



# On the Removal of Feature Toggles

## A Study of Python Projects and Practitioners Motivations

Juan Hoyos<sup>1</sup> · Rabe Abdalkareem<sup>2,3</sup> · Suhaib Mujahid<sup>2</sup> · Emad Shihab<sup>2</sup> · Albeiro Espinosa Bedoya<sup>1</sup>

Accepted: 14 October 2020 / Published online: 03 February 2021  
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC part of Springer Nature 2021

### Abstract

Feature Toggling is a technique to control the execution of features in a software project. For example, practitioners using feature toggles can experiment with new features in a production environment by exposing them to a subset of users. Some of these toggles require additional maintainability efforts and are expected to be removed, whereas others are meant to remain for a long time. However, to date, very little is known about the removal of feature toggles, which is why we focus on this topic in our paper. We conduct an empirical study that focuses on the removal of feature toggles. We use source code analysis techniques to analyze 12 Python open source projects and surveyed 61 software practitioners to provide deeper insights on the topic. Our study shows that 75% of the toggle components in the studied Python projects are removed within 49 weeks after introduction. However, eventually practitioners remove feature toggles to follow the life cycle of a feature when it becomes stable in production. We also find that not all long-term feature toggles are designed to live that long and not all feature toggles are removed from the source code, opening the possibilities to unwanted risks. Our study broadens the understanding of feature toggles by identifying reasons for their survival in practice and aims to help practitioners make better decisions regarding the way they manage and remove feature toggles.

**Keywords** Feature toggles · Continuous integration · Continuous delivery · Empirical studies

## 1 Introduction

Continuous delivery is rapidly gaining traction in the development of software systems. Previous work has shown the many advantages of continuous delivery, including higher quality

---

Communicated by: Sarah Nadi

✉ Juan Hoyos  
jdhoyosr@unal.edu.co

Extended author information available on the last page of the article.

and user satisfaction (Adams and McIntosh 2016). Today, all major software companies such as Netflix, Facebook, Microsoft, Mozilla, and Google have adapted their development and business processes to incorporate continuous delivery (Zapata 2014; Feitelson et al. 2013; Harry 2012; Adams et al. 2015; Mäntylä et al. 2015; Claps et al. 2015).

One of the main supporting techniques of continuous delivery is *feature toggles*<sup>1</sup> (Rahman et al. 2015). Feature toggles provide a mechanism to hide/show certain features from being executed. The purpose of using this technique is to allow teams to control the execution of features at will and modify the behavior of running software.

Although feature toggles provide a powerful tool to software teams, history has shown that they can also lead to catastrophic outcomes. For example, in 2012 Knight Capital Group lost approximately \$460 million dollars because code behind a repurposed feature toggle was incorrectly deployed (Securities and E. Commission 2013). Clearly, proper management of these feature toggles is key. In fact, Neely and Stolt reported the need to remove toggles in order to avoid a “mess of feature specific logic” (Neely and Stolt 2013). Other efforts were more explicit, for example, when the Google Chrome developers started a campaign to cleanup an unwanted number of toggles lingering in their source code (Rahman et al. 2016).

We specifically focus on the study of toggles removal since the clean-up of feature toggles can be complex, and if done incorrectly, can lead to major failures (Rahman et al. 2016; Securities and E. Commission 2013). Also, while the control of the toggles inventory is an activity suggested by prior work (Hodgson 2016; Rahman et al. 2016), yet very little is known about the removal of toggles in reality (Neely and Stolt 2013; Rahman et al. 2016). Our goal is to provide empirical evidence about what types of toggles are removed, when and why. Doing so, provides unique insights to practitioners and researchers in the area and enables us to better manage toggles.

Therefore, in this paper, we conduct an in-depth, mixed-method empirical study to better understand *the removal of feature toggles* in open source software projects. Specifically, we employ two empirical approaches. First, we perform a quantitative source code analysis to determine the lifetime of toggle components through analyzing the source code of 12 Python projects to better understand for how long they live in the software. Then, we perform a qualitative survey to inquire 61 practitioners on their management and removal practices of feature toggles.

Our study is formalized through the following main research questions:

- **RQ1.** *How long do toggles remain in a project before they are removed?* A previous study on Google Chrome highlighted that a significant portion of feature toggles remain in the source code across multiple releases (Rahman et al. 2016), although others have advocated to keep them under a manageable number. We want to uncover whether this characteristic extends to other projects that are using feature toggles.
- **RQ2.** *Why and when practitioners remove toggles?* We do not know if the removal trend from our qualitative study remains for other projects and we want to understand the conditions practitioners have to control the inventory of feature toggles in their projects.

Our findings indicate that 1) feature toggles allow practitioners to execute features with precise control, mostly to leverage *trunk-based* development, *dark launches* and *kill switches*; 2) 75% of the feature toggle components in the studied Python projects are removed from the code within 49 weeks, although some of them are not intended to live for

---

<sup>1</sup>Toggles are also called flags, bits, flippers, switches or gates. In this paper, we use the term toggles.

such a long-term; and 3) most practitioners remove feature toggles guided by the life cycle of the feature, when regular audits are exercised, or when refactoring code; a minority (5%), keep *kill switches* for long periods or do not remove feature toggles. In particular, our paper makes the following contributions:

- We perform a study that focused on the removal of feature toggles in Python open source projects.
- Our study is one of the largest studies on feature toggles, involving the analysis of 72 projects and a survey of more than 60 practitioners.
- A publicly available dataset of open source projects that are using feature toggles and an extensible tool to extract feature toggles across multiple Python projects.<sup>2</sup>

**Paper Organization** The remainder of this paper is organized as follows. In Section 2, we provide background on the components of feature toggles. In Section 3, we discuss the related work. Section 4 describes our study setup. Feature toggles usage patterns are reported in the preliminary Section 5. In Section 6 and Section 7, we present and discuss the results of our empirical study. Section 8 discusses the threats to validity of our study. Finally, Section 9 concludes our paper.

## 2 Background

This section introduces the architectural elements of a toggling subsystem in a software project and describes the usage patterns of feature toggles. Both toggling subsystem elements and usage patterns, are fundamental to the rest of our study.

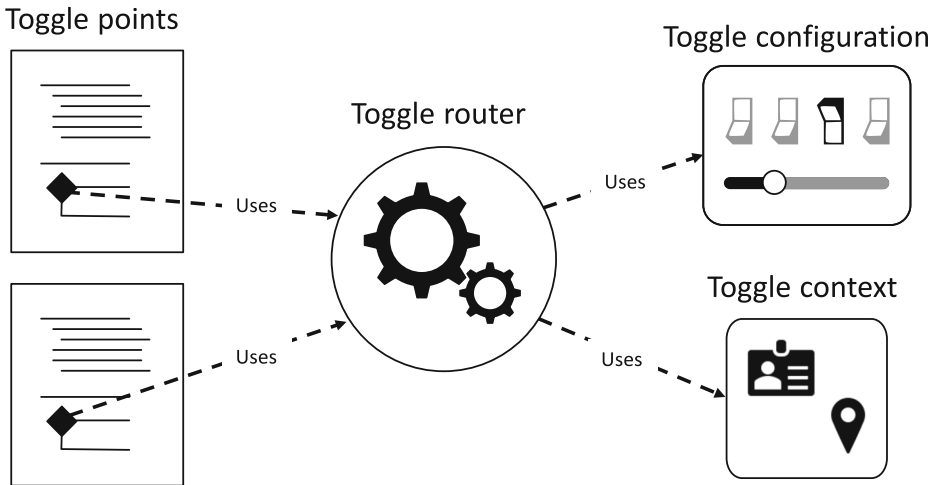
### 2.1 Toggling Subsystem

In this subsection, we briefly introduce the architectural elements that model the behaviour of feature toggles in a software system as described in the literature. We later use these elements for our extraction purposes in the study setup (Section 4.3).

Rahman et al. (2016) observed that feature toggles in Google Chrome are located in multiple configuration files where they can be enabled or disabled. These feature toggles are used directly inside “conditional statements” to decide on the execution of the evaluated features. Alternatively, Rahman et al. (2016) noticed that toggles are also assigned to a variable for later evaluation and allow “more flexible and complex usages”.

According to Hodgson (Hodgson 2016), toggled projects have a common set of elements: toggle points, routers, configurations, and context. Figure 1 shows these four main parts and how they are connected. When the regular code of a project is executed, the toggle points are executed as well. These toggle points modify the behaviour of the project by means of evaluating different logical conditions and deciding on the execution of certain paths of the code. The evaluated logical conditions include the results of toggle routers that make “decisions” based on a toggle configuration and an optional toggle context. Hodgson indicates the toggle configuration affects the routing decision in a static way when the toggle router simply relays the state “On or Off” to the toggle point, or dynamically when the toggle router is instructed by the configuration to decide a state depending on certain conditions, like if the user belongs to the quality assurance team or the user matches the target country of an A/B

<sup>2</sup>A replication package is available on [https://github.com/elhoyos/toggles\\_package](https://github.com/elhoyos/toggles_package)



**Fig. 1** A toggling subsystem. The toggle points modify the execution of a feature using a router, which in turn, makes decisions using the toggle configuration and context. Adapted from Hodgson (Hodgson 2016)

test. In turn, the toggle context refers to this complementary data required for the toggle router to make decisions, e.g. the user identity, usually not provided by the toggle configuration but other project subsystems. Furthermore, Hodgson distinguishes between static and dynamic configuration management. The configuration of toggles is static when managed using the version control system (e.g. git) and re-deployments. Alternatively, the configuration of toggles can be managed dynamically when the implemented technologies allow a more rapid re-configuration (e.g. key/value stores, application databases, etc.) (Sayagh et al. 2018; Kästner 2019).

