testing* can cover A/B testing for feature toggles. *Functional* value can cover turning on and off a feature, releasing new features faster, and use as kill switches from feature toggles usage purposes. The rest of the intentions of feature toggle usages are not covered by the values for this dimension.

Meinicke et al. distinguish three main goals: hiding incomplete implementation, experimentation and release, and configuration. From these goals and based on their definitions, *hiding incomplete implementation* can cover implementing trunk-based development, *experimentation and release* can cover A/B testing, and gradual rollout, *configuration* is same as *functional* value in Siegmund's values. Also, Meinicke et al. stated that the goal behind the feature toggle can be changed over time. For example, an experiment feature toggle with the goal of A/B testing can become a configuration option with the goal of changing functionality based on user requests. Other researchers also mentioned this situation in their studies (6; 5).

Model Completion. Other than the intents from feature toggle management systems' documents, we observed that feature toggles are also used for continuous delivery, context switching, migration from an environment to another environment, and blue/green deployment in analyzed publications (6; 31). The context switching, when a developer is working on a feature and an emergency bug fixes came up for another feature, is covered by *knowledge preservation*. Migrating from one environment to another environment and blue/green deployment are covered by *Unknown Environment*.

Based on new observed values in the feature toggle management systems' documents, goals in (11), and analyzed publications, we define the following new values for this dimension:

1. *Gradual Roll Out*: when a feature will be exposed to a subset of users and gradually is made available for all users' systems. This value covers gradual rollout and targeted releases in the list of feature toggle usage intents,

2. *Continuous Integration/Continuous Delivery*: when using toggles helps to implement trunk-based development and prevents having long-lived branches. Moreover, the development team can release the system rapidly even with a long-term feature development and help them to do dark launches.

Intent (S1, S2, S3). The intent is the purpose of using software configuration. The values for this dimension are: A/B Testing (S1, S2, S3), Code Reuse (S1), Knowledge Preservation (S1, S3), Reduced Testing Effort (S1), Distributed Environment (S1), Unknown Environment (S1, S3), Non-Functional (S1), Functional (S1, S2, S3), Gradual Roll Out (S2, S3), Continuous Integration/Continuous Delivery (S2, S3).

5.3.9 Activation Strategy (S3)

Definition. We define *Activation Strategy* as a dimension that explains rules to activate feature toggles.

Model Extension. While we analyzed the GitHub repositories' documents, we identified a new dimension specifically related to feature toggles: *Activation Strategy*. Activation strategies are used to enable a feature for specific users or a subset of users. Each feature toggle management system defines rules as activation strategies.

Model Completion. The values we have identified *UserID* (6; 31), *GroupID* (31), *UserPercentage* (6; 30), and *IPs* (30; 31) are observed in the publications as well.

We observed the following activation strategies in our analysis:

1. *Default*: the feature is enabled for all users (106);
2. *AlwaysOn*: the feature is always enabled for all users (107);
3. *AlwaysOff*: the feature is always disabled for all users (107),
4. *UserID*: the feature is enabled for users with specified user ids. Users can select randomly or specifically (106);
5. *GroupID*: the feature is enabled for users in groups with specified group id (104);
6. *IPs*: the feature is enabled for a list of IPs. Using IP information, the feature could be enabled for users in a particular region or country as well (104);
7. *HostNames*: the feature is enabled for a list of hostnames (104);
8. *UserPercentage*: the feature is enabled to a percentage of users. In gradual rollout when the percentage increases, previous users should remain in (106);
9. *TimePercentage*: the feature is enabled for a percentage of time (115);
10. *TimePeriod*: the feature can be enabled on or after a date, on or before a date, between two dates, or on specified days of the week (107);

11. *Random*: the toggle randomly is enabled or disabled (107);
12. *Computed*: the feature is enabled based on the values of other feature toggles (111);
13. *Flexible*: which combines more than one strategy into one strategy to have a more complex strategy. An example is combining the UserID strategy, and Percentage strategy (104); and
14. *Custom*: in some cases, developers need a specific strategy that is not implemented as a built-in activation strategy. Some feature toggle packages allow developers to define a custom activation strategy (113).

Activation Strategy (S3). Activation strategies explain rules to activate a feature toggle. The values for this dimension are: Default (S3), AlwaysOn (S3), AlwaysOff (S3), UserID (S3), GroupID (S3), IPs (S3), HostNames (S3), UserPercentage (S3), TimePercentage (S3), TimePeriod (S3), Random (S3), Computed (S3), Flexible (S3), Custom (S3).

5.4 Application of the Extended Model of Software Configuration (MSCv2)

In this section, we show how MSCv2 can be used by researchers and practitioners (**answer to RQ3.2**). As explained in Subsection 5.2.5, to use MSCv2, researchers and practitioners should select appropriate values for each dimension to describe software configuration in their system. The result will be an instance of MSCv2. In Subsection 5.4.1, we present analysis of three publications that ask similar research questions about feature toggles and configuration options.In Subsection 5.4.2, we discuss two publications that perform research on Chrome. In Subsection 5.4.3, we show how the MSCv2 can be used to model existing configurations in the Chrome repository.

5.4.1 Similar Papers on Industry Practices

We selected three publications with similar research question of practices used by practitioners to use software configuration. Rahman et al. performed a thematic analysis of 17 videos and blog posts from practitioners to understand the challenges, and advantages of using *feature toggles* (6). Mahdavi-Hezaveh et al. analyzed 109 gray literature artifacts and publications to identify *feature toggle* usage practices (5). Sayagh et al. interviewed 14 practitioners, surveyed 229 practitioners, performed a systematic literature review of 106 publications, and identified activities, challenges, and recommendations related to

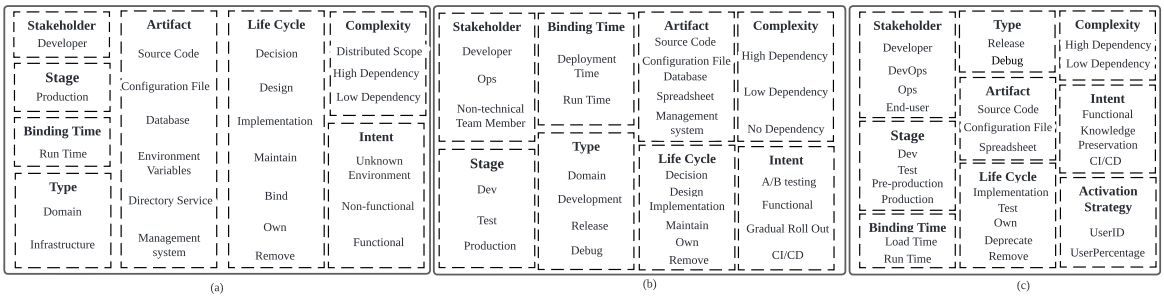


Figure 5.3: Application of the MSCv2 on three publications on practitioners' practices: (a) (4), (b) (5), (c) (6)

run-time configuration options (4).

To derive models for these three publications, we followed the steps discussed in Section 5.2.5, and the result is visualized in Figure 5.3. The publications may indicate that the research areas of these studies are different. However, looking at their MSCv2 instances in Figure 5.3 the studies have numerous similarities. Values for all dimensions have overlapped in these three instances. Considering the similarities and differences, the results in these studies can be compared or integrated. Mahdavi-Hezaveh et al.(5) discussed the partial overlap of seven out of 17 practices they identified with Sayagh et al.(4) study results. Mahdavi-Hezaveh et al. also used Rahman et al. (6) as one of the analyzed artifacts for their study.

New researchers in the software configuration research area can easily miss related publications if they are not aware of using different keywords of *feature toggles* and *configuration options* by other researchers. Also, if researchers populate an instance of MSCv2 in their publications other researchers can use their results more easily to develop new studies.

5.4.2 Studies on Chrome repository

We modeled the software configurations of Chrome from two publications (6; 7) as shown in Figure 5.4 using MSCv2. In (6), Rahman et al. quantitatively analyzed the usage of feature toggles in 39 releases of Chrome. In (7), the authors extracted four architectural representations of Chrome including feature toggle architecture. The same software repository is used in both publications. While modeling Chrome in these publications, our results indicate that even when the researchers analyze the same system, they may not define the

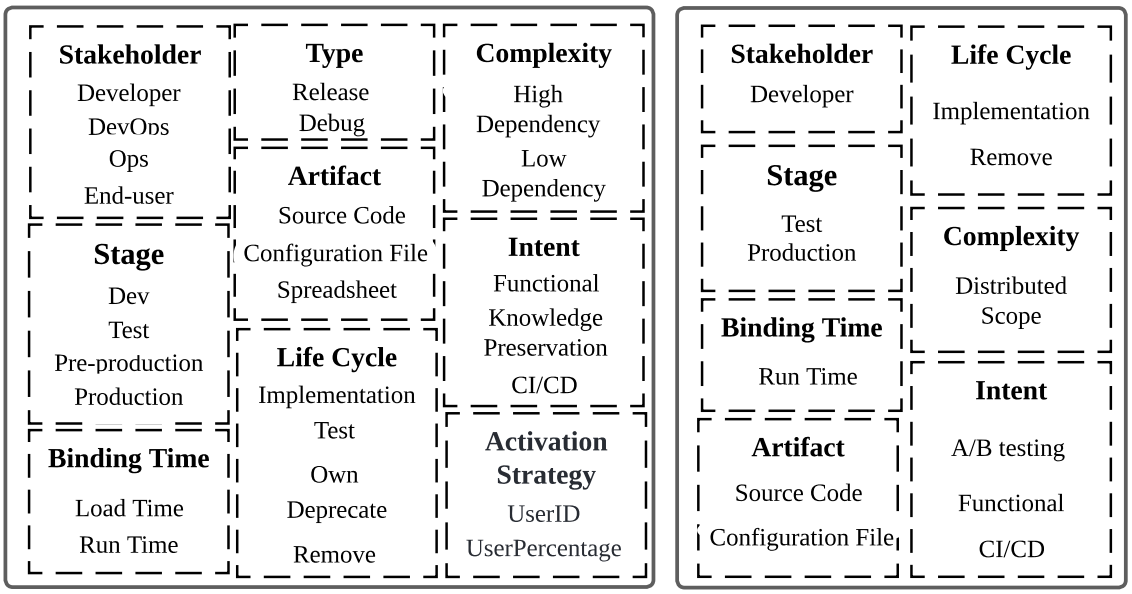


Figure 5.4: Application of the MCSv2 to two research studies on feature toggles on Chrome: (a) (6), (b) (7)

configuration the same due to the lack of a comprehensive model for defining software configuration. Some values for dimensions exist in both model instances but some dimensions and some values exist in just one of the model instances. If researchers looked at two models in Figure 5.4 without having knowledge about these two publications, they may claim that these two systems have similarities but they are not the same.

Researchers can use MSCv2 when designing their studies, and show the characteristics of software configuration in a MSCv2 instance. Looking at the instance, other researchers will have the same viewpoint of the system for their analysis.

5.4.3 Chrome Repository

To better understand the configurations in the Chrome repository, we applied the MSCv2 on the repository documentation (as we discussed in Section 5.2.5) to find values for each dimension for each configuration. The result is shown in Figure 5.5. In the Chrome documentation, five different terms are used for the software configuration: 1) Prefs, 2) Flags, 3) Settings, 4) Features, and 5) Switches. Based on the names of these configurations, the difference between them is unclear. For instance, in the research literature and gray literature switches and flags are the same concepts (6; 127), but in the Chrome repository, these

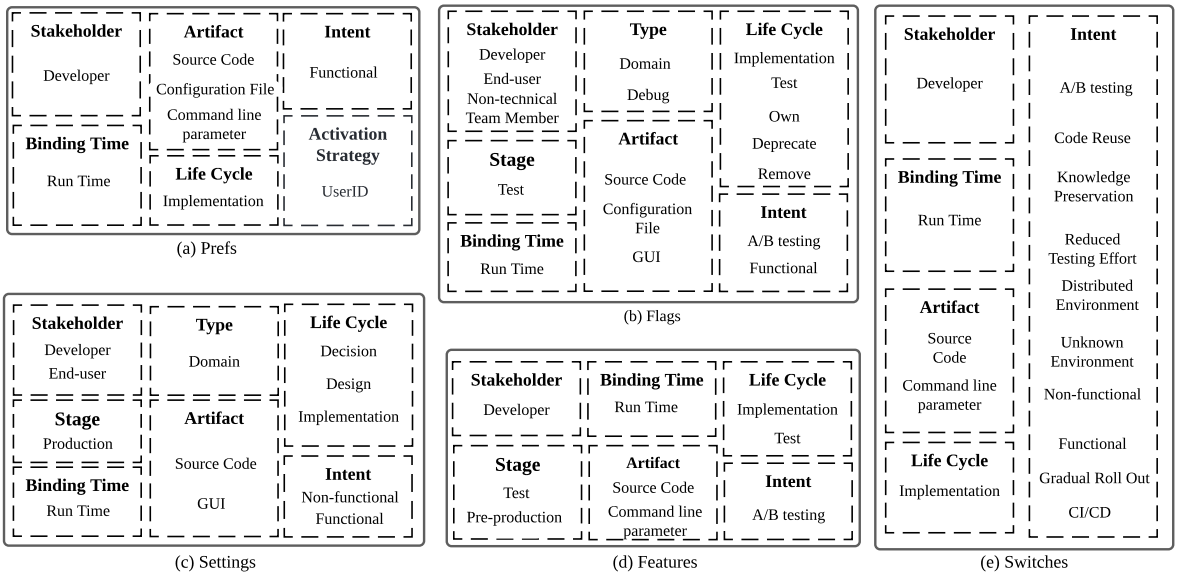


Figure 5.5: Application of the MSCv2 on Chrome

terms seem to be two separate concepts. Figure 5.5 indicates similarities and differences between these five MSCv2 instances. For example, developers are stakeholders of all five configurations, but non-technical team members only have access to Flags. We did not find values for some of the dimensions. For example, the documentation does not include information related to the *Complexity* dimension for any of the configuration terms. Architectures can use MSCv2 to define configuration in software projects to make sure that every characteristic of a configuration is clear to team members. If developers do not have a correct understanding of the concept behind each configuration, they may use them incorrectly.

