Piranha: Reducing Feature Flag Debt at Uber

Murali Krishna Ramanathan
Uber Technologies, Inc.
New York City, NY, USA
murali@uber.com

Lazaro Clapp, Rajkishore Barik
Uber Technologies, Inc.
San Francisco, CA, USA
lazaro,rajbarik@uber.com

Manu Sridharan
University of California, Riverside
Riverside, CA, USA
manu@cs.ucr.edu

ABSTRACT

Feature flags are commonly used in mobile app development and can introduce technical debt related to deleting their usage from the codebase. This can adversely affect the overall reliability of the apps and increase their maintenance complexity. Reducing this debt without imposing additional overheads on the developers necessitates the design of novel tools and automated workflows.

In this paper, we describe the design and implementation of Piranha, an automated code refactoring tool which is used to automatically generate differential revisions (*a.k.a* diffs) to delete code corresponding to stale feature flags. Piranha takes as input the name of the flag, expected treatment behavior, and the name of the flag's author. It analyzes the ASTs of the program to generate appropriate refactorings which are packaged into a diff. The diff is assigned to the author of the flag for further processing, who can land it after performing any additional refactorings.

We have implemented Piranha to delete code in Objective-C, Java, and Swift programs, and deployed it to handle stale flags in multiple Uber apps. We present our experiences with the deployment of Piranha from Dec 2017 to May 2019, including the following highlights: (a) generated code cleanup diffs for 1381 flags (17% of total flags), (b) 65% of the diffs landed without any changes, (c) over 85% of the generated diffs compile and pass tests successfully, (d) around 80% of the diffs affect more than one file, (e) developers process more than 88% of the generated diffs, (f) 75% of the generated diffs are processed within a week, and (g) Piranha diffs have been interacted with by ~200 developers across Uber.

Piranha is available as open source at https://github.com/uber/piranha.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

ACM Reference Format:
Murali Krishna Ramanathan, Lazaro Clapp, Rajkishore Barik, and Manu Sridharan. 2020. Piranha: Reducing Feature Flag Debt at Uber. In *Software Engineering in Practice (ICSE-SEIP '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3377813.3381350

ICSE-SEIP '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7123-0/20/05.
https://doi.org/10.1145/3377813.3381350

1 INTRODUCTION

In traditional software development workflows [1], shipping new features involves determining the set of features to be released in a version, and implementing these features as part of a new branch of the source code. This can introduce unnecessary complexities [60] in terms of building and testing different versions of the software application. Further, dependencies among features across versions can hamper the agility of the development process [7, 36].

Modern software development practices avoid this complexity by implementing all features of an application as part of a monolithic source code repository [11, 22, 28, 30, 33, 37, 45, 53], and subsequently configuring the set of features that are visible to the user. This configuration is achieved by guarding each new feature under a feature flag.[1] The configuration pertaining to the flags that are enabled is dynamically supplied to the user as a network payload during application startup. Experiences in the recent decade [11, 31, 51, 52] have shown that there are many other advantages of feature flags beyond easing the software development process. For example, app developers can easily control the configuration of features and deliver a customized user experience to each individual user or user group. Further, testing new features in the field before a global roll-out becomes a trivial task [52].

While there are many benefits of employing feature flags, there are also challenges that need to be addressed. Introduction of flags increase the number of control paths in the application, affecting test coverage [15]. Further, unintended flag dependencies due to the structure of the code can affect the application's reliability. For instance, a high frequency trading firm lost 465M USD in 30 minutes when obsolete code was made live due to flipping the behavior of a feature flag [41]. Also, flag usage can impose technical debt on the organization when the flags have served their purpose and need to be removed [52, 57]. In this paper, we focus on the problem of managing the technical debt associated with *stale* flags.

We consider a flag to be stale when the purpose that necessitated its introduction is resolved. For example, when a flag that is used to control the geographical locations where it is rolled out is rolled out globally, the flag becomes stale. Therefore, the code in the true branch of the flag can execute unconditionally, and the code in the false branch can be eliminated from the source. Further, all unreachable code artifacts due to this refactoring should also be removed. Obsolete tests related to the flag should also be deleted.