## 2.2 Usage Patterns

In this subsection, we introduce a set of usage patterns where feature toggles play a central role to help practitioners gain better control of the development life cycle, the delivery pipeline or the operation of their software. We obtained these patterns from relevant studies and from our related work in Section 3, after we found that their naming and goals remain similar across these sources. The identification of usage patterns of feature toggles is important to support the design of our survey as we will detail later in Section 4.4.

**Trunk-Based Development** Practitioners contribute their code into the main code branch and hide their work from execution behind a feature toggle, while they keep the project releasable at all times (Hodgson 2016; Rahman et al. 2015; Neely and Stolt 2013; Schermann et al. 2016). According to Hodgson (2016), this practice is an enabler for continuous delivery because the integration of the work is checked-in early in the development process. Additionally, Rahman et al. (2015) mentions that trunk-based development reduces the total merge effort (i.e. big-bang merges) when compared to a *feature branches* approach, in which a feature is developed in an isolated branch and is merged into the main branch only when it is completely ready. Also, Neely and Stolt (2013) report a reduction in the pain and risks of merging long-running branches in their experience using this practice.

**Dark Launches** Practitioners release features into production environments without users noticing and evaluate how features perform under real conditions with the help of feature toggles (Neely and Stolt 2013; Schermann et al. 2016). Adams and McIntosh (2016) describe “dark launching” as releasing new features without exposing them publicly, and indicate it is commonly employed to test the behaviour of the features under real loads. For example, Facebook used a tool called Gatekeeper to install code in all their servers that had the client applications to send “dummy chat messages” to a new chat server they were load-testing without the users noticing (Feitelson et al. 2013).

**Kill Switch** Practitioners use feature toggles to quickly disable long-term features when not working as expected (Hodgson 2016). Hodgson explains that it is not uncommon to see long-lived feature toggles when practitioners realize these features could at some point degrade the behaviour of the system, thus, they use this mechanism to quickly return the system to a desirable state.

**A/B or Multivariate Testing** Practitioners use feature toggles to deliver multiple versions of features to subsets of users with the intention to statistically validate each version against technical or business expectations (Hodgson 2016; Schermann et al. 2016). Consequently, A/B tests solutions can be built on top of toggling implementations using a context to select the test variations and deliver the appropriate version to the right set of users. For example, Facebook’s Andrew Bosworth explained in 2012 that they ran hundreds of tests every day using Gatekeeper to control tests from colliding each other and to further obtain “statistically meaningful results” (Bosworth 2012).

**Canary Releases** Practitioners can use feature toggles to release a version of a project to a subset of users and validate the features in-the-field (Neely and Stolt 2013; Hodgson 2016; Humble and Farley 2010). This type of feature toggle allows the practitioners to enable features to the desired users in a controlled manner.

**Blue-Green Deployments** Practitioners can minimize the downtime of their projects while moving all the users from the current version of a software project (green) to the new version (blue) (Hodgson 2016; Humble and Farley 2010). This pattern occurs when coordinating the release of a product-centric version of a software; for example, a revamped website that should be built and later launched in coordination with a marketing campaign. When compared to canary releases, Humble and Farley (2010) explain blue-green deployments switch all the users to one of the versions of the project while canary releases granularly expose features to a subset of them. In this sense, blue-green deployments can be leveraged, for example, if feature toggles are implemented in a fully-managed middleware application load balancing the traffic to both green and blue front-end versions.

### 3 Related Work

While feature toggles have been the center of attention in multiple studies, most of which focus on the use and management of feature toggles. Rahman et al. (2016) examined the feature toggles across 35 releases of Google Chrome. They observed that the use of feature toggles increased linearly over time except for a temporary reduction, explained by an internal removal campaign. They also noticed that no less than 30% of the toggles are assigned to variables for further check, contrary to toggles used directly inside a conditional statement.

In addition, they found that only  $\sim 3\%$  of the changes in feature toggles occur during the release stabilization phase as opposed to the development phase. More interestingly, they concluded that 50% of toggles survived more than 12 releases and that 53% of short-lived toggles remaining in the code for more than 10 releases suggested technical debt.

Later, Sayagh et al. (2018) associated the feature toggles in Google Chrome (Rahman et al. 2016) with “configuration options”. They indicated feature toggles “should be actively maintained” to avoid undesirable dead and unused options. In this sense, feature toggles should be removed from the source code once their features become stable to reduce the risk of turning into dead options, and that unused options should become a constant given their “users” do not change their value often. In summary, the authors reported these suggestions under the “pro-active dead option detection” expert recommendation, to maintain a healthy configuration in a software system.

Other work reported the experiences of adopting feature toggles in software companies. For example, Neely and Stolt (2013) reported Rally Software used feature toggles to approach the challenges faced towards continuous delivery. The company shrunk their release cycles from eight-weeks to “at-will” in decreasing steps. They created a shared understanding of work in progress (WIP) limits and managed it into their workflows with the help of continuous delivery techniques. Feature toggles allowed the engineering team at Rally Software to put in practice a no-branch policy, reducing the story sizes and increasing the commit frequency. Specifically, they created a “framework of conditional logic” to hide their WIP whenever needed, and built an interface to operate the state of the toggles depending on the users running the application. Canary releases and Dark launches where built-in benefits this technique provided to Rally Software. Neely and Stolt also documented the experienced complexities when using feature toggles. They highlight that toggles rarely overlap, reducing the need to test every single combination of the toggle. However, the engineering team must be aware which tested toggle code path is turned on in production. Finally, they found non-retired toggles transform the codebase into “a mess” and adapted their workflows to write a story to remove a toggle once the feature behind it is decided to be available for a general release.

Schermann et al. (2016) interviewed and surveyed a diverse set of practitioners regarding the practices allowing continuous delivery. The survey results evidenced 36% of the participants use feature toggles for partial rollouts. Similarly, the practice is common across different company sizes: startups (50%), small-medium enterprises (35%) and corporations (32%). Furthermore, the practice is not only employed in Web-based applications (45%). Also, a significant portion of the surveyed participants using Canary Releases (57%) and A/B Testing (50%) indicated an architectural barrier to adopt partial rollouts. Contrary to the survey results, only 15% of the interviewed practitioners used toggles for partial rollouts and expressed operating “basic” tooling to manage them. Finally, Schermann et al. suggested the necessity for a proper architectural support to reduce the adoption barriers of partial rollouts and to draw attention to aspect oriented architecture or product line engineering.

Hodgson (2016) categorized feature toggles and broke them up into components. The author analyzed feature toggles depending on their toggling dynamism, longevity and intention. In this sense, he organized toggles in four categories: Release, Ops, Experiment and Permission. Hodgson provided examples of associated costs when using feature toggles. He stated that toggles introduce a “validation complexity” due to an exponential increase of test cases for each new code path affected by a toggle, and suggested to test only the

code paths that are expected to be in production. And more importantly, he mentioned that “savvy teams” manage feature toggles as inventory and “keep the number of feature flags manageable” by proactively removing the ones the team does not need.

In other work, Rahman et al. (2018) studied the feature toggle architecture of Google Chrome. The authors mapped the definition of toggles to the architectural modules spanning four major releases and observed the resulting relationships from two different views. The feature’s perspective shows the modules a feature spans while the module’s perspective identifies the features a module contains. Their results revealed these two views differ significantly and change over time. The feature toggles architecture was constructed to help developers understand the feature-module relationship and locate feature complexities or modularity violations.

Our work differs from the previous work, since we focus on the removal of feature toggles in multiple open source projects, rather than its management and usage on a single project. Also, our interest is the removal focuses on feature toggles, rather than configuration engineering. To the best of our knowledge, the relationship between feature toggles and configuration engineering is not clear and well understudied. In our empirical study, we quantitatively examine the removal of feature toggles in 12 Python projects, and we survey 61 practitioners to understand why and when they remove feature toggles. In many ways, our study complements and increases our prior understanding of the life cycle of feature toggles.

## 4 Study Setup

The primary *goal* of this paper is to better understand the removal of feature toggles in software projects. To achieve our goal, we first performed a quantitative study to examine the removal of feature toggles in the selected software projects. Then, we surveyed 61 practitioners to investigate why and when feature toggles are introduced and removed from software projects.