We can compare the models in Figure 5.4 and Figure 5.5. In the two publications (6; 7) that are shown in Figure 5.4, Rahman et al. consider *Switches* in the Chrome repository as feature toggles in the system. Hence, both models in Figure 5.4 should be the same as Switches (e) in Figure 5.5. We observe that not only these three models are not the same but also they are not even subsets of each other. This observation shows that using unstructured documentation to define configuration in industry and in research publications may result in an incomplete and inconsistent understanding of a configuration definition.

5.4.4 Takeaways

The takeaways are based on our observations in Sections 5.4.1, 5.4.2, and 5.4.3.

For researchers

Researchers may define software configuration differently in their publications even if the industrial system is the same (Sections 5.4.2 and 5.4.3). They can use MSCv2 by creating an instance and selecting proper values for each dimension to model the software configuration in the system of their publications. Additionally, by using MSCv2 researchers can compare and integrate their results with similar studies, prevent duplication of effort, increase the impact of their research result through a bigger family of research, and meta-analyze the results of a family of research publications (Section 5.4.1). Also, using MSCv2 to model the configuration of industry systems by software architects when designing configurations, could help researchers to record the configuration more similar to what might be done by a practitioner, and enable knowledge transfer between industry and academia (Sections5.4.2).

For practitioners

The understanding of researchers from the same industrial system may not be the same because of a lack of clarity in the definition of configuration in the system's documentation and in publications (Sections 5.4.2 and 5.4.3). Hence, using MSCv2 to explain the configuration of industry projects by software architectures when designing configurations, could help researchers to record the configuration more similar to what might be done by a project practitioner, and enable knowledge transfer between industry and academia. In addition, an industrial system can have more than one kind of software configuration (Section 5.4.3), so a systematic definition of configurations by using MSCv2 can make the definitions more coherent, and more precise.

5.5 Threats to Validity

In Phase 1 of the methodology, we used keyword search to find related repositories. We may have missed related repositories if the keywords are not used in the name and description of the repository. In addition, in Step 4 of the filtering process of the search results, we read the name and description of the repositories. We did not check the content of the repositories.

However, since we selected the top 20 repositories in our result and checked the content of them, this filtering process did not affect our results. We selected 20 repositories from the top 5 programming languages and analyzed their documentation. The analysis of more repositories may enrich the results of this research study. To overcome this limitation, we defined the stopping criteria for analyzing documentation when we did not find any new dimension and any new value.

In Phase 2 of the methodology, we analyzed the feature toggle management systems' documentation using coding practices. To prevent subjective results, the coding was done by two individuals separately. Then we discussed and resolved disagreements. We may have missed some values because practitioners may use feature toggle in a different way that is not mentioned in the documentation.

In Phase 3 of the methodology, we applied MSCv2 to five publications and Chrome. To show the generalization of MSCv2, the publications were selected to be related to either feature toggles or configuration options, and the process was done by two individuals. In future work, we will ask other researchers to apply the MSCv2 in software configuration studies and share their observations.

5.6 Conclusion

Feature toggles and configuration options are used widely in the software development process for different purposes such as practicing CI/CD and system customization. Feature toggles and configuration options fall under the same umbrella with different names. Hence, we have extended the existing model of software configuration (MSC) to cover both feature toggle and configuration option concepts. The MSCv2 has 9 dimensions with 70 values in total. We observed that the MSCv2 may help researchers to build a comprehensive model to define configuration characteristics precisely in their research studies, and practitioners to define their system's configuration characteristics, and provide a common language for two communities to transfer knowledge about software configurations.

CHAPTER

6

META-ANALYSIS OF SOFTWARE CONFIGURATION RELATED PUBLICATIONS

6.1 Motivation

Researchers answer research questions on both feature toggles and configuration options. In the rest of the dissertation, we use the term *software configuration* to refer to the joint concepts of feature toggles and configuration options. Because of the similarity between these two concepts, similar problems exist that have similar solutions (11). For example, Mahdavi-Hezaveh et al. (128) showed that software configuration characteristics in research studies on practitioners' practices in the feature toggle research area (6; 5) and in the configuration option research area (4) have similarities that lead to similarities in their findings. Hence, making the similarities and differences between the software configuration characteristics more explicit in research studies may reduce duplication of effort, incomplete comparisons and generalizations, and inefficiency in research.

To provide terminology for the software configuration family of research, Siegmund et al. (12) proposed a Model of Software Configuration (MSC). MSC includes eight *dimensions* (Stakeholder, Stage, Binding Time, Type, Artifact, Life Cycle, Complexity, and Intent) that are higher-order topics related to configurations and a total 47 *values* assigned to these dimensions. Siegmund et al. proposed MSC by performing a set of interviews by practitioners and analysis of configuration option-related publications. Mahdavi-Hezaveh et al. (128) (Chapter 5) extended the MSC to MSCv2 by including feature toggle-related resources consisting of documentation of feature toggle management systems on GitHub and feature toggle-related publications. In MSCv2, Mahdavi-Hezaveh et al. added one new dimension (Activation Strategy) with 14 values and nine new values to existing dimensions in MSC. Using MSCv2, as a common terminology, aids in a comprehensive and more complete documentation of context variable in software configuration research studies. In this chapter (129), we provide a guideline on the usage of MSCv2 for researchers to document context variables in their research studies such that a meta-analysis of the software configuration family of research can be performed and replications of existing research studies can be done. In addition, MSCv2 can be used to surface the differences and similarities in different research publications, help researchers to find similar publications, transfer knowledge from one research area to another research area, and generate a mental model of all the research that has been done on software configurations.

Hence, *the goal of this research study is to aid researchers in performing a meta-analysis of the software configuration family of research through the application of the extended Model of Software Configuration (MSCv2) to related publications.*

To achieve this goal, we will examine the following three research questions:

RQ4: Can the application of the extended Model of Software Configuration (MSCv2) be used to perform a meta-analysis of a family of research on software configurations?

- **RQ4.1:** What are the similarities and differences in context variables in feature toggle-related publications and configuration option-related publications?
- **RQ4.2:** What research goals are common between the feature toggle research area and configuration options research area?
- **RQ4.3:** What research gaps exist in the feature toggle research area and configuration options research area?
- **RQ4.4:** What are the comprehensive definitions of feature toggles and configuration options based on the application of the Model of Software Configuration on related

publications?

RQ4.5: How can researchers use the extended Model of Software Configuration (MSCv2) in documenting context variables in their research studies?

6.2 Methodology

In this section, we explain our methodology. Figure 6.1 shows three phases of the methodology. In Phase 1, we systematically searched for research publications related to software configuration. In Phase 2, we applied MSCv2 to the selected publications. In Phase 3, we analyzed our observations, answered our research questions, and provided guidelines for future researchers. We explain the details of each phase in the following subsections.

6.2.1 Phase One: Search for Publications

The first step in our study is to find software configuration-related publications.

Digital Libraries Following Kuhrmann et al.'s guidelines (130), we select six digital libraries IEEE Digital Library (Xplore), ACM Digital Library, SpringerLink, ScienceDirect (Elsevier), Wiley Interscience, and IET (Add footnotes for all of them).

Search Strings To search these digital libraries, we constructed a set of search strings. To do so, we followed the Trail-and-Error search approach by (130). In this approach, researchers start with searching meta-search engines such as Google scholar with initial keywords, check the search result, change keywords or combine them, and iteratively find a list of search strings. We constructed search strings for feature toggles and configuration options separately as follows:

1. For feature toggles, we started by searching Google scholar with the "feature toggle" search string. The results of the search were a mix of feature toggle-related and unrelated publications on the first page. We added "software engineering" to "feature toggle". All of the results were related to the feature toggle on the first page. In the literature, feature toggles are also called feature flags, feature switches, feature bits, and feature flips (8). We replaced "feature toggle" in the initial search string with each of the four individual synonym phrases. Using "feature flag" and "feature switch" returns feature toggle-related publications in the mix with unrelated publications, but "feature bit" and "feature flip"

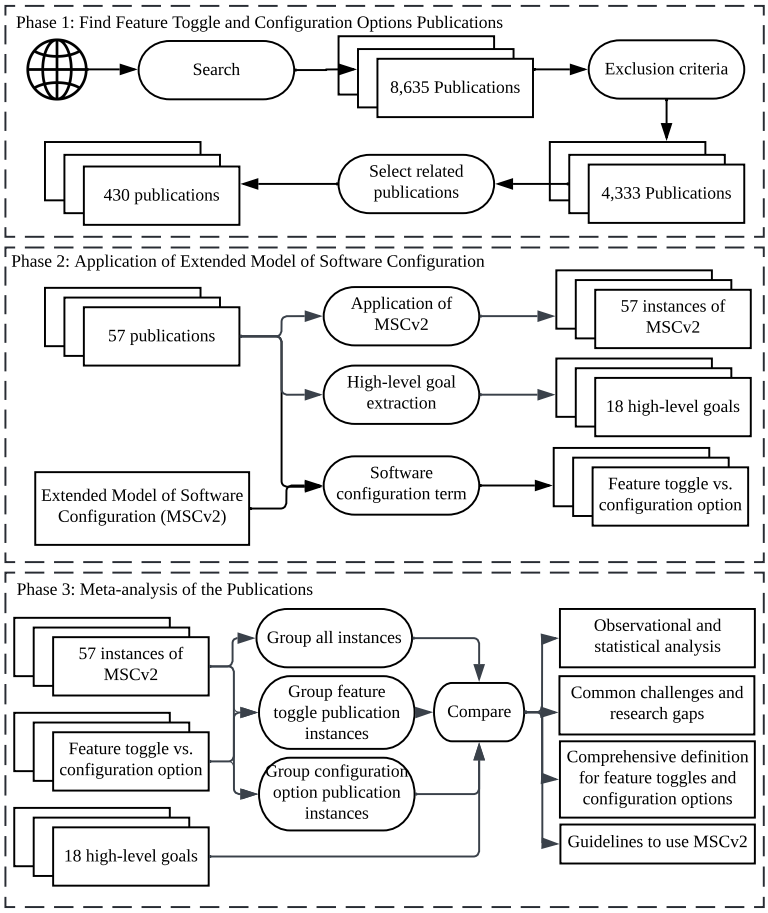


Figure 6.1: Research methodology outline.

did not return any related publications on the first page. So we used “feature toggle” AND “software engineering”; “feature flag” AND “software engineering”; and “feature switch” AND “software engineering” as search strings to find feature toggle-related publications.

2. For configuration options, we started by searching Google scholar with the “configuration option” search string. The results were a mix of configuration option-related and unrelated publications on the first page. However, after adding the “software engineering” to the search string, all the results were configuration option-related publications on the first page. After manually reviewing these search results, we realized that “software configuration” is also a used phrase as a synonym for “configuration option” in related publications. Hence, we tried “software configuration” AND “software engineer-

ing" and retrieved configuration option-related publications as a result. Also, during the manual review of the results, we found that "product line" is also frequently used in related publications. So, we tried "configuration options" AND "product line", and found search results related to configuration options. We also checked "software configuration" AND "product line", and after seeing configuration option-related results, we added that to our search strings.

As a result of mentioned steps, we obtain the following search strings:

- "feature toggle" AND "software engineering";
- "feature flag" AND "software engineering";
- "feature switch" AND "software engineering";
- "configuration option" AND "software engineering";
- "software configuration" AND "software engineering";
- "configuration option" AND "product line";
- "software configuration" AND "product line".

Search Digital Libraries On September 2 2022, we searched the six digital libraries with the seven search strings that we constructed. We retrieved 8,635 publications. Then, we apply the following exclusion criteria to the search result:

- We removed duplicate publications by checking the title, publication year, and authors of a publication;
- We removed publications that are not written in English;
- We removed publications that are not peer-reviewed, for example, books and chapters in books; and
- We removed magazine publications because they do not have enough details.

We had 4,333 publications after applying exclusion criteria.

Select Related Publications Two researchers reviewed the list of 4,333 publications using the ASReview LAB software [1]. ASReview uses active learning techniques to help screening a large amount of text (131) more efficiently. ASReview LAB accepts a set of publications as prior knowledge on related and unrelated publications, trains a classification model based on that knowledge, and suggests the next publication for labeling to the user. After each labeling, the model has trained again, and the process continues until the user decides to stop screening. The first researcher selected 439 relevant publications (after reviewing 849 publications), and the second researcher selected 404 relevant publications (after reviewing 815 publications). Then, the two researchers resolved their disagreements by discussing the content of disagreed publications. In the end, the two researchers selected a list of 430 publications as relevant publications.