Surprisingly, refactoring code and eliminating technical debt due to stale flags can be a non-trivial task. Our experience has shown that this is due to a confluence of diverse problems:

- sub-optimal flag management that makes information on staleness of flag to be ambiguous,
- churn in the organization affecting flag ownership, and

[1] Also, referred to as gatekeepers [32], toggles [31, 52].

- lack of developer incentives to cleanup source code related to stale flags.

Consequently, the source code becomes bloated with dead code paths affecting overall ease of development and software reliability.

In this paper, we share our experiences with overcoming the challenges related to stale flags and detail the practices that we have evolved to improve the overall code hygiene. We have also designed and implemented a code refactoring tool, named Piranha, to automatically cleanup code related to stale feature flags. Apart from building processes to enable timely removal of stale code using Piranha, we have also designed coding guidelines to reduce additional manual effort on generated diffs.

We applied Piranha on multiple mobile applications at Uber that are used by millions of users daily. These applications are built for Android and iOS platforms, and are implemented using Java, Swift, or Objective-C. Our investigation of the usage of Piranha from Dec 2017 to May 2019 shows that it was used by around 200 developers to delete 1381 flags, which corresponds to deletion of 71KLoC spread across the codebases of multiple apps, where 65% of the generated cleanup diffs are landed to the master branch without any manual changes. We also observe that developers act on 88% of the generated diffs either by landing them or purposefully abandoning the diff, and perform this activity within a week for 75% of the diffs. For 80% of the flags, we observe that more than one file is affected.

Our experiences with deploying Piranha revealed interesting challenges ranging from simple behavioral issues corresponding to handling auto-generated source code refactorings, to deeper technical challenges associated with constraining the coding patterns related to feature flags. Based on our findings, we present detailed recommendations on managing code associated with feature flags. We believe our experiences can be leveraged by other industry practitioners to apply similar strategies within their organizations, and can enable academicians to design novel program analysis strategies to overcome the problems discussed in this paper.

1.1 Technical contributions

We make the following technical contributions in this paper:

(1) We present the challenges due to stale feature flags and describe the design and implementation of a source-code refactoring tool, named Piranha, to reduce the technical debt efficiently.
(2) We propose workflows surrounding Piranha to minimize the overhead for cleaning stale code artifacts.
(3) We apply Piranha on mobile applications built for Android and iOS platforms, using Java, Swift and Objective-C, and present our findings on the usage of Piranha over 18 months across multiple dimensions.
(4) We provide recommendations for increasing the underlying automation.

2 BACKGROUND AND MOTIVATION

In this section, we provide the background on feature flags and motivate the problem due to stale feature flags. Initially, we discuss the terminology used in the paper.

- *diff*: Short form of differential revision [26, 43], which contains the code modifications and metadata pertaining to the modifications including its author, code reviewer(s), summary of the changes, test plan, etc. This is similar to a pull request [50].
- *landing a diff*: The process of merging the code modifications in the diff into the master branch of the repository.
- *task*: A work item assigned to a developer that summarizes the work that needs to be accomplished (e.g., building a new feature, fixing a bug, etc).
- *feature flag*: Variable in source code that enables or disables a specific feature of the application.
- *treatment behavior*: Defines the functionality corresponding to the treatment group [21] for a given feature.
- *control behavior*: Defines the functionality corresponding to the control group [21] for a given feature.

Feature flags are classified into various categories depending on their application. The most common flags are:

- *optimistic feature flags*: provide flexibility in rolling back a feature in the presence of unforeseen errors,
- *gradual roll-out flags*: help constrain the features to a limited audience before general availability,
- *A/B testing flags*: used for experimenting and obtaining user feedback, and
- *parameterized flags*: introduce complex dynamic settings for customized user experience.