### 4.1 Identifying Toggled Projects

To perform our quantitative analysis, we needed a representative sample of projects that use feature toggles. To identify these projects, we followed a three-staged approach: 1) we selected a set of libraries that provide building blocks to implement feature toggles, 2) we searched for the dependent projects of the selected libraries in two data sources, and 3) we filtered the projects that will serve useful for our quantitative study.

We manually selected 53 *libraries* with feature toggle capabilities to be used in software projects. To do so, we collected all the related libraries that are available in public web pages associated with the studied technique, namely, *featureflags.io* (LaunchDarkly 2015), *djangopackages.org* (Django Packages : Feature Flipping 2018), and *enterprisesdevops.org* (Osherove 2016). These three websites contain resources dedicated to inform software developers about features toggles, specifically.

We chose open source libraries that were non-deprecated and ready-for-production according to their authors. Table 1 presents the programming languages, examples of the libraries, and their distribution across all languages.

With the list of libraries, we identified open source projects that depend on them. To achieve this, we searched for projects via GitHub’s REST API v3 (GitHub 2011a) and Libraries.io (Libraries.io - The Open Source Discovery Service 2015). We queried GitHub

**Table 1** Libraries per Language

Languages	Example of Libraries	%Librs.
JavaScript, TypeScript	flipit, launchdarkly, feature-toggles	18.9%
Ruby	rollout, Flip, Setler,	17.0%
Python	launchdarkly, flagon, Waffle	15.1%
C#, Visual Basic	NFeature, FeatureSwitcher, nToggle	15.1%
Java, Kotlin	launchdarkly, toggle, Toggglz	13.2%
PHP	launchdarkly, rollout,	9.4%
Go	launchdarkly, Toggle, dcdr	7.5%
Scala	Bandiera	1.9%
Objective-C, Swift	launchdarkly	1.9%

between December 13th – 14th of 2019, using a two-phased approach. First, we searched for various relevant terms to each library against the code search API endpoint (GitHub 2011b). To overcome the limitation of maximum 1,000 results and to control for partial results when a search is slow for GitHub to process (GitHub 2011b), we bisected by file size any affected search. Thus, we created two new searches that covered in tandem the file-size bounds of the previous search. The upper bound of the bisect was limited by GitHub to files smaller than 384 KB (GitHub 2013). In the second phase, we matched the resulting code files with regular expressions built with dependency declarations, usages or imports of each library. Subsequently, we queried the latest available dataset of Libraries.io in Google BigQuery, which contained data that was last modified by March 20th, 2019. In this case, we executed a SQL query to find dependent projects from the artifacts of the libraries. We decided to selected these two data sources to augment the recall and because we found there is a significant amount of non-overlapping results across them; we will present these findings later in this section. At this point, we have a list of 2,273 projects listed in GitHub that reference a toggling library in their source code.

As recommended by prior work (Kalliamvakou et al. 2014), we eliminated dummy projects that may exist on GitHub. To do that, we augmented the remaining projects with additional GitHub metadata like the parent repository, the id of the first commit, the creation date, the number of commits, and the size in bytes. This augmentation helped us to choose non-forked projects with more than 100 commits and a size of more than 1MB. As a final step, we manually ran through the remaining results to spot and remove projects not of our interest (e.g. sole dependency references, course templates, library clones). After this filtering, 90 (approximately 22%) projects were found in both data sources, 240 (58%) of the projects identified via GitHub's code search were unique to that data source, and 82 (20%) projects were unique to Libraries.io. In total, we obtained 412 projects that depend on one of the toggling libraries.

Table 2 shows the name of the top five used toggling libraries, the programming languages, and the number and participation of projects. Overall, we found that from the 412 projects in our dataset, 39% are written in Python, 23% in Ruby and 18% in Java. Also, we only found projects for 35 out of the initial list of 53 toggling libraries.

Then, we focused on the study of projects that use the Waffle library since 1) we manually examined each project and wanted a dataset that is sufficiently large, but at the same time manageable in size to analyze manually, 2) the extraction of toggles from projects is not a trivial task as it requires the comprehension of a toggling library and the knowledge of its



**Table 2** Toggled Projects per Library

Library	Languages	#Projects (%)
Waffle	Python	132 (32%)
Togglz	Java, Kotlin	38 (9.2%)
rollout	Ruby	30 (7.2%)
Flipper	Ruby	28 (6.8%)
unleash-client-java	Java, Kotlin	22 (5.3%)

underlying programming language. The cumulative proficiency we can contribute to this study is more towards Python based on the top five libraries of Table 2. Furthermore, our findings show that Waffle is the most used toggling library with 32% (132) of the projects in our dataset. For these two reasons, we decided to study projects that use the Waffle library.

## 4.2 Toggle Components

In this section, we present the toggle components for the purposes of our study, and we compare them to the elements of a toggling subsystem as previously introduced in Section 2.1.

We classify the components that integrate feature toggles to support static analysis. In the further sections of this study we identify and extract feature toggles from the source code of several projects using static analysis techniques. In consequence, this process requires a structured definition of what a feature toggle looks like in the source code.

To perform our classification, we combined the implementation information from the existing literature with the results of a manual analysis of two Python projects, `osf.io` (Center for Open Science 2013) (53,321 commits) and `edx-platform` (`edx-platform` 2011) (52,538 commits). These two projects use Waffle (Django Waffle — `django-waffle` 0.14.0 documentation 2018), a library to implement feature toggles in applications built on top of the Django web framework (The Web framework for perfectionists with deadlines — Django 2019) for Python (Ray et al. 2014). We selected these two projects because they have more commits than any other project with feature toggles in our dataset. Lastly, we grouped the intended effects of the relevant source code and we consolidated the resulting groups in the following toggle components: *Declaration*, *Router* and *Point*.

A *Declaration* is a toggle configuration in Hodgson terms (Hodgson 2016). Declarations map a feature toggle with a unique identifier, usually a human-friendly name. Declarations could include a default state or logic necessary for a Router to evaluate. It is not mandatory to store the default state of a toggle in the source code, some implementations use databases or external services for this purposes, hence, Declarations are optional in our studied projects. The Code Listing 1 shows an example of a Declaration.

*Routers* take a feature toggle identifier, evaluate the Declaration with an optional context and return a value, i.e. the state of the feature toggle at a specific execution. A context can be

```
DEBUG_MESSAGE_WAFFLE_FLAG = WaffleFlag (
    WAFFLE_FLAG_NAMESPACE, u'enable_debugging'
)
```

**Listing 1** A Declaration example named after a combination of `WAFFLE_FLAG_NAMESPACE` and `enable_debugging`. Adapted from `edx-platform` (`edx-platform` 2011)

```
def is_secondary_email_feature_enabled():
    return waffle.switch_is_active(
        ENABLE_SECONDARY_EMAIL_FEATURE_SWITCH
    )
```

**Listing 2** Example of a Router wrapped in a function. Adapted from `edx-platform` (edx-platform 2011)

any object or value relevant to the Router at the moment of evaluation. Examples of a context are a user entity or a session id. Prior work (e.g., Rahman et al. 2016) have shown that Routers can be wrapped in other functions or the returned value can be assigned to a variable. The Listing 2 presents an example of wrapping a router. In this example, the function `is_secondary_email_feature_enabled` wraps the Router `switch_is_active`, from the Waffle library.

Our Router definition resembles to the toggle router seen in the different implementation techniques suggested by Hodgson, however, we also embrace the observation of Rahman et al. when the value of their router is assigned to a variable.

A *Point* is where the feature branching happens. These components use the values from the Routers to decide which code branch is executed. From our manual analysis, Points were always present in the code. We also consider all code branches as part of the Point, thus, this concept extends the definition of a point stated in other studies. Listing 3 shows an example of a Point.

Interestingly, we also observed other forms of toggle states wrapping behind special language-dependant expressions or statements. For example, Python provides Decorators as a way to extend functions (PEP 318 – Decorators for Functions and Methods — Python.org 2003). Listing 4 depicts the usage of this pattern serving both as a Router and a Point using the `@waffle.switch` decorator.

### 4.3 Toggles Extraction

To support the analysis of our first research question, we extracted the toggle components across the toggled projects identified in Section 4.1. We used our classification of toggle components from Section 4.2 to serve as an indicator of the existence of toggles. In this sense, wherever Declaration, Router or Point are identified while exercising our static analysis, we can tag that portion of code as part of a feature toggle.