6.2.2 Phase Two: Application of MSCv2 on Selected Publications

In this phase, we applied MSCv2 to feature toggle-related publications and configuration option-related publications. For feature toggle-related publications, the total number of publications is 10, so we will apply MSCv2 to all of them. For configuration option-related publications, we applied MSCv2 to the 47 publications published in 2021 and 2022. In total, we applied MSCv2 to 57 publications.

To apply MSCv2 to each publication, two individuals:

1. Read the abstract, introduction, background (definitions), and analyzed system explanation (case studies);
2. Extracted parts of the text explaining and defining the concept of software configuration in their research;
3. Performed deductive coding on the extracted text with the list of dimensions and values explained in Chapter 5, and created one instance of MSCv2. Deductive coding is an approach in which codes are developed ahead to guide the coding process, and inductive coding is an approach in which codes are developed while analyzing the text (124).; and
4. We recorded the high-level goal and the term used for the software configuration concept (feature toggle, configuration option, or their synonyms) of the publication.

After performing the mentioned steps for all publications in our list, two individuals resolved their disagreements on the values of dimensions.

[1] https://asreview.nl/

6.2.3 Phase Three: Meta-analysis of the publications

We analyzed the result of the application of MSCv2 to publications in three ways:

All publications. We aggregated MSCv2 instances of all publications in one aggregated instance. Using this aggregated instance, we discussed our observations on the frequency of specifying values of dimensions and what dimensions and values are missed in the software configuration research area in Section 6.3.1.

Feature toggle versus Configuration option. We created two instances of MSCv2 by integrating MSCv2 instances of publications based on the software configuration term used, feature toggle (and its synonyms) or configuration option (and its synonyms), for naming software configuration in the publications into two separate aggregated instances. Then, we analyzed the similarities and differences between these two aggregated instances in two ways: (1) Statistically and (2) qualitatively. For statistical analysis, we used Fisher's exact test (132) to compare the proportions of counts for each value in feature toggle aggregated and configuration option aggregated instances. For each value in the feature toggle aggregated instance and configuration option aggregated instance, p and q are the proportion of publications that have that value, respectively. So, the null hypothesis of Fisher's exact test is $p = q$ for each value in the MSCv2 model. MSCv2 has nine dimensions and 70 values in total, and we found 4 new values while performing Phase two of the methodology. Hence, because of having 74 values on the same data, we used Bonferroni correction (133) to control the false positive rate. We present the result of the statistical analysis in Section 6.3.2. In addition, we discuss our observations during the qualitative analysis of publications results in the two aggregated instances in Section 6.3.2. Considering these analyses of similarities and differences between the two aggregated instances, we proposed definitions for feature toggles and configuration options in Section 6.4.2.

High-level goals. During the application of MSCv2 to publications in Phase Two of the methodology, we recorded the high-level research goal of each publication. We discuss our analysis based on high-level goals in feature toggle-related publications and configuration option publications in Section 6.3.3.

Based on the observations and analysis of Phase Three, we propose a set of guidelines for researchers to use MSCv2 in documenting context variables in their research studies in Section 6.5.

6.3 Results

We applied MSCv2 as shown in Figure 5.2 in Chapter 5 to 57 selected publications. In Table 6.1, we list the 57 analyzed publications. Each publication has an identifier; a reference to the publication; the year of publication; the software configuration term used in the publication; and the high-level goal of the publication. In this table, we listed the feature toggle-related publications at the top and the configuration option-related publications at the bottom. As we discussed in the methodology, we applied MSCv2 to each one of the publications in this list by following the steps we explained in Section 6.2.2.

In Section 6.3.1, we aggregated the MSCv2 instances of all the publications and discuss our observations. In Section 6.3.2, we aggregated the MSCv2 instances of feature toggle-related publications and MSCv2 instances of configuration option-related publications separately and compare them to each other. In Section 6.3.3, we discuss high-level goals in the publications.

6.3.1 All publications

In Figure 6.2, we aggregated the MSCV2 instances of the analyzed 57 publications. The number in parenthesis after the dimension name is the number of publications that specify one or more values for the dimension. The numbers after each value are the number of publications that use that value in their publications. Because each publication can have more than one value for each dimension, the numbers after the dimension names are less than or equal to the sum of the numbers after the values of the dimension. Based on Figure 6.2, the most mentioned dimensions in all publications are Stakeholder (36 publications), Artifact (53 publications), Life Cycle (53 publications), Complexity (36 publications), and Intent (37 publications). Based on our observations, values of these four dimensions play key roles in problem statements and analysis in analyzed publications. Even though we observed the values for all dimensions in the publications, for Stakeholder, Type, Artifact, Life Cycle, Complexity, Intent, and Activation strategy dimensions, we did not find a total of 20 values in the publications. Two reasons may explain the lack of finding all values: 1) We did not analyze *all* the software configuration-related publications. So, we have missed the values in not analyzed publications, and 2) MSCv2 and original MSC were developed based on practitioners' experiences and researchers' experiences. Some of the missed values may be important in practice but not in research.

Table 6.1: Analyzed relevant publications

ID	Reference	Year	Software configuration term	High-level goal
P1	Rahman et al. (6)	2016	Feature toggle	Practices
P2	Rahman et al. (16)	2019	Feature toggle	Effect on architecture
p3	Meinicke et al. (14)	2020	Feature flag, feature toggle	Detection
P4	Ramanathan et al. (134)	2020	Feature flag	Removal
P5	Hoyos et al. (31)	2021	Feature toggle	Removal
P6	Mahdavi-Hezaveh et al. (5)	2021	Feature toggle	Practices
P7	Jezequel et al. (135)	2022	Feature toggle	Modeling
P8	Mahdavi-Hezaveh et al. (136)	2022	Feature toggle	Effect on code
P9	Prutchi et al. (32)	2022	Feature toggle	Effect on development process
P10	Sugnanam et al. (137)	2022	Feature toggle	Effect on code
P11	Ananieva et al. (138)	2021	Feature, feature option	Maintenance
P12	Bhagwan et al. (139)	2021	Configuration parameter	Maintenance
P13	Chen et al. (140)	2021	Configuration option	Selection
P14	Cordy et al. (141)	2021	feature, option, configuration option	Statistical model checking
P15	Dimovski et al. (142)	2021	Configuration option, feature, statically configurable option	Static analysis
P16	Dimovski et al. (143)	2021	Feature, configuration option	Static analysis
P17	Ding et al. (144)	2021	Configuration parameter	Selection
P18	Ferreira et al. (145)	2021	Configuration option, feature	Evaluating test strategies
P19	Franz et al. (146)	2021	Configuration option, feature	Selection
P20	Gottmann et al. (147)	2021	Configuration option	Static analysis
P21	Han et al. (148)	2021	Configuration Option	Performance prediction
P22	Krishna et al. (149)	2021	Option, configuration option	Selection
P23	Lesoil et al. (150)	2021	Option, configuration option	Performance prediction
P24	Linsbauer et al. (151)	2021	Feature, Configuration option	Maintenance
P25	Markiegi et al. (152)	2021	Feature	Test case prioritization
P26	Nguyen et al. (153)	2021	Configuration option	Effect on code
P27	Oh et al. (154)	2021	Configuration option	Static analysis
P28	Pereira et al. (155)	2021	Configuration option, option, feature, parameter	Selection
P29	Pinnecke et al. (156)	2021	Configuration, option	Practices
P30	Ramos et al. (157)	2021	Configuration option	Selection
P31	Saber et al. (158)	2021	Feature	Selection
P32	Safdar et al. (159)	2021	Configurable parameter, configuration option	Selection
P33	Sundermann et al. (160)	2021	feature	Feature model analysis
P34	Temple et al. (161)	2021	Configuration option, feature, attribute	Generate adversarial configurations
P35	Velez et al. (162)	2021	Configuration option	Performance prediction
P36	Weber et al. (163)	2021	Configuration option	Performance prediction
P37	Zhang et al. (15)	2021	Configuration parameter, parameter	Effect on Code
P38	Zhang et al. (164)	2021	Parameter, configuration parameter	Static analysis
P39	Ananieva et al. (165)	2022	Feature option	Modeling
P40	Ananieva et al. (166)	2022	Feature option	Maintenance
P41	Bertolotti et al. (167)	2022	Feature	Modeling
P42	Cheng et al. (168)	2022	Configuration option, feature	Performance prediction
P43	Damiani et al. (169)	2022	Configuration option, feature	Modeling
P44	Dimovski et al. (170)	2022	Configuration option, feature	Static analysis
p45	Dubslaff et al. (171)	2022	Configuration option, feature	Selection
P46	Heradio et al. (172)	2022	Feature	Selection
P47	Hou et al. (173)	2022	Feature	Selection
P48	Kroher et al. (174)	2022	Configuration option	Selection
P49	Mahmood et al. (175)	2022	Configuration option	Modeling
P50	Mercan et al. (176)	2022	Configuration option, Option setting	Selection
P51	Michelon et al. (177)	2022	Feature	Maintenance
P52	Nieke et al. (178)	2022	Configuration option, feature	Maintenance
P53	Prado et al. (179)	2022	Configuration option	Test case prioritization
P54	Randrianaina et al. (180)	2022	Compile-time configuration option	Incremental build
P55	Randrianaina et al. (181)	2022	Compile-time options, features	Incremental build
P56	Schubert et al. (17)	2022	Feature	Static analysis
P57	Ternava et al. (182)	2022	Compile-time/configuration options	Effect on system size and security

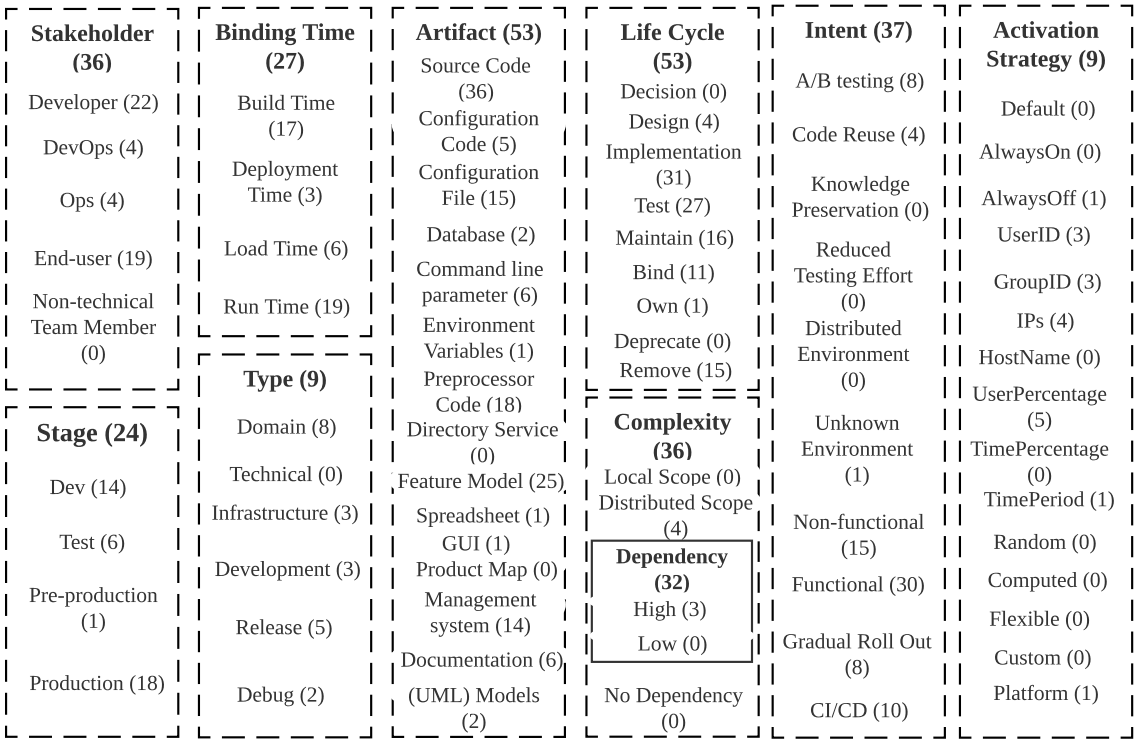


Figure 6.2: aggregated MSCv2 instances for all 57 publications

6.3.2 Feature toggle versus Configuration option

In Figure 6.3 and Figure 6.4, we aggregated all instances of MSCV2-1 for feature toggle-related publications and configuration option-related publications.

In Figure 6.3, the total number of publications is 10. We observed that values for Stakeholder, Stage, Artifact, Life Cycle, and Intent dimensions are mentioned in all of the publications. The least mentioned values are for Type and Complexity. We observed that values for all dimensions are mentioned in these publications. In total, we observed instances of 41 values and did not observe instances of 33 values in feature toggle-related publications.

In Figure 6.4, the total number of publications is 47. We observed that values for Stakeholder, Artifact, Life Cycle, Complexity, and Intent are mentioned more than others. In total, we observed instances of 40 values and did not observe instances of 34 values in feature toggle-related publications.

We compared these two models in two ways: 1) With Fisher's exact test (132) with Bonferroni correction (133) to find statistically-significant differences (Section 6.3.2); and 2)

Based on our observations during the application of MSCv2 on publications (Section 6.3.2).