In order to introduce a flag in the codebase, a developer creates an entry in the flag management system, and inputs attributes related to the name of the flag, type of the flag, roll-out percentages, targeted platforms, geographical locations where the flag is operational, etc. Simultaneously, the flag is defined in the source code which helps build a connection between the backend system and the mobile app instance. Subsequently, this flag can be used in the code as any other variable to manage the app behavior.

```
public enum RidesExpName implements ExpName {
   RIDES_NEW_FEATURE,
   ...
}

if (cachedExperiments.isTreated(RIDES_NEW_FEATURE)) {
     // implementation for treatment behavior
} else {
     // implementation for control behavior
}

@Test
@RidesExpTest(treated=RidesExpName.RIDES_NEW_FEATURE)
public void test_new_feature() {
   ...
}
```

Listing 1: Illustrative example demonstrating usage of feature flags in source code

Listing 1 presents a simple example illustrating the usage of feature flags in the source code.[2] Initially, a new flag, named `RIDES_NEW_FEATURE`, is defined as part of a list of flags in

[2]Flag usages are more complex and do not always conform to this style.

`RidesExpName`. This flag is also registered in the flag management system. Subsequently, the flag is used in the code using a feature flag API, `isTreated`, and the implementations of treatment/control behaviors are provided under this conditional guard. Finally, to test the code with various flag values, for each unit test, an annotation is added to specify the value of the feature flag. In the example code shown above, we observe that the test, `test_new_feature` runs where the flag under consideration is in a `treated` state.

Feature flag APIs are used for interacting with flags in the codebase. There are three kinds of flag APIs:

(1) boolean APIs that return a boolean value and are used to determine the control path taken by the execution (e.g., `isTreated` shown in Listing 1),
(2) update APIs, which update the feature flag value in the running system, and
(3) parameter APIs that return a non-boolean primitive value (integer, double, etc) which corresponds to an experimental value being controlled from the backend.

In our experience, boolean APIs are the most popular, while update APIs are typically used for testing. Parameter APIs are infrequent and correspond to a very small set of flags.

When the goal of a feature flag is accomplished (e.g., the experiment related to A/B testing flag is finished), the flag needs to be disabled in the feature flag management system and all code artifacts related to the flag need to be removed from the source code. This ensures improved code hygiene and avoids technical debt [52, 57]. In practice, this simple post cleanup process is not always performed. Oftentimes, developers do not cleanup code related to obsolete flags, causing accumulation of technical debt.

Based on our experience, the flags that introduce technical debt for the development teams can be classified as follows.

- *Stale flags:* Flags which are no longer used to test multiple behaviors. In other words, the execution path corresponding to the flag is the same across all app instances.
- *Orphaned flags:* Flags whose owners have left the organization and the final status of the flag roll out is unclear.
- *Forgotten flags:* Flags for which the associated code execution exhibits different behaviors across app instances, even though developers are no longer interested in the differing behaviors.

The presence of code related to these unnecessary flags can affect software development across multiple dimensions. First, the overall reliability of the application can be adversely affected in the presence of unnecessary control flow paths. For example, a simple bit corruption of the payload from the backend system to the application can alter the overall behavior of the application. Second, effort must be spent to maintain test coverage of these unnecessary paths. Third, the presence of dead code and tests impacts the overall build and testing time, affecting developer productivity. Finally, coding complexity can become quite high as the programmer has to reason about the control paths related to these obsolete flags.

We motivate this further by explaining a major outage due to this technical debt. A financial market maker experienced a major outage and significant financial loss due to the poor use of feature flags and not cleaning up the related dead code [41, 49]. The system mistakenly routed thousands of orders per second into the NYSE market. Within 45 minutes, more than 4M unwanted trades corresponding to 397M shares were executed and the firm lost more than 460M USD. This happened due to a confluence of multiple events – the presence of dead code in critical code regions, re-purposing a feature flag previously used for testing (confusing the DevOps), and significant code refactorings without effective regression testing. The ability to delete dead code due to flags could have potentially reduced the magnitude of the outage.

In this paper, we present the strategies employed to reduce technical debt from stale flags and discuss our experiences implementing and deploying an automated code cleanup tool, named Piranha.

3 CHALLENGES

We discuss the challenges with respect to designing an automated approach for cleaning up stale flags.