The toggle components of the selected projects were obtained using *extractor*,<sup>3</sup> a command-line tool we built to extract the toggle components across the history of a git repository, pair the versions of the toggle components, and report the resulting differences in a JSON format, as shown in Fig. 2. From a detail perspective, *extractor* collects the toggle components of a given file using library-dependant processes called *parsers*, via standard I/O. For this reason, we created a Python parser and integrated it into *extractor* to obtain the toggle components provided by Waffle (Django Waffle — django-waffle 0.14.0 documentation 2018). We made the Python parser capable of identifying Routers and Points in both regular Python files and Django templates using the Django Template Language for Django 2.1 (The Django template language — Django documentation — Django 2019). In contrast, we decided not to support Declarations, because Declarations are usually stored in a separate database system and we could not obtain access to this data from some

<sup>3</sup><https://gitlab.com/juan.hoyosr/extractor>

```
if waffle.flag_is_active(request, 'course_view'):
    return link_reverse_func('edx.course_experience.course_home')
else:
    return link_reverse_func('courseware')
```

**Listing 3** A Point with two branches of executions. Adapted from *edx-platform* (edx-platform 2011)

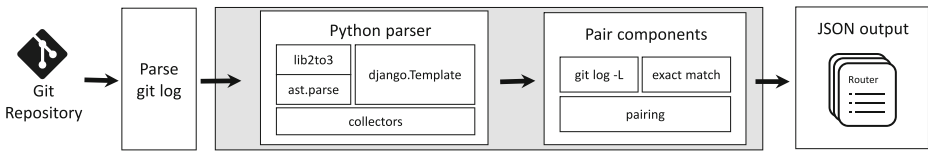
organizations that use the selected Waffle projects. Our Python parser initially analyzes Python 2.7 code to Python 3.6.2 using the standard *lib2to3* module, and later processes the code with the built-in *ast* module. However, if we identify the file is an Django template, we only parse the contents of the file using the module *django.Template*. In any case, both resulting abstract syntax trees are walked to collect the toggle components according to their own specified syntax. Additionally, *extractor* is capable to pair the toggle component versions of two commits by exercising different lookup strategies. First, it uses the `-L <start>, <end>:<file>` mode of *git-log* to trace the originating line numbers of a component at commit  $t_n$  and choose the component with matching line numbers at commits  $t_{n-1}, t_{n-2}, \dots, t_0$ . Alternatively, and only in case *git-log* is unable to trace the origins of a component due to complex modifications in the files, a second strategy is used to choose a component at commit  $t_{n-1}$  if certain attributes, like the filename and the line numbers, match between the two versions of toggle components.

We executed *extractor* and obtained the toggle components of 72 Waffle projects out of 132. We manually verified a random sample of 25 projects that we could not find any toggle component, and determined three different reasons: First, the Waffle library is never used, but referenced as a dependency. Second, the practitioners use the `Flag` or `Switch` Waffle models directly. According to our manual inspection, the Waffle models usage is not a common practice and is solely used to verify the value of a feature toggle. In contrast, the Waffle documentation suggests to use the `Flag` model to create custom functionality when the default model is not sufficient (Django Waffle — django-waffle 0.14.0 documentation 2018). Lastly, a project uses a modified Waffle utility from a dependant project. This is the case of *api-integration*, that depends on custom Waffle extensions defined in *edx-platform*. Interestingly, 40 out of 60 projects without toggle components distribute across 3 organizations: *ccnmtl* (28), *thraxil* (7) and *edx* (5).

We continued our process towards the elaboration of a curated list with the remaining 72 Waffle projects for which we found toggle components. To focus our analysis in the most valuable projects, the number of feature toggles was considered necessary. We decided to use an approximation to count the number of feature toggles in a project based on the toggle name used in the Routers. The reason behind this approximation is that Routers carry most of the identification information of a feature toggle when compared to Points, and have more power to identify other missing toggle components. However, there are edge cases with a dynamic toggle name and it is not possible to

```
@waffle_switch('accessibility')
def accessibility(request):
    return render('accessibility.html', {
        'language_code': request.LANGUAGE_CODE
    })
```

**Listing 4** A decorator of the view *accessibility*. If the toggle is active the method will be executed, if not, a 404 response will be returned. Adapted from *edx-platform* (edx-platform 2011)



**Fig. 2** Internals of *extractor*, a tool to extract toggle components. *extractor* collects the toggle components of a git repository using parsers. Later, at every commit, the toggle components are paired across the history using tracing and matching strategies. The results are given in a JSON format

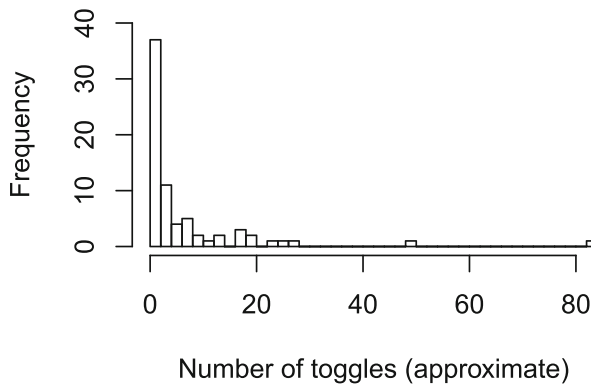
extract a string value with a simple static analysis. For example, the usage of two components `Router1 := flag_is_active(feature.EZID.SWITCH)` and `Router2 := flag_is_active(EZID.SWITCH)` take the attribute of an object and a variable to reference the real toggle name, respectively. In our studied projects, we verified that 96% of the names, attributes and variables are non-generic and meaningful. Accordingly, `Router1` and `Router2` reference the same feature toggle.

Figure 3 shows that distribution of number of toggles in the 72 Python projects. The figure shows that most of the projects (83%) use less than 10 feature toggles. Thus, we decided to analyze projects with no less than 10 Waffle toggles. We choose these projects because they substantially use more toggles than the rest of the projects that use Waffle as their toggling library.

Finally, we ended up with a curated list of 12 projects that substantially use feature toggles. Table 3 shows the names, number of lines of Python code, number of commits, and the number of features toggles in the selected projects.

### 4.4 Developer Survey

To answer our second research question, we surveyed 61 practitioners to better understand why they introduce toggles, and why and when they remove feature toggles in their projects. With this purpose, we designed an online survey with two sections. First, we ask questions about the participant’s background. Second, we inquire the practitioners about their experience with the usage of feature toggles in their projects, from introduction to removal, specifically, we asked the following questions:



**Fig. 3** Right skewed distribution of the approximate number of toggles for 72 projects using Waffle

**Table 3** 12 Python projects using feature toggles with Waffle

ID	Projects	NLOC	Commits	Toggles
1	zamboni	102,582	25,542	84
2	ecommerce	63,118	3,201	49
3	course-discovery	61,693	2,469	27
4	osis	109,950	23,863	26
5	edx-platform	406,602	52,538	23
6	edx-analytics-dashboard	14,528	1,734	20
7	kitsune	67,201	7,357	20
8	osf.io	217,599	53,321	18
9	Jiller	6,350	994	17
10	wardencllyffe	12,017	3,991	13
11	tndata_backend	55,123	3,368	13
12	bedrock	85,726	12,410	12
–	Mean	100,207	15,899	27
–	Median	65,160	5,674	20

Q1. How would you best describe your profession?

A question with the following multiple choices and the last choice is a free-text form: Release Engineer, Software Architect, Chief Technical Officer (CTO), Software Engineer or Developer, Quality Assurance (QA) Engineer, and Other.

Q2. How many years of professional or development experience do you have?

A selection question with the following options: <1 year, 1-3, 4-5, more than 5 years.

Q3. When/for what reason do you introduce toggles in your project?

A question with the following multiple choices and the last choice is a free-text form: Trunk-based development/WIP, Canary Release, Dark Launches, A/B or Multivariate Testing, Blue-green Deployments, Kill Switch, and Other.

Q4. Do you ever remove features toggles? If so, why/when do you remove these feature toggles?

We ask this open-ended question to give respondents maximum flexibility to express their opinion and experience with the usage of toggles.

**Participant Recruitment** To identify the sample of participants in our survey, we collected a list of emails and names from two different sources: direct contacts of the authors, references from our related literature in Section 3, and contributors to our list of 53 toggling libraries from Section 4.1. To do so, we extracted the contributors' names and emails for each project in our dataset. This specific set of practitioners can be considered to be the developers who have experience with using feature toggles. Later, we sent email invitations of our survey to 942 unique participants.

However, since some of the emails are returned for several reasons (e.g., the email server name does not exist, etc.), we successfully reached 877 possible practitioners. After 10 days we stopped receiving answers and we ended up with 61 responses. These entries translate into a 6.96% response rate, which is acceptable when it is compared to the response rate reported in other software engineering surveys (Smith et al. 2013).

Table 4 shows the development experience of the participants, their positions and the source where we collected their email from. Of the 61 participants in our survey, 47

**Table 4** Background of survey participants

Experience	# (%)	Position	# (%)	Source	# (%)
1 - 3	8 (13.1)	Developer	45 (73.8)	Library maintainers	24 (39.3)
4 - 5	6 (9.8)	Architect	7 (11.4)	Anonymous	22 (36.1)
>5	47 (77.1)	CTO	5 (8.2)	Direct contacts	8 (13.1)
		Other	4 (6.6)	Referrals	7 (11.5)

participants have more than 5 years of development experience, 6 responses have between 4 to 5 years, and 8 participants claim to have between 1 to 3 years of experience. As for the participants position, 45 participants identify themselves as software engineers or developers and 7 participants as software architects. Interestingly, 5 participants identify themselves as chief technology officers (CTO). The remaining four participants identify as other positions not listed in the question including, designer, IT consultant, and build configuration. In the final column of the table, we provide the source of the participants.