Even though we discuss differences between software configuration definition in Figure 6.3 and Figure 6.4 in the following subsections, our results show that there is no difference in 65 values in MSCv2. The similarities and differences between feature toggle-related publications and configuration option publications confirm the blurred border between these two concepts.

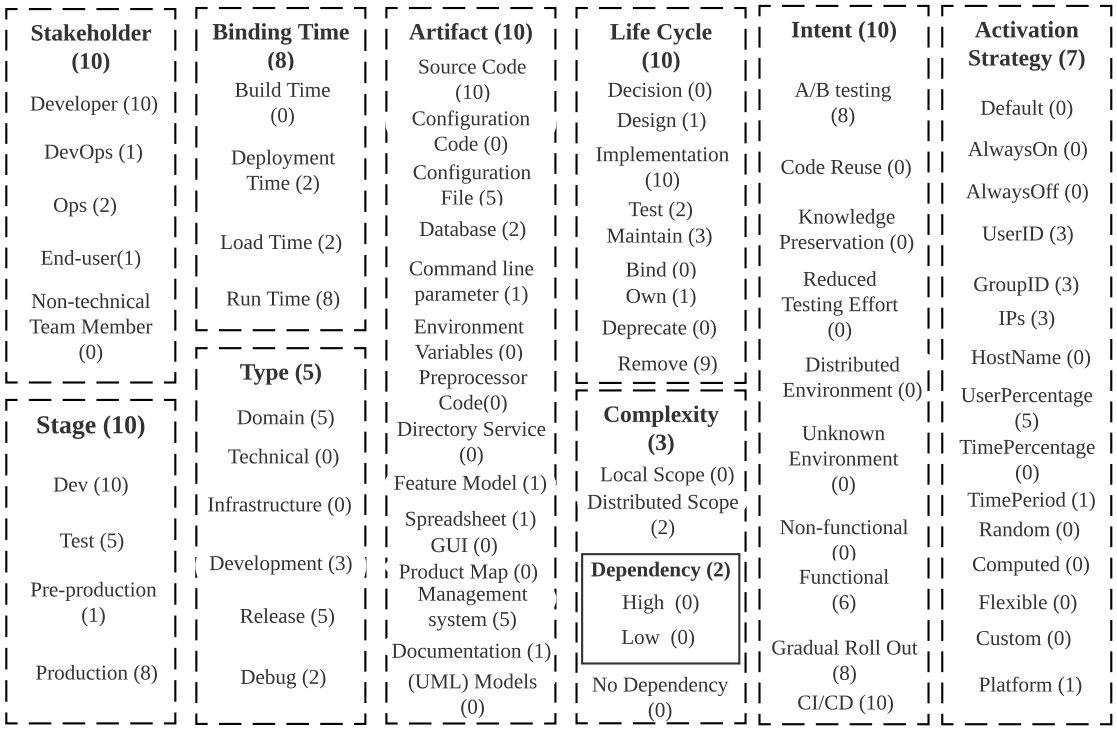


Figure 6.3: aggregated MSCv2 instances for 10 feature toggle-related publications

Statistical Analysis

To compare two models in Figures 6.3 and 6.4, we use Fisher's exact test. For each value in the aggregated instances, p and q are the proportion of feature toggle-related publications and configuration option-related publications that have the value, respectively. So, the null hypothesis of Fisher's exact test for each value is $p = q$. Because of having a large number

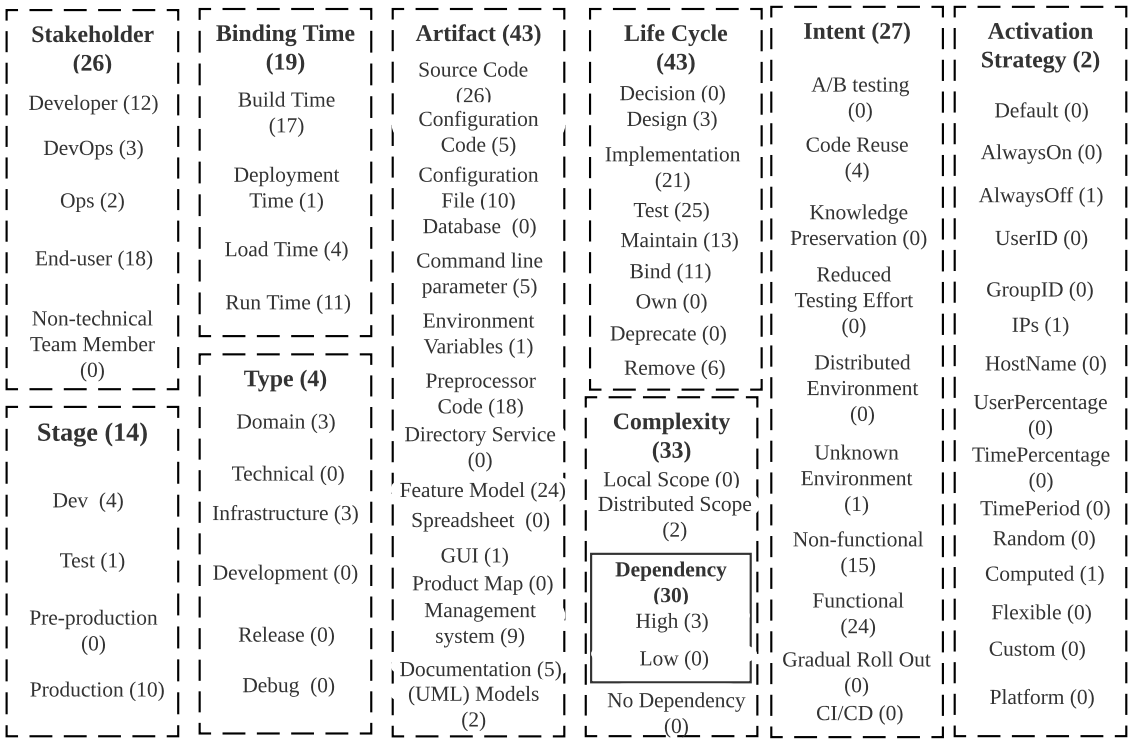


Figure 6.4: aggregated MSCv2 instances for 47 configuration option-related publications

(74) of values to perform the test on, we could not use 0.05 as the significance level. We used Bonferroni correction and control type I error. Hence, the significance level in our Fisher's exact test was $0.05/74 = 0.00068$. We list the p-values and results in Figure 6.5. The underlined values are statistically significant, which means the p-value is less than the significance level, and we reject the null hypothesis. All of the values with statistically significant p-value have higher proportions in feature toggle-related publications.

Based on the statistical result, differences in the following values are statistically significant between feature toggle-related and configuration option-related publications (sorted from smallest to largest p-values). Four out of nine statistically significant different values (Release, Gradual roll out, CI/CD, and Userpercentage) are the added values in MSCv2 (128) after analyzing feature toggle-related resources. For each value, we explain our observations as well:

CI/CD (Intent). Mahdavi-Hezaveh et al. (128) added CI/CD as a value to the Intent dimension in MSCv2 when including feature toggle-related sources. Therefore, it is reasonable to

have this value in all feature toggle-related publications and not in configuration option publications.

Dev (Stage). One of the goals of using feature toggles is CI/CD (including trunk-based development), and using feature toggles in trunk-based development is happening in the development stage of software development. In contrast, configuration options are mostly used in the production stage of software development to configure software systems based on user requirements.

A/B testing (Intent). A/B testing is mentioned as an intent of using feature toggles in 8 out of 10 feature toggle-related publications, and none of the configuration option-related publications. Similar to CI/CD, this value was also added as a value to the Intent dimension in MSCv2 by Mahdavi-Hezaveh et al. (128) after including feature toggle-related resources. So, A/B testing is one of the intents of using feature toggles and not configuration options.

Gradual Roll Out (Intent). Similar to CI/CD, gradual roll out is added to the MSCv2 by using feature toggle-related resources (128). This value is mentioned as intent in 7 out of 10 feature toggle-related publications and no configuration option-related publications.

Remove (Life Cycle). Researchers in feature toggle-related publications emphasized the importance of removing feature toggles after the goal of using them is achieved in 9 out of 10 publications. However, in configuration option-related publications, removing options is not important, and configuration options are mostly assumed to remain in the source code forever. This difference in removing feature toggles and keeping configuration options is mentioned in other research studies as well (11; 5; 128).

Developer (Stakeholder). Meinike et al.'s (11) stated that one of the key differences between feature toggles and configuration options is their stakeholder of them. They mentioned that feature toggles are generally controlled by developers and ops, and configuration options are generally by end-users. Our observation, as shown in Figures 6.3 and 6.4 confirms this key difference, and this difference for developer value is confirmed statistically.

Release (Type). Mahdavi-Hezaveh et al. (128) added Release as a value for the Type dimension after analyzing feature toggle-related resources. Developers use Release toggles to hide the incomplete implementation of a function from users. The Release type is not

mentioned in any of the configuration option-related publications. The Release type of the software configuration is specific to feature toggles.

UserPercentage (Activation Strategy). Activation Strategy is added in MSCv2 as a feature toggle specific dimension (128). Hence, we observed the values of this dimension more in feature toggle-related publications. UserPercentage is the activation strategy that is mentioned more than other values in feature toggle-related publications as a popular activation strategy.

Test (Stage). One of the usages of feature toggles that are mentioned in feature toggle-related publications is to test new features (6). In contrast, in configuration option-related publications, we did not find usage of configuration options in the Test stage. Hence, it is reasonable to have a statistically significant appearance of this value in feature toggle-related publications.

Observational Analysis

While we were applying the model to the publications, we observed some differences in the frequency of the values that appear in feature toggle-related publications and configuration option-related publications. Some of our observations were confirmed by the statistical analysis presented in Subsection 6.3.2. In this subsection, we explain the differences that we observed, but they are not confirmed by statistical analysis.

Binding Time. The main difference between values of Binding Time dimension in feature toggle-related publications and configuration option-related publications is Run Time versus Build Time (including compile time) as shown in Figures 6.3 and 6.4. In feature toggle-related publications, we did not observe compile time, and 8 out of 10 publications mentioned run time as the binding time of feature toggles. In configuration option-related publications, compile time is mentioned more than run time as the binding time. Since the compilation is part of the build process, we consider Build time as the value for these publications. Based on this observation, if a software configuration binds at compile time, it could fall into the definition of configuration option. However, if the binding time of a software configuration is run time, it may be considered as a feature toggle or configuration option.

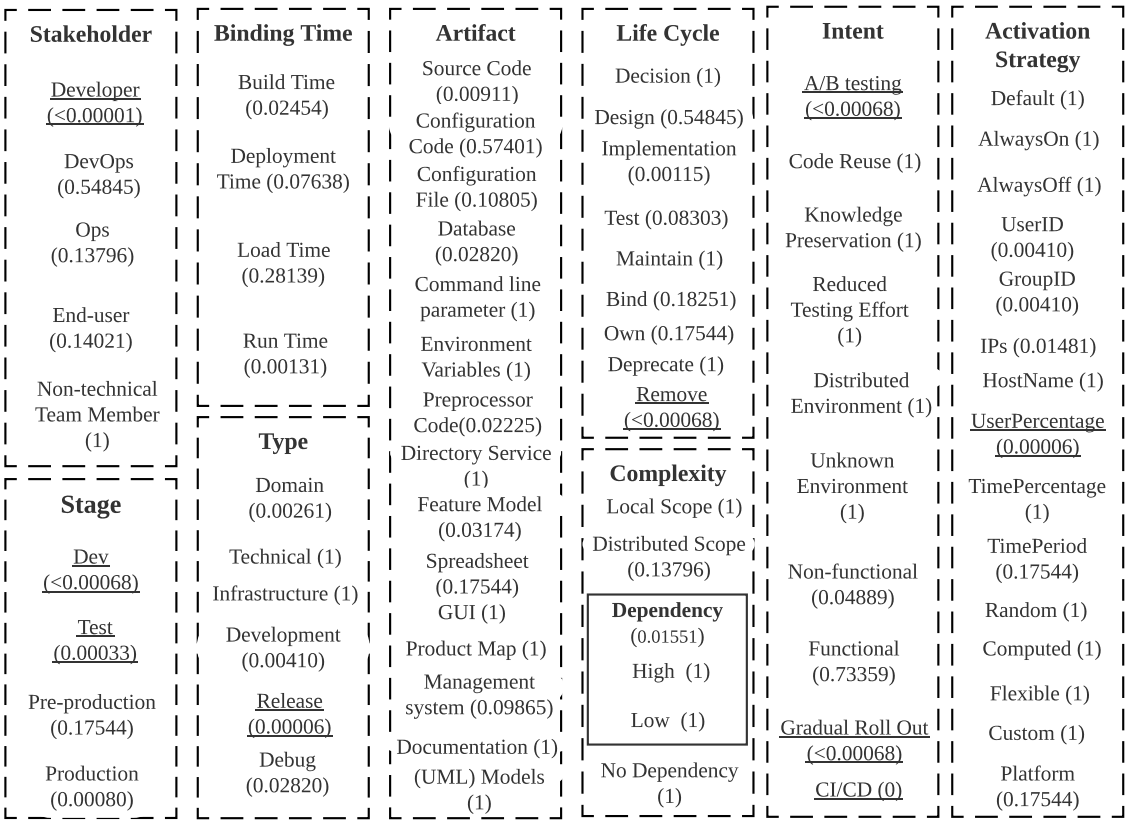


Figure 6.5: p-values of Fisher exact test with Bonferroni correction

Artifact. We observed that preprocessor code and feature models are frequently used in implementing and using configuration options. Using preprocessor code helps to implement compile-time software configuration which is not a binding time for feature toggles in related publications. Feature models are used to show the dependencies between existing configuration options in a system. In nine of the 10 feature toggle-related publications, modeling of the existing feature toggles in the software system with their possible dependencies was not discussed. Hence, researchers and practitioners can benefit from the practice of using feature models to list feature toggles and their dependencies in a model.