- **Staleness of flag:** Determining whether a flag is stale or not is surprisingly non-trivial. Firstly, the flag should have been rolled out 100% either as treatment or as control. Even when it is rolled out, the developer may still not be ready to eliminate the flag. For example, certain flags are used as kill switches whose values can be modified for emergency purposes, flags that are used for monitoring debug information, etc. Therefore, even when flags are completely rolled out, they may not necessarily be stale.
- **Flag ownership:** Since flags were not cleaned up for a long period of time, determining ownership information for stale flags became problematic. A few engineers who worked on the flags either had moved to other teams or had left the organization.
- **Developer inertia:** Even when the flag is definitively stale, the flag owner may not necessarily cleanup the code because building newer features is usually better recognized and rewarded [5] than reducing technical debt.
- **Coding style:** The lack of any restrictions pertaining to code related to feature flags increases the complexity underlying the design of an automated tool. For example, helper functions for flag related code cannot be easily differentiated from any other function in the code. Also, the complexities introduced by tests where developers introduce manual state changes to the flag can restrict the tool in performing a comprehensive cleanup of tests. For instance, when flag related code is being unit tested, sometimes it is unclear whether the test can be discarded in its entirety because the functionality is removed, or specific state changes within the body of the test needs to be removed so that the remaining functionality can continue to be tested.

4 DESIGN

4.1 Functionality

Piranha takes as input: the stale flag under consideration, the treatment behavior [52], and the owner of the flag. It analyzes the code for uses of this flag in pre-defined feature flag APIs and refactors it to delete code paths based on the treatment behavior.

The obtained patch is packaged into a diff (see Section 2), containing the modified code, author of the modification, reviewer for the code, etc, all packaged to facilitate code reviewing and landing the modifications to the source master. In this case, the diff generated by Piranha has the owner of the flag as the reviewer, who can review the diff and land the changes if there are no further

modifications. Otherwise, as the owner of the flag, they can perform further manual modifications and update the diff accordingly.

To simplify project management related to processing Piranha generated diffs, Piranha creates a task [47] to cleanup each stale flag. The task is assigned to the owner of the flag and the generated diff is linked to the task. In this manner, the stale flag cleanup is seamlessly embedded into the developer's workflow as this task is handled similar to any other new feature or bug fixing task.

4.2 Design trade-offs

We now discuss the design of the refactoring component of Piranha. There are three key dimensions for performing the cleanup.

(1) Delete code that *immediately* surrounds the feature flag APIs
(2) Delete code that becomes unreachable due to performing the previous step. We refer to this as *deep cleaning*.
(3) Delete tests related to feature flags.

In order to perform cleanup across all three dimensions, it is necessary to perform reachability analysis [54] to identify code regions that become unreachable, implement algorithms to identify tests related to testing feature flags, etc. While this would ideally ensure complete automation where a developer simply needs to review the deletions and land the changes in master, it requires overcoming two challenges – (a) ensuring that the underlying analysis to perform cleanup is sound and complete [46], and (b) the engineering effort required to implement and scale such analysis to processing millions of lines of code in a useful timeframe.

To the best of our knowledge, we are unaware of any analyses that can ensure complete automation as determining reachability in a sound and complete manner is infeasible [42]. This necessitates some form of manual intervention by the developer after the cleanup is performed by the tool. Therefore, instead of building out a complex analysis where the amount of developer intervention is unknown and the return on engineering investment is unclear, we choose a practical approach of designing the technique iteratively, based on the coding patterns observed in the codebase.

4.3 Overview of refactoring

In this section, we describe the patterns handled by Piranha and an overview of the rewrites that are performed. Our refactoring technique parses the abstract syntax trees (ASTs) [4] of the input source code to detect and rewrite the expressions and statements.

Boolean feature flag APIs are defined to identify the treatment or control behavior pertaining to a given flag. For those APIs, we define rules for rewriting them. For example, given an API `isTreated(Experiment flag)`, we define the rewriting as follows:

```
// flag = piranha.flag, piranha.treatment = true
isTreated(flag) → true

// flag = piranha.flag, piranha.control = true
isTreated(flag) → false

// flag ≠ piranha.flag
isTreated(flag) → isTreated(flag)
```

For update APIs, we simply delete the corresponding statement. We also discard the use of parameter APIs which can result in compilation failures with the generated diff.