24 respondents are maintainers of a toggling library, 22 did not provide any email and 15 correspond to either direct contacts of the authors or referrals.

## 5 Preliminaries: What are Feature Toggles Used for?

Before delving into our study about the removal of toggles, we want to better understand what the practitioners use feature toggles for. We do so since prior work only briefly examined the use of feature toggles or provided anecdotal evidence (Neely and Stolt 2013; Rahman et al. 2016; Schermann et al. 2016). Thus, we answer this question in order to better understand the usage of feature toggles from the practitioners perspective.

We ask practitioners in our survey “What they use toggles for?” and we provide them with a list of usage patterns of feature toggles as detailed in Section 2.2. Then, we analyze the responded usage pattern across the demographics of the participants. In addition, to uncover potential relationships between the reported usage patterns and the demographics of our surveyed practitioners, we also aggregate the usage patterns across the position and years of experience of the practitioners.

Our surveyed practitioners reported more than six usage patterns of features toggles. Table 5 shows the feature toggles usage patterns. For each usage pattern, we provide the

**Table 5** Usage patterns according to the years of experience and the position of the practitioner

Usages	# (%)	Developer			Architect	CTO	Other
		1 to 3	4 to 5	>5	>5	>5	>5
Trunk-based	50 (82.0)	7	4	25	6	4	4
Dark	38 (62.3)	3	3	18	7	5	2
Kill	38 (62.3)	4	3	18	6	5	2
Canary	32 (52.5)	4	4	15	3	4	2
A/B	27 (44.3)	4	2	16	2	3	0
Blue-green	7 (11.5)	0	1	4	1	1	0
Other	6 (9.8)	2	0	3	1	0	0

frequency, the development experience and the role of the participants who reported these usage patterns. From Table 5, we observe that the most used patterns of feature toggles are trunk-based development/WIP (82%), dark launches (62%), and kill switch (62%), regardless of the experience or position of the practitioner. In contrast, only 7 practitioners (12%) indicated they use feature toggles to exercise blue-green deployments. Interestingly, 6 practitioners (9.8%) reported other usages of feature toggles that do not belong to the aforementioned patterns. For example, some practitioners reported that they use feature toggle as a “*configuration management in preprod environments*”, as a “*way of toggling messages to users or temporarily switching off stuff*”, and as “*a[sic] la carte pricing of some features*”. Due to the fact that some practitioners provided more than one usage pattern, the percentages add up to more than 100%.

These preliminary results can be compared in various ways with other studies. Schermann et al. (2016) mentioned Canary Releases and A/B Testing usages account for 57% and 50% respectively, and in our case, we see a similar usage trend with nearly 53% and 44% respectively. Furthermore, Rahman et al. (2016) mentioned feature toggles can enable fast context switches when comparing trunk-based development to feature branches. They associate this practice in 97% of toggle changes happening in the development phases of Google Chrome. In our study, we observe a similarity with the usage of this practice, because it is the most indicated pattern according to 82% of our surveyed practitioners.

*Most practitioners use feature toggles to exercise trunk-based development in combination with dark launches and kill switches.*

## 6 Results

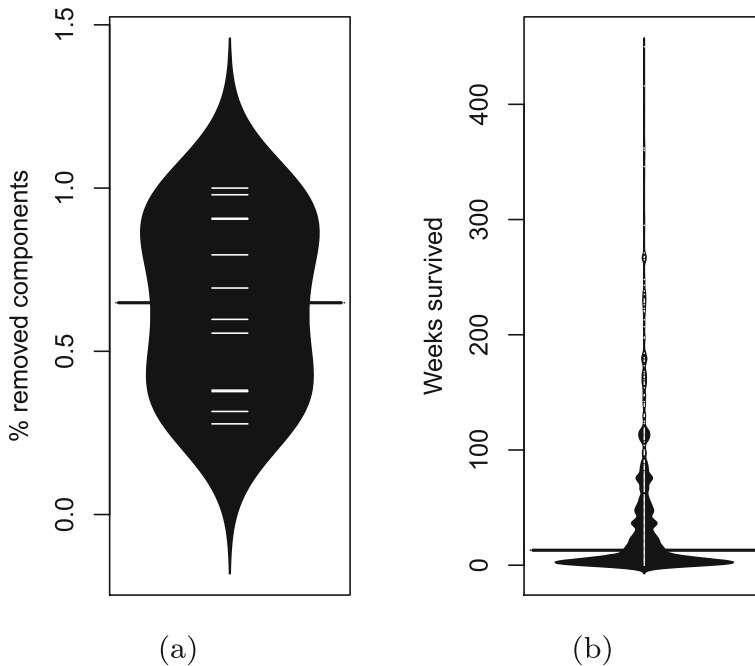
In this section, we answer our main research questions about the removal of feature toggles. Specifically, we take a two-pronged approach where we first quantitatively examine the removal of feature toggles. Then, we complement these findings with qualitative findings from our developer survey. For each research question, we discuss the motivation behind the question, then, we detail the approach to obtain an answer, and finally, we present and discuss our findings.

### RQ1. How long do toggles remain in a project before they are removed?

**Motivation** Our goal is to examine how long feature toggles remain in a software project. Specifically, since prior work advocates to keep a manageable number of feature toggles and that long-term toggles challenge this practice, we want to quantitatively verify the longevity of the toggles in reality. Answering this question helps us provide empirical evidence on how long a feature toggle lives in a software project.

**Approach** To answer this question, we perform two complementary quantitative analyses. First, we examine the amount of removed feature toggles. Second, we run a survival analysis to determine how likely the feature toggles would remain in our studied projects after a given time.

To measure the amount of removed feature toggles in our study, we use the introduction time of the extracted toggle components as described in Section 4.3 and check whether these components are removed or not. Then, we measure the percentage of toggle components that



**Fig. 4** Distributions of removed toggle components and the number of weeks toggle components survive

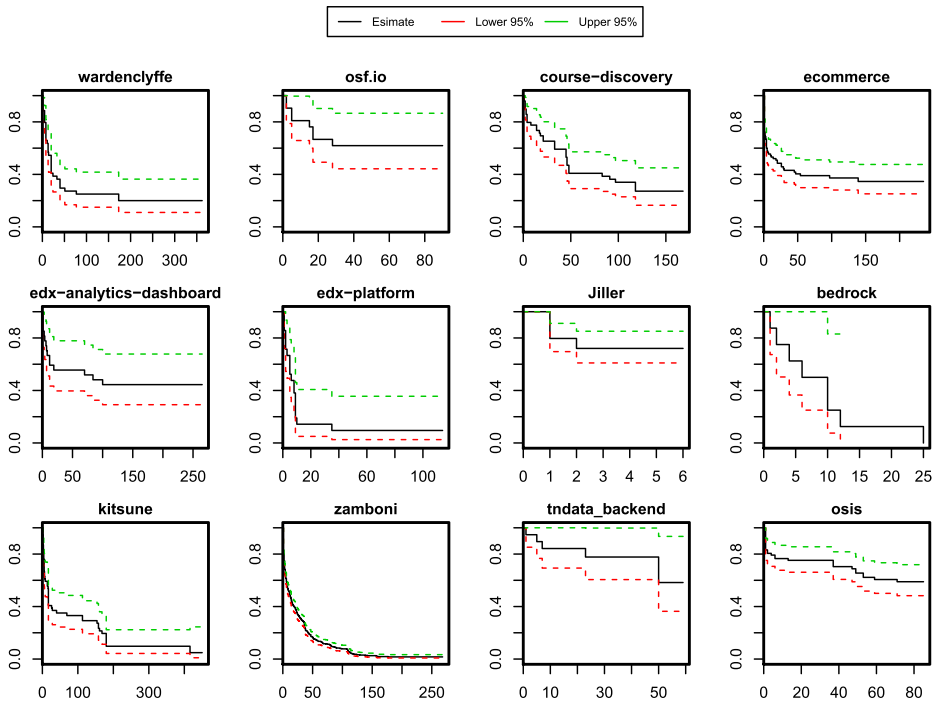
are removed over all the toggle components ever used in the projects. We use the percentage of removed feature toggles instead of the raw number since these projects vary in the number of used feature toggles.

In the second part of answering this research question, we run a survival analysis of feature toggles for each project to discover the proportion of surviving feature toggles after a given time. Each survival analysis is resolved using the Kaplan-Meier estimator, a non-parametric statistic used to estimate the survival probability of subjects across time (Kaplan and Meier 1958). We decide to choose the Kaplan-Meier estimator because it can take into account the toggle components not yet removed at the last commit of each studied project (i.e. right-censored data).