Complexity. In configuration option-related publications, the dependency between configuration options is discussed in the majority of the publications, as shown in Figure 6.4. Researchers may mention the existence of dependencies in their research studies (30), but dependency between feature toggles does not have a key role in research questions and anal-

ysis in feature toggle-related publications. Our observation supports Meinicke et al. (11)'s interview study that lists dependency as one of the differences between configuration options and feature toggles.

Life Cycle. Test as a value in the Life Cycle dimension appeared more in configuration option-related publications. Testing configuration options is a well-known research topic among researchers. However, even though researchers in feature toggle-related publications mentioned the importance of testing feature toggles (6), none of the 10 feature toggle-related publications focused on this challenge.

6.3.3 High-level goals

We show the high-level goal of each publication in Table 6.1. In this section, we discuss our observations regarding high-level goals in the software configuration family of research. Using the results in this subsection, we will explain the research gaps and future research opportunities in Section 6.4.1.

Similarities

Three of the high-level goals, namely *Practices*, *Effect on code*, and *Modeling* exist in both feature toggle-related publications and configuration option-related publications. Researchers in *Practices* publications, P1, P6, and P29 (two feature toggle publications and one configuration option publication), discuss practitioners' practices for using feature toggles and configuration options. In *Effect on code* publications, P8, P10, P26, and P37 (two feature toggle publications and two configuration option publications), researchers analyzed systems at the code level and discussed the effects of using software configurations on code. Researchers discuss modeling of software configurations in publications with *Modeling* as the high-level goal, including P7, P39, P41, P43, and P49 (one feature toggle publication and three configuration option publications). Because of the similarity between the feature toggles and configuration options, the results in publications with the same high-level goals in the feature toggle research area and the configuration option research can have overlaps. Raising awareness about these similarities can prevent duplication of effort in future research studies.

Differences

Other than similar high-level goals, the rest of the goals are different between feature toggle and configuration option research areas.

For feature toggles, researchers in P2 and P9 discussed how the use of feature toggles affects the architecture of the software system and affects the development process, *Effect on architecture* and *Effect on development process* respectively. In *Removal* publications, including P4 and P5, researchers proposed approaches to remove feature toggles from the code. In Section 6.3.2, we statistically showed that removing is one of the differences between the lifecycle of feature toggles and configuration options. In addition, having a publication with *Detection* in the feature toggle research area, P3, shows the difficulty of finding feature toggles because of a lack of proper documentation and modeling.

For configuration options, researchers in the publications with high-level goals of *Static analysis* including P15, P16, P20, P27, P38, P44, P56, *Selection* including P13, P17, P19, P22, P28, P30, P31, P32, P45, P46, P47, P48, and P50, *Performance prediction* including P21, P23, P35, P36, and P42, *Test case prioritization* including P25, and P53, *Generate adversarial configurations* which is P34, *Evaluating test strategies* which is P18, and *Statistical model checking* which is P14 focused on *testing* configuration options. Because of having a large number of configuration options in the software systems and the dependency between them, testing configuration options is not easy, and it is well studied in the configuration option research area. Other than these high-level goals, *Feature model analysis* in P33, *Maintenance* in P11, P12, P24, P40, P51, and P52, *Incremental build* in P54 and P55, and *Effect on system size and security* in P57 are the high-level goals that appear in configuration option research area and not feature toggle research area.

6.4 Discussion

6.4.1 Research Gaps

In this subsection, we describe the research gaps in the software configuration family of research based on our analysis in Sections 6.3.2, and 6.3.3.

Testing feature toggles. In the analysis in Sections 6.3.2 and 6.3.2 and the discussion on high-level goals in Section 6.3.3, we observed that testing configuration options are one of the high-level goals in related publications. In feature toggle-related publications, the

importance of testing feature toggles are explained, but none of the related publications addressed this challenge. Hence, we suggest researchers design research studies on testing feature toggles.

Dependency between feature toggles. Meinicke et al. (11) mentioned dependency as one of the differences between configuration options and feature toggles. In their interview study, Meinicke et al. found that configuration options have many dependencies on each other, but feature toggles have few dependencies. Our observations in Section 6.3.2 are aligned with their findings. However, none of the feature toggle-related publications is focused on finding the dependencies between feature toggles in large-scale datasets. We suggest researchers perform research studies on doing empirical analysis on feature toggle dependencies in code repositories.

6.4.2 Definitions of Feature Toggle and Configuration Option

The application of MSCv2 to 57 publications in this study shows that feature toggle and configuration option concepts have characteristics in common as was mentioned in other research studies (11; 12; 128). At the same time, Figure 6.5 and discussions in Sections 6.3.2 and 6.3.2 show the differences and the blurred border between feature toggle and configuration option concepts. Also, in Section 1 we showed the inconsistency in defining these two concepts in the research community. Hence, based on the application of MSCv2 to 57 publications and our analysis, we propose definitions for each concept in this subsection. Future researchers can use these definitions in their research studies.

To provide the definitions, we use the aggregated MSCv2 model for feature toggle-related publications (Figure 6.3), the aggregated MSCv2 model for configuration option-related publications (Figure 6.4, statistical analysis on significant values in the two models (Figure 6.5 and Section 6.3.2), and our observations while applying the MSCv2 to publications (Section 6.3.2. These definitions can aid in distinguishing feature toggles from configuration options. The provided definitions are as follows:

Feature toggles. Using Feature toggles is a technique to guard blocks of code with a variable in a conditional statement, and by changing the value of the variable, enable or disable that part of the code. Feature toggles primarily bind at run-time, and developers use them to implement A/B testing, gradual rollout, and CI/CD. Feature toggles should be removed from the source code when their purpose is complete.

Configuration options. Configuration options are key-value pairs that allow software customization by including or excluding functionality in a software system. Configuration options primarily bind at compile-time, are modeled in feature models, and depend on each other.

6.4.3 Relations between dimensions and values

While we were performing Phase Two of our methodology and applied MSCv2 to 57 publications, we observed the following relations between dimensions and values:

Dimensions: Stage, Type, and Intent Values in the three dimensions of Stage, Type, and Intent have a direct mapping to each other. For example, when a software configuration is used to change the functionality of a product based on the user's need, the configuration happens in the *production* Stage, the Type of the software configuration is *Domain*, and the Intent of using the software configuration is *functional*. Hence, if the value of the Intent dimension is mentioned, values of Stage and Type can be derived. Future researchers can examine the relationship between these three dimensions in the future to gain more understanding of the software system.

Values: Dependency and Feature Model The feature model is the most used artifact in configuration option-related publications. Researchers emphasize feature models are used to show the dependency between configuration options in software systems. Hence, there is a direct relation between feature model value (in the Artifact dimension) and dependency (in the Complexity dimension) in the MSCv2.

Values: Preprocessor Code and Compile-time *Preprocessor Code* is an approach to implementing compile-time software configurations in software systems. In MSCv2, we record compile time in the value *Build Time* in the Binding Time dimension, because compilation is part of the building process. In the publications with a preprocessor as an Artifact, build time should be selected as binding time.

6.5 Guidelines to Use MSCv2

We suggest researchers who work on the software configuration family of research use MSCv2 to document context variables in their research studies. To use MSCv2, the researchers should answer the following questions considering the definition of software configuration in their proposed research study, their problem statement, and the case studies they will use in their research study for evaluation:

1. Who is making configuration decisions? (Stakeholder)
2. In what stages of the development process configuration happens? (Stage)
3. When a value is binding to a software configuration? (Binding Time)
4. What is the type of software configuration in the system? (Type)
5. What artifacts contain software configurations? (Artifacts)
6. What are the lifetime phases of software configurations? (Life Cycle)
7. What is the scope and dependency level between software configurations? (Complexity)
8. What is the purpose of using software configurations? (Intent)
9. What are the rules to enable software configuration for a subset of users? (Activation Strategy)

Each question reflects one of the dimensions in MSCv2, and researchers should select one or more values of the dimension from Figure 5.2 in Chapter 5. After selecting values, researchers can create a visual instance of MSCv2 and include it in their research study.

Other than documenting context variables in research studies on software configuration, when a research study includes an instance of MSCv2, other researchers: 1) can understand the characteristics of software configurations in the study; 2) compare the study having MSCv2 instance with their own studies based on similarities and differences between MSCv2 instances; 3) use the MSCv2 instance to design a replication study.

6.6 Threats to validity

In Phase One of the methodology, we used search strings to find software configuration-related publications. We may have missed some related publications because of using the limited search strings. To minimize this threat, we followed Kuhrmann et al.'s guidelines (130) to construct our search stings.

In Phase Two of the methodology, because of having a large number of configuration option-related publications, we applied MSCv2 to publications that are published in 2021 and 2022. So, we have missed some insights about the configuration option search area.

In Phase Three of the methodology, statistical analysis and qualitative analysis of the differences and similarities between feature toggle aggregated MSCv2 instance, and configuration option aggregated MSCv2 instance is based on the application of MSCv2 on 57 publications. We may have missed some differences and similarities because of the limited number of analyzed publications.

In the discussion section, we proposed comprehensive definitions for feature toggles and configuration options based on our results in this study. To perform this research study, we analyzed academic publications, so the proposed definitions are from the researchers' perspective. Practitioners may have a different perspective on how to define these two concepts in the industry. In addition, we listed our observed relations between dimensions or between values in different dimensions. We may have missed some relations because of analyzing 57 publications in the research community. To address this threat, the observed relations are confirmed by two individuals.

6.7 Conclusion

Feature toggles and configurations options are *software configurations* used in software development, and they have similarities and differences in their definitions in research studies. Researchers can use common terminology to document context variables in their research studies to enable the meta-analysis of the software configuration family of research and prevent duplication of effort. We proposed guidelines for researchers in using MSCv2 to document the context variable in their research studies. We also proposed definitions for feature toggles and configuration options based on systematic documentation of context variables in 57 software configuration-related publications. Researchers can use the provided definitions in their future research studies.

CHAPTER

7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

In Chapter 3, we performed a qualitative analysis of 99 artifacts from grey literature and 10 peer-reviewed papers. We identified 17 feature toggle practices in four categories: Management practices (6), Initialization practices (3), Implementation practices (3), and Clean-up practices (5). We also quantified the frequency of usage of these identified practices in the industry by analyzing company-specific artifacts and conducting a survey. The feature toggle development practices discovered and enumerated in this work could raise practitioners' awareness of feature toggle practices and their usage in the industry. Using the result of this chapter could help practitioners to better use feature toggles in their projects, which was the goal of doing this work.

In Chapter 4, we conducted a qualitative analysis of 80 repositories that use feature toggles in their development cycle. We proposed 7 heuristics to guide structuring feature toggles and identified 12 metrics to support the principles embodied in the heuristics. Our survey result shows that practitioners agree that managing feature toggles is difficult, and using identified heuristics can reduce technical debt. Since the identified FT-heuristics are

similar to general software engineering best practices in software development literature, feature toggles should be considered as regular code even though they will not reside permanently in the code. However, even these general software engineering best practices are not followed by all of the analyzed repositories. Not structuring feature toggles correctly can impact the quality of the code base in the project. Thus, increasing the awareness of the effect of following and not following FT-heuristics is important.

In Chapter 5, we extended an existing Model of Software Configuration through the qualitative analysis of feature toggle-related software repositories and publications, and proposed MSCv2 with 9 dimensions and 70 values. The software configuration (feature toggles and configuration options) family of research can benefit from MSCv2 as a common ontology to enable meta-analysis. Researchers can use MSCv2 to document the context variables in their research studies. As a result, by considering similarities and differences between the characteristics of software configuration in research studies, researchers can prevent duplication of effort, inefficiency in research, and unclear comparisons and generalizations of research results.

In Chapter 6, we applied MSCv2 to 57 software configuration research publications. We analyzed similarities and differences between feature toggle-related publications and configuration option-related publications statistically and qualitatively. We found common research challenges and gaps between two research areas of feature toggles and configuration options, proposed comprehensive definitions for feature toggles and configuration options. At the end, we proposed MSCv3 and guidelines on its usage for researchers. Researchers can use MSCv3 based on proposed guidelines to document context variables in their research studies and compare our research studies with similar research studies easier.

7.2 Future Work

In Chapter 3, the identified feature toggle practices discovered through the study can enable a future quantitative analysis to automatically identify practice use in code repositories. Also, a future study can be conducted on the empirical analysis of the effectiveness of the identified practices in repositories which practices are used. Additional future work involves the automatic identification of feature toggle bad smells in the code, such as unused feature toggles, nested feature toggles, and development of a tool to automatically refactor the code when bad smells are identified. Also, the quality of the parts of the code which is activated

or deactivated by feature toggles is one of the concerns mentioned by practitioners (60). Studying the impact of using feature toggles on code quality, such as high cohesion and low coupling, could also be future work.