We then simplify boolean expressions involving the flag API checks previously rewritten into constants, by performing partial evaluation [38]. Finally, we rewrite code statements as follows:

```
// cond = true
if (cond) b1 else b2 → b1

// cond = false
if (cond) b1 else b2 → b2

//flag = piranha.flag
enum Expt {a1, flag, a2} → enum Expt {a1, a2}

// flag = piranha.flag, piranha.control = true
@ExpTest(flag = control) testX(. . . ) { . . . }
     → testX(. . . ) { . . . }
// flag = piranha.flag, piranha.treated = true
@ExpTest(flag = control) testX(. . . ) { . . . } → _

// cond = true | false
x.f = cond → _

// cond = true | false
get(. . . ) { return cond } → _
```

In general, for any statement that is rewritten due to the presence of a boolean constant, our analysis expects that the boolean constant is a result of a simplification due to our analysis.

The `if` conditions are rewritten based on the evaluation pertaining to the conditional. The `enum` statements handle the deletion of flag declarations. The `ExpTest` annotations handle the experimental annotations code pertaining to tests. The assignment rewriting handles the case where the results of flag APIs are stored and used. The method definition rewriting handles the scenario pertaining to the wrapper method for the flag API.

The last two rewriting rules implement the first part of our *deep cleaning* step (see Section 4.2). They remove assignments and function definitions when the expression on the right operand (or the function's body) evaluates to a boolean constant after stale flag replacement and simplification. Piranha then performs a follow up pass to replace references to these fields and calls to these methods with the derived boolean constant, and apply rewrite rules to the statements around them, which are analogous to the ones for flag check APIs. In theory, this process needs to be repeated to a fix point [44]. We found $k = 2$ iterations sufficient in practice for our codebases, and a larger k can be substituted if needed.

4.4 Piranha pipeline

While Piranha as a standalone tool can be used to generate diffs and tasks, a key problem with flag cleanup is the lack of prioritization which can result in Piranha not being used effectively. Therefore, we built a workflow pipeline that periodically (weekly) generates diffs and tasks to cleanup stale feature flags. The Piranha pipeline queries the flag management system for a list of stale flags, and invokes Piranha, providing as input the name of the stale flag, its owner, and the intended output behavior (treatment or control).

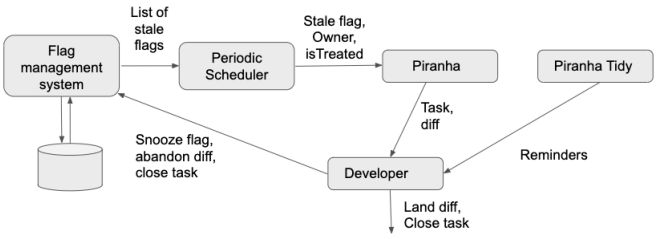

Figure 1: Architecture of Piranha pipeline

Figure 1 presents the architectural diagram of the pipeline. Generated tasks and diffs for cleaning up stale feature flags are *pushed* to the developer, who can either approve and land the changes, or abandon the diff when the flag is not yet ready for cleanup. Since developers may not tend to act on these diffs in a timely manner, we also introduced a reminder bot, named PiranhaTidy, to periodically add reminders on open Piranha tasks.

5 FINDINGS

In this section, we address the following key research questions pertaining to Piranha and flag cleanup:

(1) Do developers process generated diffs?
(2) How quickly does a developer process a generated diff?
(3) How often does a developer need to make manual changes before landing a diff?
(4) How many lines of code did Piranha automatically remove from the codebase? How many more lines did Piranha encourage the developers to remove manually?
(5) What is the breadth of changes while performing cleanup due to stale flags?
(6) Is maintaining a Piranha pipeline sustainable?
(7) Do regular reminders to process diffs help with the cleanup?

Tooling details We have implemented Piranha to refactor `Objective-C`, `Swift`, and `Java` programs.

PiranhaJava is used to refactor stale feature flag related code in `Java` applications, specifically those targeting the Android platform. It is implemented in `Java` on top of Errorprone [3] as an Errorprone plugin [3, 9]. We handle the basic cleanup, without deep cleaning of methods and fields. We perform precise cleanup of test code in the presence of flag related test annotations.