**Findings** Figure 4 shows two distributions: the proportion of removed toggle components and the number of weeks the toggle components survive for all 12 Python projects using Waffle. From Fig. 4a, we observe that the projects remove an average of 65% toggle components. However, we find no strong trend in the amount of removed toggle components as the measurements spread along the axis. 50% of projects remove between 38% and 90% of toggle components. We detail some interesting cases on the remaining 50% of projects. On one side, `tndata_backend` (13 toggles) removed 25% of Routers. On the other hand, the team behind `bedrock` (12 toggles) removed 100% of its toggle components. We confirmed with Mozilla developers<sup>4</sup> that they have not created feature toggles recently and the Waffle database tables have been hard to remove, thus, the Waffle library remains as a dependency. In Fig. 4b we see that most of the toggle components are removed before week 49,

<sup>4</sup>Through our personal correspondence with Mozilla Engineers.





**Fig. 5** Survival analysis of feature toggles in 12 Python projects. The y-axis indicates the percentage of toggle components that are likely to survive after a number of weeks

with a median of 13 weeks. On the contrary, `edx-analytics-dashboard` (20 toggles) and `osis` (26 toggles) keep more than 50% of their toggle components beyond 49 weeks after introducing them in the code, with 15 and 40 remaining toggle Points respectively. According to the chief architect of Open edX,<sup>5</sup> a feature toggle that lives beyond 49 weeks is acceptable when it aligns with an “*Opt-out*” (2 years), “*Open edX option*” (3 years), “*Ops - Graceful Degradation*” (5 years) or “*VIP/White Label*” (5 years) use cases, as detailed in their best practices proposal, OEP-17 (Asthagiri 2018). On the contrary, 49 weeks for a feature toggle to live in `osis` is beyond than expected, based on the information provided by a Software Architect of `osis`.<sup>6</sup> The team behind `osis` uses feature toggles to hide partially implemented features and as kill switches to deactivate a feature that could cause data damage in a production environment. This same developer mentioned their feature toggles should be “short term”, to align with their methodology to deliver value every two weeks.

In Fig. 5 we provide the survival curves of the toggle components for the 12 Projects using Waffle. Here we see that `osf.io` (18 toggles) is likely to keep 62% of their toggle components after week 28. However, one of their software engineers<sup>7</sup> acknowledges that they usually keep feature toggles in the code longer than “a couple of months”, due to a trade-off between the “team’s external development commitments”, the “current throughput” and the effort that involves the removal of the feature toggle. In short, practitioners in

<sup>5</sup>Contacted via the slack channel of OpenedX.

<sup>6</sup>Contacted via personal correspondence.

<sup>7</sup>Via personal correspondence with a Software Engineer of the Center for Open Science.

osf.io remove toggles when they “have time to remove” them. Conversely, our results in Fig. 5 show that `edx-platform` will likely keep 14% of their feature toggles after week 10 and 1% of their toggle components after week 35, which indicates they invest efforts to remove feature toggles used as *Incremental Release*, *Launch Date* and *Ops - Monitored Rollout*, before the 3-months expected lifetime. Similar efforts can be evidenced in the survival curves of `zamboni` (84 toggles), as they are likely to remove 36% of feature toggles before their 30-days canary release usages.<sup>8</sup> However, `zamboni` and `kitsune` have the two-highest survival median times of living feature toggles, with 245 and 346 weeks respectively.

To determine which kind of toggles remained in the source code and assess potential implications, we manually collected evidence on the usage patterns of 103 toggles with a lifespan greater than 49 weeks. To obtain the evidence of usage patterns, the first two authors manually looked up in the source code, commits, pull requests, and issue tracking systems of each feature toggle. The first two authors determined the expected lifetime of the toggles (short/long) independently. We measured our agreement with a weighted Cohen’s Kappa coefficient, a well-known statistical method scaled between -1.0 and 1.0, where a negative value indicates a poorer than chance agreement, zero indicates a chance agreement, and a positive value indicates a better than chance agreement. In our case, the level of agreement was 0.88, which is considered as an excellent agreement. In cases where there was a difference, the two annotators discussed each case to reach a consensus using the concrete evidence.

Based on our manual analysis to determine the expected lifetime of the toggles that stay for more than 49 weeks in the source code, we found that 64% of the feature toggles are meant to be long-term, and 19% short-term; for 17% of them we could not find any clear usage pattern (e.g. kill switch, A/B test, etc.) evidence. In an example of a long-term toggle, `allow_uploads` (360 weeks) from `wardenclyffe`, the developer indicates in the commit message that the toggle is introduced to control the “maintenance mode” in the web application, which is a feature expected to remain available in case the application needs to enter in this state. Comparatively, the case of `optional_location_fields` (117 weeks) in `ecommerce` project, which is a short-term toggle, used as “a fallback mechanism for the backend to not require some fields while running” an A/B test in case “something bad happens”. In total, 7 projects accrue for more than 2,900 weeks of short-term toggles that live after 49 weeks; `kitsune` (3 toggles) and `ecommerce` (6 toggles) lead with 934 and 747 weeks respectively; on the contrary, `tndata_backend` (2 toggles) and `course-discovery` (1 toggle) accrue for 120 and 114 weeks respectively.

To conclude, projects remove feature toggles in different proportions and moments after introducing them. However, 25% of the feature toggles remain in the source code after 49 weeks and, for some projects, this lifetime is significantly longer than expected.

*75 % of the analyzed feature toggles are removed within 49 weeks after their introduction in the studied Python projects. These findings show that practitioners should pay careful attention to the maintenance of feature toggles since they may live in the project for a long time.*

## RQ2. Why and when practitioners remove toggles?

<sup>8</sup>Via our personal correspondence with the project’s Engineering Manager at Mozilla.

**Motivation** Although our quantitative results provide us with hard numbers of how much feature toggles are removed and how long they stay in a Waffle project, we still do not understand why practitioners perform these removals and when. In this sense, we would like to know whether there are certain activities or times in a project where practitioners remove feature toggles, to later compare and discuss.

**Approach** We survey practitioners with different backgrounds to understand why and when they remove feature toggles. To analyze the free-text answers for these two questions, we perform an iterative coding process to categorize the participants' responses (Corbin et al. 2008). We first print all of the responses and manually analyze them. The first two authors carefully read the participant's answers and come up with a number of categories that the responses fell under. Next, the same two authors go through the responses and classify them according to the extracted categories. To confirm that the two authors correctly classified the responses to the right category, we measure the classification agreement between the two authors. To do so, we use the well-known Cohen's Kappa coefficient (Cohen 1960). Cohen's Kappa coefficient is a well-known statistical method that is used to evaluate the inter-rater agreement level for categorical scales. The resulting coefficient is scaled to range between -1.0 and 1.0, where a negative value means poorer than chance agreement, zero indicates exactly chance agreement, and a positive value indicates better than chance agreement. We found that the level of agreement was 0.81 which is considered to be an excellent agreement. At the end, for the few cases that annotators failed to agree on, a third author was consulted to resolve the differences and categorize these cases.

**Findings** Our results show that nearly 82% of the participants responded 'YES', meaning they remove features toggles from their projects, while almost 5% responded 'NO'. We also found that 8 participants (13.11%) indicated that they rarely removed the toggles or it depends on the type of toggles. For example, the participants that use feature toggles as kill switches expressed that they do not tend to remove feature toggles as part of their usual removal cycle.

As for the reasons *why practitioners remove feature toggles*, we were able to extract three main categories across the 58 participants that conditionally or unconditionally remove feature toggles (95.08%). We describe the categories and provide some examples below:

1. **Follow release process (21.32%)**. The most cited reasons for removing features toggles reported by the participants of our survey is the fact that they follow a release process that enforces the removal of feature toggles after a certain time. For instance, the participant P6 explains her reasons as follows: "*It is built into our development cycle. Each toggle has an expiration date in which it needs to be merged in the code.*"
2. **Feature no longer supported (8.2%)**. Practitioners also reported that they remove features toggles once they decided not to release the feature that was toggled. For instance, P12 stated "*Yes, ASAP after the feature is stable and has been turned on for everybody. Or after the feature is rejected. Usually within 1-2 months.*"
3. **Reduce technical debt (8.2%)**. Interestingly, more than 8% of the participants in our survey see that unused feature toggles can lead to some technical debt in their projects. Thus, practitioners reported that one of the main reasons of removing feature toggles is to eliminate the technical debt. For instance, P21 said "*Yes. We view toggles as technical debt, and try to remove them as soon as possible, e.g when we are confident a feature works as intended in production*".

About 13% of our surveyed participants stated that they remove feature toggles depending on the situation. Particularly, 4.92% of the participants indicated that feature toggles related to *Kill switches* remain in the code for a long time. For instance, P37 reported that *“The only feature flags that persist in the codebase long-term are kill switches.”*

We also ask the practitioners *when they decide to remove feature toggles* from their projects. We were able to classify the responses of 58 practitioners that remove feature toggles into four main categories. In the following paragraphs, we describe these categories and provide concrete examples from the practitioners responses.