In Chapter 4, the identified FT-metrics can be validated by 47 criteria to validate software metrics extracted by Meneely et al. (183) in future work. Additionally, we manually computed the metrics. An automated tool for computing FT-metrics and applying FT-heuristics for structuring feature toggles in the code base is a future direction. Such a tool could be used to conduct a larger scale evaluation effort to generalize our findings outside of open source repositories. In this study, we analyze the last snapshot of repositories. A direction for future work is to consider metrics related to "lifetime" of feature toggles from the history of repositories, such as how long the toggle lived in the code base, or who is the last developer who touched the toggle. Considering additional metrics can result in additional heuristics about structuring feature toggles in code bases. Our case study result shows that Testing (H6) is not followed by a large number of repositories (54). We focus on raising awareness about having test cases for feature toggles. Considering the importance of testing strategies and especially combinatorial testing related to configuration options (184), research on combinatorial testing for feature toggles can be a future direction. We conduct preliminary statistical analysis (Appendix B) on the case study dataset and report our findings. A future direction is to conduct a larger-scale empirical study with rigorous statistical analysis to strength these findings on the relationship between following FT-heuristics and improving FT-metrics and finding new relationships.

In Chapter 6, a future path is to fill research gaps that we found in our analysis. Testing configuration options is one of the main challenges addressed by researchers. However, testing feature toggles is not researcher's focus in any of the analysed feature toggle related publications. Future researchers can develop testing strategies for feature toggles by using research results on testing configuration options in publications. In addition, researchers can perform research studies regarding dependencies between feature toggles. Dependencies between configuration options as modeled in feature models is the basis of some research studies, however none of the feature toggle related publications investigate dependencies between feature toggles. Researchers can start with configuration option related publications and transfer knowledge to feature toggle research area for investigating dependencies between feature toggles.

We hope our dissertation will facilitate further software configuration related research in academia and practices in industry.

REFERENCES

[1] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, vol. 106, pp. 101–121, 2019.

[2] A. Kenner, C. Kästner, S. Haase, and T. Leich, "Typechef: Toward type checking# ifdef variability in c," in *Proceedings of the 2nd international workshop on feature-oriented software development*, pp. 25–32, 2010.

[3] A. Tiwari, "Decoupling deployment and release- feature toggles." [Online]. Available: `https://www.abhishek-tiwari.com/decoupling-deployment-and-release-feature-toggles/` Accessed 24 April 2019, 2013.

[4] 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*, 2018.

[5] R. Mahdavi-Hezaveh, J. Dremann, and L. Williams, "Software development with feature toggles: practices used by practitioners," *Empirical Software Engineering*, vol. 26, no. 1, pp. 1–33, 2021.

[6] M. T. Rahman, L.-P. Querel, P. C. Rigby, and B. Adams, "Feature toggles: practitioner practices and a case study," in *Proc. of the 13th International Conference on Mining Software Repositories*, pp. 201–211, ACM, 2016.

[7] M. T. Rahman, P. C. Rigby, and E. Shihab, "The modular and feature toggle architectures of Google Chrome," *Empirical Software Engineering (EMSE)*, vol. 22, no. 2, pp. 1–28, 2018.

[8] M. Fowler, "Feature toggle." [Online]. Available: `https://martinfowler.com/bliki/FeatureToggle.html` Accessed 24 April 2019, 2010.

[9] 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.

[10] J. Bird, "Feature toggles are one of the worst kinds of technical debt." [Online]. Available: `https://dzone.com/articles/feature-toggles-are-one-worst` Accessed 24 April 2019, 2014.

[11] J. Meinicke, C.-P. Wong, B. Vasilescu, and C. Kaestner, "Exploring differences and commonalities between feature flags and configuration options," in *To be appear in Proc. of the 42nd International Conference on Software Engineering - Software Engineering in Practice (ICSE-SEIP)*, 2020.

[12] N. Siegmund, N. Ruckel, and J. Siegmund, "Dimensions of software configuration: on the configuration context in modern software development," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 338–349, 2020.

[13] S. Apel, D. S. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer, 2013.

[14] J. Meinicke, J. Hoyos, B. Vasilescu, and C. Kästner, "Capture the feature flag: Detecting feature flags in open-source," in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, (Seoul), pp. 169—-173, 2020.

[15] Y. Zhang, H. He, O. Legunsen, S. Li, W. Dong, and T. Xu, "An evolutionary study of configuration design and implementation in cloud systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 188–200, 2021.

[16] M. T. Rahman, P. C. Rigby, and E. Shihab, "The modular and feature toggle architectures of google chrome," *Empirical Software Engineering*, vol. 24, no. 2, pp. 826–853, 2019.

[17] P. D. Schubert, P. Gazzillo, Z. Patterson, J. Braha, F. Schiebel, B. Hermann, S. Wei, and E. Bodden, "Static data-flow analysis for software product lines in c," *Automated Software Engineering*, vol. 29, no. 1, pp. 1–37, 2022.

[18] G. A. Moore, *Crossing the chasm: Marketing and selling technology project.* Harper Collins, 2009.

[19] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader).* Pearson Education, 2010.

[20] M. Fowler, "Continuousdelivery." [Online]. Available: `https://martinfowler.com/bliki/ContinuousDelivery.html` Accessed 24 April 2019, 2013.

[21] R. Harmes, "Flipping out." [Online]. Available: `http://code.flickr.net/2009/12/02/flipping-out/` Accessed 24 April 2019, 2009.

[22] P. Hodgson, "Feature toggles (aka feature flags)." [Online]. Available: `https://martinfowler.com/articles/feature-toggles.html` Accessed 24 April 2019, 2017.

[23] M. Jacobs and E. Kaim, "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 April 2022, 2021.

[24] 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.

[25] C. Lefebvre, E. Manheimer, and J. Glanville, "Searching for studies," *Cochrane handbook for systematic reviews of interventions: Cochrane book series*, pp. 95–150, 2008.

[26] V. Garousi, M. Felderer, M. V. Mäntylä, and A. Rainer, "Benefitting from the grey literature in software engineering research," *arXiv preprint arXiv:1911.12038*, 2019.

[27] V. Garousi, M. Felderer, and M. V. Mäntylä, "The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature," in *Proceedings of the 20th international conference on evaluation and assessment in software engineering*, pp. 1–6, 2016.

[28] 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.

[29] 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*, pp. 1–10, IEEE, 2015.

[30] 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), pp. 1–10, ACM, jul 2020.

[31] J. Hoyos, R. Abdalkareem, S. Mujahid, E. Shihab, and A. E. Bedoya, "On the removal of feature toggles," *Empirical Software Engineering*, vol. 26, no. 2, pp. 1–26, 2021.

[32] E. S. Prutchi, H. de S. Campos Junior, and L. G. Murta, "How the adoption of feature toggles correlates with branch merges and defects in open-source projects?," *Software: Practice and Experience*, vol. 52, no. 2, pp. 506–536, 2022.

[33] 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)*, pp. 483–494, 2016.

[34] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 105–114, 2010.

[35] D. Jin, X. Qu, M. B. Cohen, and B. Robinson, "Configurations everywhere: Implications for testing and debugging in practice," in *Companion Proceedings of the 36th International Conference on Software Engineering*, pp. 215–224, 2014.

[36] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim, "Splat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 257–267, 2013.

[37] L. A. Williams, W. Krebs, L. Layman, and A. I. Anton, "Toward a framework for evaluating extreme programming," tech. rep., North Carolina State University. Dept. of Computer Science, 2004.

[38] L. A. Williams, L. Layman, and W. Krebs, "Extreme programming evaluation framework for object-oriented languages version 1.4," tech. rep., North Carolina State University. Dept. of Computer Science, 2004.

[39] P. J. Morrison, *A security practices evaluation framework*. North Carolina State University, 2017.

[40] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

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

[42] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, p. 38, Citeseer, 2014.

[43] J. Saldaña, *The coding manual for qualitative researchers*. Sage, 2015.

[44] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, "Development and deployment at facebook," *IEEE Internet Computing*, vol. 17, no. 4, pp. 8–17, 2013.

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

[46] S. Neely and S. Stolt, "Continuous delivery? easy! just change everything (well, maybe it is not that easy)," in *2013 Agile Conference*, pp. 121–128, IEEE, 2013.

[47] G. Schermann, J. Cito, P. Leitner, U. Zdun, and H. C. Gall, "We're doing it live: A multi-method empirical study on continuous experimentation," *Information and Software Technology*, vol. 99, pp. 41–57, 2018.

[48] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic configuration management at facebook," in *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 328–343, 2015.

[49] K. Schneid, "Branching strategies for developing new features within the context of continuous delivery.," in *CSE@ SE*, pp. 28–35, 2017.

[50] G. Schermann, J. Cito, and P. Leitner, "Continuous experimentation: challenges, implementation techniques, and current research," *Ieee Software*, vol. 35, no. 2, pp. 26–31, 2018.

[51] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner, and M. Oivo, "Continuous deployment of software intensive products and services: A systematic mapping study," *Journal of Systems and Software*, vol. 123, pp. 263–291, 2017.

[52] E. Rusovsky, "Feature flag management systems save you from technical debt." [Online]. Available: https://rollout.io/blog/feature-flag-management-technical-debt/ Accessed 18 June 2019, 2017.

[53] Chromium, "Chromium flags." [Online]. Available: https://docs.google.com/spreadsheets/d/1XwGEpBidtPKS_xFzW8bfQ4kbuVe0OuPlpMsSd8wTmmo/edit Accessed 24 April 2019, 2019.

[54] R. Stephens, *Beginning software engineering*. John Wiley & Sons, 2015.

[55] LaunchDarkly, "Launchdarkly best practices." [Online]. Available: https://github.com/launchdarkly/featureflags/blob/master/5%20-%20Best%20Practices.md Accessed 24 April 2019, 2018.

[56] Split, "Audit logs." [Online]. Available: https://docs.split.io/docs/streaming-audit-logs Accessed 24 April 2019, 2019.

[57] I. C. Østhus, "Continuous deployment: Unleash your features gradually!." [Online]. Available: http://bytes.schibsted.com/unleash-features-gradually/ Accessed 24 April 2019, 2017.

[58] I. Engineering, "Flexible feature control at instagram." [Online]. Available: https://instagram-engineering.com/flexible-feature-control-at-instagram-a7d3417658df Accessed 24 April 2019, 2015.

[59] G. J. Kieser, "Removing risk from product launches: a webinar with launchdarkly, circleci and gopro." [Online]. Available: https://circleci.com/blog/removing-risk-from-product-launches-a-webinar-with-launchdarkly-circleci-and-gopro/ Accessed 27 January 2020, 2017.

[60] R. L. Erik Sowa, "Feature bits: Enabling flow within and across teams." [Online]. Available: https://www.infoq.com/presentations/Feature-Bits Accessed 24 April 2019, 2010.

[61] T. UL, "There is no devops without feature flags!." [Online]. Available: https://www.ibm.com/developerworks/community/blogs/c914709e-8097-4537-92ef-8982fc416138/entry/THERE_IS_NO_DEVOPS_WITHOUT_FEATURE_FLAGS?lang=en Accessed 24 April 2019, 2016.

[62] B. Nadel, "Using the launchdarkly dashboard and json types to create light-weight application administrative features." [Online]. Available: https://www.bennadel.com/blog/3465-using-the-launchdarkly-dashboard-and-json-types-to-create-light-weight-application-administrative-features.htm Accessed 27 January 2020, 2018.

[63] B. Nadel, "Launchdarkly lunch-and-learn panel discussion: New york city." [Online]. Available: https://www.bennadel.com/blog/3464-launchdarkly-lunch-and-learn-panel-discussion-new-york-city.htm Accessed 18 June 2019, 2018.

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

[65] L. Darkly, "Feature flags, toggles, controls - cleaning up." [Online]. Available: http://featureflags.io/feature-flags-cleaning-up/ Accessed 13 January 2020, 2020.

[66] P. Hodgson, "Lean product development: Managing feature flags at scale." [Online]. Available: https://www.youtube.com/watch?v=uFW4SSRtkUU Accessed 24 April 2019, 2018.

[67] M. Meyer, "Using feature flags to ship changes with confidence." [Online]. Available: https://blog.travis-ci.com/2014-03-04-use-feature-flags-to-ship-changes-with-confidence/ Accessed 24 April 2019, 2014.

[68] D. Piessens, "It's more than feature toggles." [Online]. Available: https://www.youtube.com/watch?v=TCwMxTuDLwI Accessed 27 January 2020, 2015.

[69] B. Day, "Get good at devops: Feature flag deployments with asp.net, webapi, & javascript." [Online]. Available: https://channel9.msdn.com/Events/Visual-Studio/Visual-Studio-Live-Redmond-2016/W07 Accessed 27 January 2020, 2016.

[70] J. Roberts, "Featuretoggle in .net." [Online]. Available: http://jason-roberts.github.io/FeatureToggle.Docs/ Accessed 24 April 2019, 2012.

[71] A. Tsvetkov, "Feature toggles in .net: tips and tricks." [Online]. Available: https://surfingthecode.com/feature-toggles-in-.net-tips-and-tricks/ Accessed 13 January 2020, 2017.

[72] T. Mclaughlin, "Introducing stormcrow." [Online]. Available: https://blogs.dropbox.com/tech/2017/03/introducing-stormcrow/ Accessed 24 April 2019, 2017.