1. **The Feature is stable and in production (77.05 %)**. The majority of the practitioners in our survey reported that they remove feature toggles when the toggled features are stable and become part of the projects. As an example, the practitioner P24 said *“[W]e aim to remove toggles some time after successful activation in all context[s] and environments.”*
2. **Regular/scheduled audit (16.39%)**. A less common category reported by practitioners is that they remove features toggles when practising cleanup audits. For example, the participant P44 stated that: *“Yes, after regular audits of toggles in code”*. Similarly, the maintenance campaign of Chrome (Rahman et al. 2016) can be located in this group.
3. **A/B test is done (6.56%)**. Also, practitioners reported that the removal of feature toggles is related to test strategies they follow. They stated that they remove toggles when their A/B Test is passed. One practitioner explain this as follows; P27 *“Yes. When done with A/B testing for instance. Canary release toggles and Kill switches have a tendency to stay for a while, until someone decides to clean it up.”*
4. **During refactoring (1.64%)**. A less frequent category reported by practitioners is the fact that they remove feature toggles when they perform refactoring activities. For example, P5 *“Rarely, mostly during refactoring to reduce tech debt/feature no longer supported”*.

Finally, almost 9% of the practitioners stated that they remove toggles in different occasions but we could not group them into a specific category since they are a small portion. For example, the participant P53 said *“Yes, not however in a consistent fashion. I just tend to notice after a while that a given toggle is only ever in one state (because the need for it to be in multiple states has passed) and I then remove it while making other changes to the code.”*

*Most practitioners remove feature toggles as part of the life cycle of a feature, in audits, or when refactoring. Still, 5 % keep kill switches for long periods or do not remove feature toggles at all.*

## 7 Discussion

Given that this study is one of the first to focus on the removal of feature toggles, we present a number of possible implications of our findings. Specifically, we focus on the possible implications for software engineering researchers and practitioners.

**Implications for Researchers** Our study provides empirical evidence of special consideration to researchers interested in exploring solutions to the issues experienced by practitioners

that leverage feature toggles. Based on our results, we find that, first, researchers should avoid generalizations regarding the number of removed feature toggles or the time practitioners remove feature toggles across multiple projects. We have not only found that these measurements significantly differ across Waffle projects, but that they also differ among projects of the same organization and even when compared to projects of previous studies. Hence, the verdict is still out on when, how and if feature toggles should be removed (or managed). We believe that our study is a step in the right direction, but more empirical studies are needed.

Second, decomposing feature toggles into less complex elements can provide interesting perspectives of analysis. For example, we observed that Routers and Points are removed differently in most of the projects we studied. Also, their discrepancy is useful to identify feature toggle elements by future research.

**Implications for Practitioners** Our study demonstrates that feature toggles remain useful for software developers, architects, engineering managers and chief technology officers of any-sized projects. The results here presented amplify both the values and the struggles that experienced practitioners of feature toggles have kindly shared with us. We are not aware of any silver bullet regarding the management of feature toggles, but we have reported empirical evidence on multiple ways other projects keep their inventory of feature toggles. Special consideration should be taken to communicate and integrate the management of feature toggles into the workflow and tooling of the project.

Additionally, the accumulation of feature toggles in the code is not necessarily a bad thing when done in a methodical manner. The decision to accrue “toggle debt” should be adverted and balanced between the business and the available engineering capacity. Hence, if your project has feature toggles that live for a long time, that might not necessarily be a bad thing. As opposed to long-term toggles, the accumulation of short-term toggles after long periods could have potential negative implications, such as catastrophic misconfigurations (Securities and E. Commission 2013) or hard to maintain code bases (Neely and Stolt 2013; Rahman et al. 2016).

Further, as with any software system, practitioners should expect change in the life-time of their feature toggles. While the usage patterns of feature toggles can help planning for their management, practitioners should be aware that feature toggles can easily transition from one usage to another. Also, to reduce the risk of an unwanted behaviour of a software system, avoid the reuse of feature toggles in your system and make the necessary adjustments to remove all toggle components from your system when not needed anymore. By understanding the elements of a toggling subsystem practitioners can identify scattered Declarations, Routers or Points. The more scattered, the more places in the source code practitioners would need to affect to cleanup a feature toggle. Declarations stored in external databases, not in code, should be of special interest. Finally, maintainers can take advantage of the components to adjust their toggling libraries to help practitioners track the discrepancies between the removal of Declarations, Routers and Points, and potentially minimize lingering toggle components or dead code.

## 8 Threats to Validity

In this section, we discuss the threats to the validity of our study. We discuss internal validity, construct validity, and external validity (Yin 2009).

## 8.1 Construct Validity

Construct validity considers the relationship between theory and observation, in case the measured variables do not measure the actual factors. First, to understand the structure of used toggle components in a project, we manually examine two projects. This process is maybe influenced by human judgment. However, the first author, who lead the analysis, has several years of industrial experience and that condition give us confidence in the process. Second, we selected three sources to obtain toggling libraries. This selection could also be affected by human judgment factors. However, we did multiple explorations in academic databases and in other web-sites related with the technique, to reduce the chances of missing important libraries. Third, the augmented list of toggled projects in our study setup could show less commits than real. GitHub API returns empty results when the commit of the contributors do not match registered users. To allow these projects to make it into our final list, we do not impose high restrictions in the number of commits.

## 8.2 Internal Validity

Internal validity concerns factors that could affect our analysis and findings. First, our method to identify projects that depend on toggling libraries is prone to recall issues from the GitHub API limitations. It is known that GitHub limits the number of returned items (1,000) even if paging is used (GitHub 2011a), and to respond with incomplete results when their search engine is not fast enough to deliver all the known items (GitHub 2011b); we reduce the potential recall issues in two ways: one, by creating new searches of the same term from a bisect-by-file-size procedure whenever a response indicates this behaviour, and two, by searching GitHub projects over another data source like Libraries.io.

Another possible source of validity concerns is our *extractor* tool, whenever it performs incorrectly or when biased towards any result. *extractor* does not support identifying Declarations in Waffle, or direct usages of Waffle models to perform trivial feature toggle checks and deviate from the official library documentation (Django Waffle — django-waffle 0.14.0 documentation 2018). Also, the files encoded in a character set other than *utf-8* are ignored, even if they could be completely valid and contain toggle components. Additionally, we use relaxed settings and automatic fixes in the standard library *lib2to3* to create AST representations from Python 2.7 code files, and increase the amount of useful files. Another important threat are the potential miss of template files containing toggle components. To mitigate this issue we created to the best of our knowledge a custom template loader on top of the default Django loader to bypass special template libraries and tags.

Moreover, we explain in Section 4.3 that *extractor* heavily trusts *git-log* when pairing versions of a toggle component. Given *git-log* is not 100% accurate, the same toggle component could be reported to be removed and then added, again, like if a different one. To measure the presence of this issue in the results of our *extractor* tool, we randomly sampled 5% of the 322 extracted toggles and validated their associated toggle components. We found that *extractor* reported a toggle component as removed and then added for a second time at the same commit, in 6% of the cases after name normalization. In addition, to mitigate other potential *extractor* deficiencies we created and exercised automated tests for the different components of the tool, and we took advantage from every run against our studied projects to improve our tool to extract toggles. Lastly, our survey and its analysis are prone to human factors. However, we were careful to design the survey questions with most of the researchers involved, we selected our practitioners from trusted sources, and we followed an iterative coding process for our free-text analysis.

To study the reason why and when practitioners remove feature toggles, we surveyed 61 practitioners and none of them worked with any of the authors of this study at the moment the survey was made available. Also, no participant is a contributor to any of the 12 Python projects in Table 3. While there is potential that our survey participants may not have professional experience in using feature toggles, none of surveyed developers mentioned that she/he/they does not have feature toggle experience, which gives us confidence that our survey participants are representative of feature toggles developers.

Our method to identify feature toggles cannot automatically resolve why they were not removed. Hence, the toggles identified as not removed in our dataset, may not necessarily indicate bad practices by the project.

### 8.3 External Validity

This type of validity threat considers the generalization of our findings. First, our background study cannot guarantee all existent toggle components were identified. This process was a manual process and has a human factor of subjectivity involved. However, we explored multiple projects and revisited when necessary to resolve conflicting patterns. To uncover the removal trends of feature toggles, we examined twelve open-source Python projects that contain at least ten Waffle toggles, hence, our findings may not generalize to other Python projects, especially those that are not open-source. That said, our work is an empirical data point in the toggle removal area. We plan to (and encourage other researchers to) build on this knowledge in the future. To facilitate such efforts, we make our dataset publicly available. To understand why and when practitioners remove feature toggles, we surveyed practitioners and received 61 responses. Although we believe this is a sufficient number of responses, our results may not hold for all practitioners, including the practitioners of our 12 Python projects.

Our 53 toggling libraries are a small sample of libraries to enable feature toggle capabilities and they could be distant from all the existent toggling libraries out there. These characteristics avoid the results of our study to be generalized to different subjects. To reduce the risk of generalizing our results, multiple authors analyzed the setup, the execution and our findings.

## 9 Conclusion

Feature toggles is a technique that allows practitioners to hide features from execution at will. Several industrial companies use it in their development processes to build high quality software, while others have reported negative experiences related to the survival time of feature toggles in the source code. However, very little is known from the literature about how feature toggles are removed in software projects. In this paper, we examined the removal of feature toggles with both qualitative and quantitative methods.

We found evidence of long-purposed kill switches when analyzed 12 open source projects written in Python. Interestingly, 75% of their feature toggle components were removed from the source code within 49 weeks after introduction. We identified remaining old feature toggles not related to critical features that could degrade the behaviour of the application. In our opinion, this could indicate that new usage patterns are emerging or that the project has inappropriate feature toggles maintenance. Later, we conducted a survey with 61 practitioners to understand the conditions to remove feature toggles from their projects. Most of the practitioners remove feature toggles either as part of the life cycle of

a feature, in the case of a regular audit, or when refactoring the source code. However, 5% of the surveyed practitioners reported not to remove kill switches as frequent as other usage patterns or at all. The results of our study provide empirical evidence that despite the surveyed practitioners expressed they retire feature toggles from their projects, a significant amount of feature toggles in our quantitative experiment remain in the code for long periods of time carrying potential negative risks.

We believe the software community can benefit from a better understanding of the feature toggles landscape, as it is one of the drivers towards rapid value delivery. In the future, we aim to investigate the possible problems associated to feature toggles. Specifically, the impact in the quality of the software when long-term feature toggles are present and the technical debt caused by feature toggles as mentioned by more than 8% of our surveyed participants, are captivating open fields. Also, while this study focuses on features toggles that meant to be removed, it is also important to examine the long lived features toggles. Additionally, the study on the relationship of configuration engineering and feature toggles could lead to unify perspectives and best practices. Also, the projects that reference toggling libraries as dependencies, but do not present any feature toggles are interesting study subjects. We encourage the researchers to improve in our methods and tools; towards even more insightful perspectives of powerful techniques for the daily-software developer.

**Acknowledgments** We would like to thank the practitioners who devoted their time and effort to respond to our online survey, and to all the practitioners of Mozilla, Université catholique de Louvain, Open edX, Tennessee Data Commons and the Center for Open Science, that responded back to our call and contributed their valuable experience using feature toggles in their projects.


## References

- Center for Open Science (2013) <https://github.com/CenterForOpenScience/osf.io>. Accessed 2019-01-09
- Django Packages : Feature Flipping (2018) <https://djangopackages.org/grids/g/feature-flip/>. Accessed 2018-11-05
- Django Waffle — django-waffle 0.14.0 documentation (2018) <https://waffle.readthedocs.io/en/stable/>. Accessed 2019-01-11
- edx-platform (2011) <https://github.com/edx/edx-platform>. Accessed 2019-01-09
- Libraries.io - The Open Source Discovery Service (2015) <https://libraries.io/>. Accessed 2019-01-09
- The Django template language — Django documentation — Django (2019) <https://docs.djangoproject.com/en/2.1/ref/templates/language/>. Accessed 2019-06-16
- The Web framework for perfectionists with deadlines — Django (2019) <https://www.djangoproject.com/>. Accessed 2019-03-05
- PEP 318 – Decorators for Functions and Methods — Python.org (2003) <https://www.python.org/dev/peps/pep-0318/>. Accessed 2019-01-23
- Adams B, Bellomo S, Bird C, Marshall-Keim T, Khomh F, Moir K (2015) The practice and future of release engineering: a roundtable with three release engineers. *IEEE Softw* 32(2):42–49
- Adams B, McIntosh S (2016) Modern release engineering in a nutshell – why researchers should care. In: Leaders of tomorrow: future of software engineering, proceedings of the 23rd IEEE international conference on software analysis, evolution, and reengineering (SANER), Osaka, Japan, pp 78–90
- Asthagiri N (2018) OEP-17: Feature toggles. <https://open-edx-proposals.readthedocs.io/en/latest/oep-0017-bp-feature-toggles.html>. Accessed 2020-01-10
- Bosworth A (2012) Building and testing at Facebook. <https://www.facebook.com/notes/facebook-engineering/building-and-testing-at-facebook/10151004157328920>. Accessed 2019-03-25
- Claps GG, Berntsson Svensson R, Aurum AA (2015) On the journey to continuous deployment: Technical and social challenges along the way. *Inf Softw Technol* 57(1):21–31
- Cohen J (1960) A coefficient of agreement for nominal scales. *Educ Psychol Meas* 20(1):37–46
- Corbin J, Strauss A et al (2008) Basics of qualitative research: techniques and procedures for developing grounded theory



- Feitelson DG, Frachtenberg E, Beck KL (2013) Development and deployment at facebook. *IEEE Internet Comput* 17(4):8–17
- GitHub (2011) GitHub API v3 — GitHub Developer Guide. <https://developer.github.com/v3/>. Accessed 2019-01-09
- GitHub (2011) Search — GitHub Developer Guide. <https://developer.github.com/v3/search>. Accessed 2019-01-03
- GitHub (2013) Searching code - User Documentation. <https://help.github.com/articles/searching-code>. Accessed 2019-01-03
- Harry B (2012) Announcing Continuous Deployment to Azure with Team Foundation Service — Brian Harrys blog. <https://bit.ly/2MvKEkT>. Accessed 2019-01-22
- Hodgson P (2016) Feature Toggles. <https://martinfowler.com/articles/feature-toggles.html>. Accessed 2017-08-30
- Humble J, Farley D (2010) Continuous delivery: reliable software releases through build, test and deployment
- Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining github. In: Proceedings of the 11th working conference on mining software repositories, MSR 2014, ACM, pp 92–101
- Kaplan EL, Meier P (1958) Nonparametric estimation from incomplete observations. *J Am Stat Assoc* 53(282):457–481
- Kästner C (2019) Feature flags vs configuration options — same difference? <https://www.cs.cmu.edu/ckaestne/featureflags/>. (Accessed on 03/30/2020)
- LaunchDarkly (2015) Feature flags - feature flags, toggles, controls. <http://featureflags.io/feature-flags/>. Accessed 2018-11-03
- Mäntylä MV, Adams B, Khomh F, Engström E, Petersen K (2015) On rapid releases and software testing: a case study and a semi-systematic literature review. *Emp Softw Eng* 20(5):1384–1425
- Neely S, Stolt S (2013) Continuous delivery? Easy! Just change everything (well, maybe it is not that easy). In: Proceeding - AGILE 2013, pp 121–128
- Osherove R (2016) Feature toggles – enterprise devOps. <http://enterprisedevops.org/feature-toggle-frameworks-list/>. Accessed 2018-12-11
- Rahman AAU, Helms E, Williams L, Parnin C (2015) Synthesizing continuous deployment practices used in software development. 2015 Agile Conference, 1–10
- Rahman MT, Querel L-P, Rigby PC, Adams B (2016) Feature toggles: practitioner practices and a case study. In: Proceedings of the 13th international conference on mining software repositories, MSR '16. ACM, New York, pp 201–211
- Rahman MT, Rigby PC, Shihab E (2018) The modular and feature toggle architectures of Google Chrome. *Emp Softw Eng*, 1–28
- Ray B, Posnett D, Filkov V, Devanbu P (2014) A large scale study of programming languages and code quality in github. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering - FSE 2014. ACM Press, New York, pp 155–165
- Sayagh M, Kerzazi N, Adams B, Petrillo F (2018) Software configuration engineering in practice: interviews, survey, and systematic literature review. *IEEE Trans Softw Eng*
- Schermann G, Cito J, Leitner P, Zdun U, Gall H (2016) An empirical study on principles and practices of continuous delivery and deployment. *Peer J Preprints* 4:e1889v1
- Securities and E. Commission (2013) Administrative and cease-and-desist proceedings against knight capital americas LLC. <https://www.sec.gov/litigation/admin/2013/34-70694.pdf>. Accessed 2019-01-19
- Smith E, Loftin R, Murphy-Hill E, Bird C, Zimmermann T (2013) Improving developer participation rates in surveys. In: 2013 6Th international workshop on cooperative and human aspects of software engineering (CHASE). IEEE, pp 89–92
- Yin RK (2009) Case study research: design and methods (applied social research methods). London and Singapore: Sage
- Zapata D (2014) Going from 3 week to daily releases at netflix. USENIX Association, Philadelphia

## Affiliations

Juan Hoyos<sup>1</sup>  · Rabe Abdalkareem<sup>2,3</sup> · Suhaib Mujahid<sup>2</sup> · Emad Shihab<sup>2</sup> · Albeiro Espinosa Bedoya<sup>1</sup>

Rabe Abdalkareem  
abdrabe@gmail.com

Suhaib Mujahid  
s\_mujahi@encs.concordia.ca

Emad Shihab  
eshihab@encs.concordia.ca

Albeiro Espinosa Bedoya  
aespinos@unal.edu.co

<sup>1</sup> Universidad Nacional de Colombia, Medellín, Colombia

<sup>2</sup> Concordia University, Montreal QC, Canada

<sup>3</sup> Queen's University, Kingston ON, Canada