[73] L. SN, "Merge hells!! feature toggles to the rescue - pipeline conference 2017." [Online]. Available: https://www.youtube.com/watch?v=R9EYY0Uu250 Accessed 24 April 2019, 2017.

[74] M. Hammarberg and J. Sunden, *Kanban in action*. Manning Publications Co., 2014.

[75] Erik, "Your feature flag management needs to include retirement." [Online]. Available: https://rollout.io/blog/feature-flag-retirement/ Accessed 24 April 2019, 2018.

[76] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, 2013.

[77] G. Rugg and P. McGeorge, "The sorting techniques: A tutorial paper on card sorts, picture sorts and item sorts," *Expert Systems*, vol. 22, no. 3, pp. 94–107, 2005.

[78] A. Danial, "Cloc: Count lines of code." [Online]. Available: http://cloc.sourceforge.net/, Accessed 24 July 2019, 2017.

[79] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning*, vol. 112. Springer, 2013.

[80] NAV - The Norwegian Labour and Welfare Directorate, "pleiepengesoknad." [Online]. Available: https://github.com/FeatureToggleStudy/pleiepengesoknad/blob/master/src/app/utils/featureToggleUtils.ts#L7 Accessed 27 February 2020, 2019.

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

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

[83] M. Sayagh, N. Kerzazi, F. Petrillo, K. Bennani, and B. Adams, "What should your run-time configuration framework do to help developers?," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1259–1293, 2020.

[84] UK Education and Skills Funding Agency Team, "Cfs-frontend." [Online]. Available: https://github.com/featuretogglestudy/CFS-Frontend/blob/39961217b8aabd665c71b108903d87014a41582c/DevOps/frontend-azure.dfe.json#L237 Accessed 27 February 2020, 2019.

[85] 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, 2019.

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

[87] UK HM Courts & Tribunals Service , "div-case-orchestration-service." [Online]. Available: https://github.com/hmcts/div-case-orchestration-service Accessed 24 August 2020, 2018.

[88] The Guardian, "Grid." [Online]. Available: https://github.com/guardian/grid Accessed 27 February 2020, 2013.

[89] Australian Department of Veterans' Affairs, "myservice-prototype." [Online]. Available: https://github.com/AusDVA/myservice-prototype Accessed 9 October 2019, 2017.

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

[91] UK Department for Education, "Multiplication Tables Check (MTC) Project." [Online]. Available: https://github.com/DFEAGILEDEVOPS/MTC/tree/0eb2d765b6683c90c852ba21c225742f07f050b9/admin/config Accessed 9 October 2019, 2019.

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

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

[94] M. A. Ould and C. Unwin, *Testing in software development.* Cambridge University Press, 1986.

[95] UK Department for Education, "Multiplication Tables Check (MTC) Project." [Online]. Available: https://github.com/DFEAGILEDEVOPS/MTC/commit/0eb2d765b6683c90c852ba21c225742f07f050b9#diff-8633026cf78840f2cb5a5b32fe1aa00f Accessed 11 October 2019, 2019.

[96] UK Department for Education, "Multiplication Tables Check (MTC) Project." [Online]. Available: https://github.com/DFEAGILEDEVOPS/MTC/commit/0eb2d765b6683c90c852ba21c225742f07f050b9 Accessed 27 February 2020, 2019.

[97] IEEE Standards Coordinating Committee, "IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos," *CA: IEEE Computer Society*, vol. 169, 1990.

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

[99] 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)*, pp. 30–39, IEEE, 2007.

[100] 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.

[101] H. Xi, "Dead code elimination through dependent types," in *Practical Aspects of Declarative Languages, First International Workshop, PADL '99, San Antonio, Texas, USA, January 18-19, 1999, Proceedings* (G. Gupta, ed.), vol. 1551 of *Lecture Notes in Computer Science*, pp. 228–242, Springer, 1999.

[102] L. Layman, L. Williams, and L. Cunningham, "Exploring extreme programming in context: an industrial case study," in *Agile Development Conference*, pp. 32–41, IEEE, 2004.

[103] P. Nour, *Ontology-based retrieval of software engineering experiences*. University of Calgary, Department of Computer Science, 2003.

[104] Repository, "unleash." [Online]. Available: https://github.com/Unleash/unleash, Accessed 20 January 2022, 2022.

[105] Repository, "rollout." [Online]. Available: https://github.com/fetlife/rollout, Accessed 20 January 2022, 2022.

[106] Repository, "flipper." [Online]. Available: https://github.com/jnunemaker/flipper, Accessed 20 January 2022, 2022.

[107] Repository, "Featuretoggle." [Online]. Available: https://github.com/jason-roberts/FeatureToggle, Accessed 20 January 2022, 2022.

[108] Repository, "fflip." [Online]. Available: https://github.com/FredKSchott/fflip, Accessed 20 January 2022, 2022.

[109] Repository, "Featuremanagement-dotnet." [Online]. Available: https://github.com/microsoft/FeatureManagement-Dotnet, Accessed 20 January 2022, 2022.

[110] Repository, "qandidate-toggle." [Online]. Available: https://github.com/qandidate-labs/qandidate-toggle, Accessed 20 January 2022, 2022.

[111] Repository, "flag." [Online]. Available: https://github.com/garbles/flag, Accessed 20 January 2022, 2022.

[112] Repository, "feature-flags." [Online]. Available: https://github.com/ylsideas/feature-flags, Accessed 20 January 2022, 2022.

[113] Repository, "flags." [Online]. Available: https://github.com/happykit/flags, Accessed 20 January 2022, 2022.

[114] Repository, "react-feature-toggles." [Online]. Available: https://github.com/paralleldrive/react-feature-toggles, Accessed 20 January 2022, 2022.

[115] Repository, "flip." [Online]. Available: https://github.com/pda/flip, Accessed 20 January 2022, 2022.

[116] Repository, "rollout for php." [Online]. Available: https://github.com/opensoft/rollout, Accessed 20 January 2022, 2022.

[117] Repository, "Featureswitcher." [Online]. Available: https://github.com/mexx/FeatureSwitcher, Accessed 20 January 2022, 2022.

[118] Repository, "flagged." [Online]. Available: https://github.com/sergiodxa/flagged, Accessed 20 January 2022, 2022.

[119] Repository, "Esquio." [Online]. Available: https://github.com/Xabaril/Esquio, Accessed 20 January 2022, 2022.

[120] Repository, "swivel." [Online]. Available: https://github.com/zumba/swivel, Accessed 20 January 2022, 2022.

[121] Repository, "feature." [Online]. Available: https://github.com/mgsnova/feature, Accessed 20 January 2022, 2022.

[122] Repository, "ngx-feature-toggle." [Online]. Available: https://github.com/willmendesneto/ngx-feature-toggle, Accessed 20 January 2022, 2022.

[123] Repository, "Featureswitch." [Online]. Available: https://github.com/valdisiljuconoks/FeatureSwitch, Accessed 20 January 2022, 2022.

[124] C. G. Christians and J. W. Carey, "The logic and aims of qualitative research," *Research methods in mass communication*, vol. 2, pp. 354–374, 1989.

[125] A. J. Viera, J. M. Garrett, *et al.*, "Understanding interobserver agreement: the kappa statistic," *Fam med*, vol. 37, no. 5, pp. 360–363, 2005.

[126] Chromium, "Chromium flags." [Online]. Available: `https://docs.google.com/spreadsheets/d/1XwGEpBidtPKS_xFzW8bfQ4kbuVe0OuPlpMsSd8wTmmo/edit#gid=0,Accessed1September2021`, 2021.

[127] M. Fowler, "Feature toggle." [Online]. Available: `https://martinfowler.com/bliki/FeatureToggle.html`, Accessed 23 August 2019, 2010.

[128] R. Mahdavi-Hezaveh, S. Fatima, and L. Williams, "Paving a path for a combined family of feature toggle and configuration option research," in *Submitted to 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2022.

[129] R. Mahdavi-Hezaveh, S. Fatima, and L. Williams, "Meta-analysis of software configuration research," in *Submitted to Empirical Software Engineering (EMSE)*, 2022.

[130] M. Kuhrmann, D. M. Fernández, and M. Daneva, "On the pragmatic design of literature studies in software engineering: an experience-based guideline," *Empirical software engineering*, vol. 22, no. 6, pp. 2852–2891, 2017.

[131] R. van de Schoot, J. de Bruin, R. Schram, P. Zahedi, J. de Boer, F. Weijdema, B. Kramer, M. Huijts, M. Hoogerwerf, G. Ferdinands, *et al.*, "An open source machine learning framework for efficient and transparent systematic reviews," *Nature Machine Intelligence*, vol. 3, no. 2, pp. 125–133, 2021.

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

[133] C. Croarkin and P. Tobias, *NIST/SEMATECH e-handbook of Statistical Methods*. NIST/SEMATECH, 2012. Available online: http://www. itl.nist.gov/div898/handbook.

[134] M. K. Ramanathan, L. Clapp, R. Barik, and M. Sridharan, "Piranha: Reducing feature flag debt at uber," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pp. 221–230, 2020.

[135] J.-M. Jézéquel, J. Kienzle, and M. Acher, "From feature models to feature toggles in practice," in *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume A*, pp. 234–244, 2022.

[136] R. Mahdavi-Hezaveh, N. Ajmeri, and L. Williams, "Feature toggles as code: Heuristics and metrics for structuring feature toggles," *Information and Software Technology*, p. 106813, 2022.

[137] H. Sugnanam and M. T. Rahman, "Classifying toggles-smells and investigating development effort," in *15th Innovations in Software Engineering Conference*, pp. 1–2, 2022.

[138] S. Ananieva, "Consistent management of variability in space and time," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume B*, pp. 7–12, 2021.

[139] R. Bhagwan, S. Mehta, A. Radhakrishna, and S. Garg, "Learning patterns in configuration," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 817–828, IEEE, 2021.

[140] T. Chen and M. Li, "Multi-objectivizing software configuration tuning," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 453–465, 2021.

[141] M. Cordy, S. Lazreg, M. Papadakis, and A. Legay, "Statistical model checking for variability-intensive systems: applications to bug detection and minimization," *Formal Aspects of Computing*, vol. 33, no. 6, pp. 1147–1172, 2021.

[142] A. S. Dimovski, "A binary decision diagram lifted domain for analyzing program families," *Journal of Computer Languages*, vol. 63, p. 101032, 2021.

[143] A. S. Dimovski, S. Apel, and A. Legay, "Program sketching using lifted analysis for numerical program families," in *NASA Formal Methods Symposium*, pp. 95–112, Springer, 2021.

[144] Y. Ding, A. Pervaiz, M. Carbin, and H. Hoffmann, "Generalizable and interpretable learning for configuration extrapolation," in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pp. 728–740, 2021.

[145] F. Ferreira, G. Vale, J. P. Diniz, and E. Figueiredo, "Evaluating t-wise testing strategies in a community-wide dataset of configurable software systems," *Journal of Systems and Software*, vol. 179, p. 110990, 2021.

[146] P. Franz, T. Berger, I. Fayaz, S. Nadi, and E. Groshev, "Configfix: interactive configuration conflict resolution for the linux kernel," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 91–100, IEEE, 2021.

[147] H. Göttmann, I. Bacher, N. Gottwald, and M. Lochau, "Static analysis techniques for efficient consistency checking of real-time-aware dspl specifications," in *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, pp. 1–9, 2021.

[148] X. Han, T. Yu, and M. Pradel, "Confprof: White-box performance profiling of configuration options," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pp. 1–8, 2021.

[149] R. Krishna, V. Nair, P. Jamshidi, and T. Menzies, "Whence to learn? transferring knowledge in configurable systems using beetle," *IEEE Transactions on Software Engineering*, 2020.

[150] L. Lesoil, M. Acher, X. Tërnava, A. Blouin, and J.-M. Jézéquel, "The interplay of compile-time and run-time options for performance prediction," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A*, pp. 100–111, 2021.

[151] L. Linsbauer, F. Schwägerl, T. Berger, and P. Grünbacher, "Concepts of variation control systems," *Journal of Systems and Software*, vol. 171, p. 110796, 2021.

[152] U. Markiegi, A. Arrieta, L. Etxeberria, and G. Sagardui, "Dynamic test prioritization of product lines: An application on configurable simulation models," *Software Quality Journal*, vol. 29, no. 4, pp. 943–988, 2021.

[153] K. Nguyen and T. Nguyen, "Gentree: Using decision trees to learn interactions for configurable software," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1598–1609, IEEE, 2021.

[154] J. Oh, N. F. Yıldıran, J. Braha, and P. Gazzillo, "Finding broken linux configuration specifications by statically analyzing the kconfig language," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 893–905, 2021.

[155] J. A. Pereira, M. Acher, H. Martin, J.-M. Jézéquel, G. Botterweck, and A. Ventresque, "Learning software configuration spaces: A systematic literature review," *Journal of Systems and Software*, vol. 182, p. 111044, 2021.

[156] M. Pinnecke, "Product-lining the elinvar wealthtech microservice platform," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume B*, pp. 60–68, 2021.

[157] B. Ramos-Gutiérrez, Á. J. Varela-Vaca, J. A. Galindo, M. T. Gómez-López, and D. Benavides, "Discovering configuration workflows from existing logs using process mining," *Empirical Software Engineering*, vol. 26, no. 1, pp. 1–41, 2021.

[158] T. Saber, M. Bendechache, and A. Ventresque, "Incorporating user preferences in multi-objective feature selection in software product lines using multi-criteria decision analysis," in *International Conference on Optimization and Learning*, pp. 361–373, Springer, 2021.

[159] S. A. Safdar, T. Yue, and S. Ali, "Recommending faulty configurations for interacting systems under test using multi-objective search," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–36, 2021.

[160] C. Sundermann, M. Nieke, P. M. Bittner, T. Heß, T. Thüm, and I. Schaefer, "Applications of# sat solvers on feature models," in *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, pp. 1–10, 2021.

[161] P. Temple, G. Perrouin, M. Acher, B. Biggio, J.-M. Jézéquel, and F. Roli, "Empirical assessment of generating adversarial configurations for software product lines," *Empirical Software Engineering*, vol. 26, no. 1, pp. 1–49, 2021.

[162] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner, "White-box analysis over machine learning: Modeling performance of configurable systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1072–1084, IEEE, 2021.

[163] M. Weber, S. Apel, and N. Siegmund, "White-box performance-influence models: A profiling and learning approach," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1059–1071, IEEE, 2021.

[164] J. Zhang, R. Piskac, E. Zhai, and T. Xu, "Static detection of silent misconfigurations with deep interaction analysis.," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[165] S. Ananieva, S. Greiner, T. Kehrer, J. Krüger, T. Kühn, L. Linsbauer, S. Grüner, A. Koziolek, H. Lönn, S. Ramesh, *et al.*, "A conceptual model for unifying variability in space and time: Rationale, validation, and illustrative applications," *Empirical Software Engineering*, vol. 27, no. 5, pp. 1–53, 2022.

[166] S. Ananieva, S. Greiner, J. Krüger, L. Linsbauer, S. Gruener, T. Kehrer, T. Kuehn, C. Seidl, and R. Reussner, "Unified operations for variability in space and time," in *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*, pp. 1–10, 2022.

[167] F. Bertolotti, W. Cazzola, and L. Favalli, "Features, believe it or not! a design pattern for first-class citizen features on stock jvm," in *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume A*, pp. 32–42, 2022.

[168] J. Cheng, C. Gao, and Z. Zheng, "Hinnperf: Hierarchical interaction neural network for performance prediction of configurable systems," *ACM Transactions on Software Engineering and Methodology*, 2022.

[169] F. Damiani, M. Lienhardt, and L. Paolini, "On logical and extensional characterizations of attributed feature models," *Theoretical Computer Science*, vol. 912, pp. 56–80, 2022.

[170] A. S. Dimovski, S. Apel, and A. Legay, "Several lifted abstract domains for static analysis of numerical program families," *Science of Computer Programming*, vol. 213, p. 102725, 2022.

[171] C. Dubslaff, K. Weis, C. Baier, and S. Apel, "Causality in configurable software systems," *arXiv preprint arXiv:2201.07280*, 2022.

[172] R. Heradio, D. Fernandez-Amoros, J. A. Galindo, D. Benavides, and D. Batory, "Uniform and scalable sampling of highly configurable systems," *Empirical Software Engineering*, vol. 27, no. 2, pp. 1–34, 2022.

[173] Y. Hou, D. Ouyang, X. Tian, and L. Zhang, "Evolutionary many-objective satisfiability solver for configuring software product lines," *Applied Intelligence*, pp. 1–24, 2022.

[174] C. Kröher, M. Flöter, L. Gerling, and K. Schmid, "Incremental software product line verification-a performance analysis with dead variable code," *Empirical Software Engineering*, vol. 27, no. 3, pp. 1–41, 2022.

[175] W. Mahmood, D. Strüber, A. Anjorin, and T. Berger, "Effects of variability in models: a family of experiments," *Empirical Software Engineering*, vol. 27, no. 3, pp. 1–38, 2022.

[176] H. Mercan, A. Aytar, G. Coskun, D. Mustecep, G. Uzer, and C. Yilmaz, "Cit-daily: A combinatorial interaction testing-based daily build process," *Journal of Systems and Software*, vol. 190, p. 111353, 2022.

[177] G. K. Michelon, D. Obermann, W. K. Assunção, L. Linsbauer, P. Grünbacher, S. Fischer, R. E. Lopez-Herrejon, and A. Egyed, "Evolving software system families in space and time with feature revisions," *Empirical Software Engineering*, vol. 27, no. 5, pp. 1–54, 2022.

[178] M. Nieke, G. Sampaio, T. Thüm, C. Seidl, L. Teixeira, and I. Schaefer, "Guiding the evolution of product-line configurations," *Software and Systems Modeling*, vol. 21, no. 1, pp. 225–247, 2022.

[179] J. A. Prado Lima, W. D. Mendonça, S. R. Vergilio, and W. K. Assunção, "Cost-effective learning-based strategies for test case prioritization in continuous integration of highly-configurable software," *Empirical Software Engineering*, vol. 27, no. 6, pp. 1–45, 2022.

[180] G. A. Randrianaina, X. Tërnava, D. E. Khelladi, and M. Acher, "On the benefits and limits of incremental build of software configurations: An exploratory study," in *44th International Conference on Software Engineering (ICSE 2022)*, 2022.

[181] G. A. Randrianaina, D. E. Khelladi, O. Zendra, and M. Acher, "Towards incremental build of software configurations," in *2022 IEEE/ACM 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 101–105, IEEE, 2022.

[182] X. Tërnava, M. Acher, L. Lesoil, A. Blouin, and J.-M. Jézéquel, "Scratching the surface of./configure: Learning the effects of compile-time options on binary size and

gadgets," in *ICSR 2022-The International Conference on Software and Systems Reuse*, 2022.

[183] 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.

[184] M. B. Cohen, J. Snyder, and G. Rothermel, "Testing across configurations: implications for combinatorial testing," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 6, pp. 1–9, 2006.

APPENDICES

APPENDIX

A

SURVEY QUESTIONS FOR CHAPTER 3

The survey questions are as follows:

1. What is your company name?
2. How long has your team used feature toggles?
3. What feature toggle management system is used by your team? (Check all that apply). Options: Closed source custom system maintained by the company; Open source custom system maintained by the company; Third-party (e.g. LaunchDarkly), Open source but not maintained by the company; None; Other.
4. For what purpose(s) does your team use feature toggles? (Check all that apply). Options: Support CI of partially-completed features; Dark launches; A/B testing; Gradual rollout; Other.
5. Does your team make a decision about using feature toggles for each feature? Options: Yes. The team checks to find if using a feature toggle is necessary for the new feature; No. The feature toggle is always added when a new feature is added; Other.
6. How often does your team do the following management practices? Options: Always, Mostly, About half of the time, Rarely, and Never.
   - Document feature toggle's metadata (spreadsheet, etc) to manage data about feature

toggles. (i.e. the owner of the toggle, the current value (on, off), the current status (to remove, keep), and the time of its creation).

- Logging changes to toggle values/configurations (e.g. who changes which toggle and when, etc.).
- Grouping toggles together in any way to simplify management or giving permissions (i.e. related toggles, other).
- Allowing all team members (i.e. Q&A team) to have access to feature toggles and can make changes.

7. How often does your team do the following initialization practices? Options: Always, Mostly, About half of the time, Rarely, and Never.
   - Determining the type (permission toggle, ops toggle, release toggle, experiment toggle, short-lived toggle, long-lived toggle) of the toggle at the design step. (More information about types of toggles: `https://goo.gl/4okG5Y`)
   - Using naming conventions for toggles (similar to variable and function naming conventions).
   - Setting up a default value for toggle if toggle value is not found (i.e. toggle is off if its value is not found in the code).
8. How are the values of the toggles stored? (Check all that apply) Options: Configuration files; Databases; Other.
9. How are the values are assigned to the toggles in the system? (Check all that apply) Options: Assigned boolean values (True, False); Assigned multivariate values (e.g. Red, Yellow, Blue); Assigned string values (e.g. "disable-flash-3d", "enabled-flash-3d"); Other.
10. How does a developer access the toggle value in the code? (Check all that apply) Options: Value is accessed by checking a primitive data type (e.g. enableMyFeature == true),; Value is accessed through an object representing a toggle (e.g. MyFeature.isActive()); Value is accessed through a toggle manager/mapping from key to value (e.g. Dictionary); Other.
11. How often does your team do the following clean-up practices? Options: Always, Mostly, About half of the time, Rarely, and Never.
    - Limiting the number of existing toggles in the code.
    - Build or test failing if a toggle is not deleted by a specified date (Time bomb).
    - Automatic reminders near date to delete the toggle.

- Using tasks/stories/cards for removing toggles.
- Creating a clean-up branch for removing toggle points at the time of the creation of the toggle.
- Tracking unused toggles for removal.
- Changing feature toggle to configuration setting to keep it in the code.

APPENDIX

B

PRELIMINARY STATISTICAL ANALYSIS FOR CHAPTER 4

In this section, we report on the preliminary statistical analysis we conduct on the case study dataset in Chapter 4

We perform *Best Subset Selection* (79) and identify the best subset of predictors (including context metrics and FT-heuristics) to improve each one of the FT-metrics. Algorithm 1 outlines this process to identify the best subset for each FT-metrics. Since FT-heuristics are categorical predictors, we have created dummy variables for them (79). So, we have 17 predictors to use in our regression analysis. Following Algorithm 1, for each FT-metric, we fit 17 models with 1 predictor, 136 models with 2 predictor, 680 models with 3 predictor, 2,380 models with 4 predictor, 6,188 models with 5 predictor, 12,376 models with 6 predictor, 19,448 models with 7 predictor, 24,310 models with 8 predictor, 24,310 models with 9 predictor, 19,448 models with 10 predictor, 12,376 models with 11 predictor, 6,188 models with 12 predictor, 2,380 models with 13 predictor, 680 models with 14 predictor, 136 models with 15 predictor, 17 models with 16 predictor, and 1 model with 17 predictors. In the set of models for any number of predictors, we select the best one. Then from 17 selected models, we select a single *best* model as the overall best model.

Algorithm 1 Best Subset Selection algorithm adapted from (79)

For each FT-metric:
  For $k = 1, 2, \ldots, p$: # Identify the best model for each k
    if FT-metric is numeric **then**
      (1) Fit all $\binom{p}{k}$ linear regressions that contain exactly k predictors
      (2) Select μ_k = Model with largest adjusted R^2 (best model)
    else if FT-metric is binary **then**
      (1) Fit all $\binom{p}{k}$ logistic regressions that contain exactly k predictors
      (2) Select μ_k = Model with lowest LLR p-value
    end if
  # Select a single best model from $\mu_1, \ldots, \mu_p$
  if FT-metric is numeric **then**
    (1) Conduct ANOVA to compare the models
    (2) Select $\text{bestmodel}_{\text{metric}}$ = Model with lowest F-test p-value where $p < 0.05$
  else if FT-metric is binary **then**
    (1) Compare LLR p-values of the models
    (2) Select $\text{bestmodel}_{\text{metric}}$ = Model with lowest LLR p-value where $p < 0.05$
  end if

For each FT-metric, we report the predictors of the overall best model in Table B.1. In Table B.1, ↑ indicates the predictor in the best subset with a positive coefficient, and ↓ indicates the predictor in the best subset with a negative coefficient.

Comparing the result in Table B.1 and Table 4.2 confirms the correctness of six of the hypothesized relations. For example, we observed that repositories that do not follow H4 (UseSparingly) have a higher number of the locations for feature toggles (M8 (Location)) and, repositories that do not follow H7 (CompleteRemoval) of feature toggles have more dead code (M11 (Dead)). In addition, we find 26 new relations (non-hypothesized) between FT-metrics and FT-heuristics. For example, we find a relationship between H1 (Shared-Method) and M10 (Duplicate), meaning that in our case study set when repositories do not have a shared method, they have more duplicate code. To strengthen these newly identified relations, future works could conduct a larger-scale empirical study.

Table B.1: Best Subset Selection result for FT-metrics. ↑ indicates the predictor is in the overall best model with a positive coefficient. ↓ indicates the predictor is in the overall best model with a negative coefficient.

	Metrics	#contributors	#feature toggles	H1(SharedMethod-F)	H1(SharedMethod-NF)	H2(SelfDescriptive-F)	H2(SelfDescriptive-NF)	H3(Guidelines-F)	H3(Guidelines-NF)	H4(UseSparingly-F)	H4(UseSparingly-NF)	H5(AvoidDuplicate-F)	H5(AvoidDuplicate-NF)	H6(Testing-F)	H6(Testing-NF)	H7(CompleteRemoval-F)	H7(CompleteRemoval-NF)	H7(CompleteRemoval-Unknown)
Complexity	M1 (Paths)																	
	M2 (Methods)	↑							↑									
Comprehensibility	M3 (Guidelines)	↓	↑						↓									
	M4 (Intention)	↑	↑	↓	↑					↑	↓					↑		
	M5 (Comments)					↑	↓											
	M6 (Description)		↑	↑			↓	↑	↓		↓		↓		↑		↓	
Maintainability	M7 (Files)																	
	M8 (Locations)										↑			↑				
	M9 (LOC)														↓	↑		
	M10 (Duplicate)	↑			↑								↑			↓		↓
	M11 (Dead)																↑	
	M12 (Test cases)		↓	↑	↓	↓	↑											↓