PiranhaSwift is implemented in `Swift` using SwiftSyntax [56] for refactoring code related to stale feature flags in `Swift` programs. Apart from the basic cleanup, we implement deep cleaning by using $k = 2$. Test annotations are not handled since feature flag related annotations are unused in testing in our `Swift` apps.

PiranhaObjC is used to cleanup code related to stale feature flags defined in `Objective-C` code. It is implemented in `C++` as a Clang Plugin and uses AST matchers and rewriters internally to parse and rewrite the ASTs. This variant implements the core cleanup logic, it does not perform deep cleaning or cleanup of testing code.

Deployment and monitoring A developer can use Piranha in the following ways – (a) invoke Piranha from command line or web interface to generate diffs/tasks, or (b) process the diffs/tasks generated by Piranha pipelines. We report the findings for both forms of invocation of Piranha.

We discuss our findings based on the usage of Piranha from Dec 2017 to May 2019 which has helped in the cleanup of 1381 flags across multiple codebases (eats, rider, driver, etc), which are implemented in Objective-C, Swift and Java.

Approximately 95% of the diffs that formed part of our study are generated by the Piranha pipelines, which are run automatically once a week. The owner of the flag is marked as the reviewer for the generated diff. Since reviewing diffs is part of usual developer activity, the generated diffs are processed by the developers accordingly. The remaining 5% diffs are generated by manual invocation of Piranha by a developer. The diffs generated by Piranha are tagged with a custom project tag to enable us to differentiate them from other diffs, thereby, enabling us to gather relevant statistics.

Piranha pipelines use a heuristic to consider flags that are unmodified in the flag management system for more than a specific period (e.g., 8 weeks) as stale and generate diffs for those flags. The exact time period for staleness of a flag is configured by the individual teams that process the diffs generated using Piranha. We observe that the current time taken to generate a diff using Piranha is less than 3 minutes.

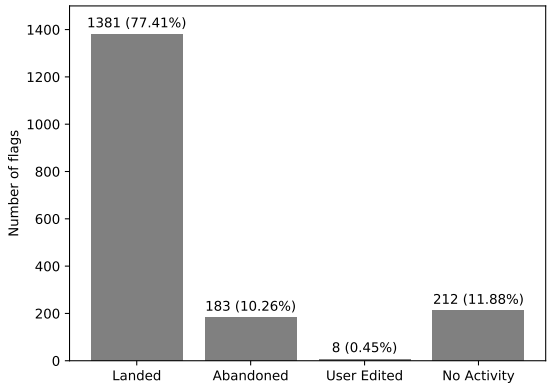

Figure 2: Total flags in each status (snapshot)

Activity on generated diffs Figure 2 presents the activity data on Piranha generated diffs. Out of all the diffs generated, we observe that there is developer activity on 88% of the diffs. This includes diffs that are landed by the developers (with any additional changes), and diffs that are abandoned because the flag is not ready to be removed from the codebase as it is not yet stale.

Figure 3 classifies the activity data across the three languages. We observe that diffs generated by PiranhaObjC have significantly higher land rates, followed by PiranhaSwift. Interestingly, over a quarter of PiranhaJava generated diffs go unprocessed by developers. This is due to two reasons – (a) more manual effort required to process diffs generated by PiranhaJava over the other two variants, and (b) strict size requirements for over-the-air updates [10] on iOS apps forcing developers to optimize for size.

Statistics on code deletion Figure 4 presents the statistics for various attributes related to flag cleanup – flags cleaned up,

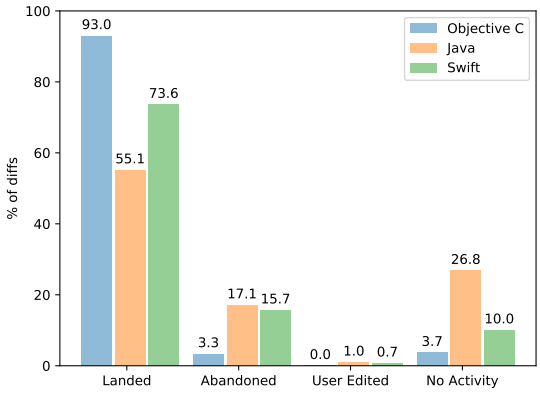

Figure 3: Percentage of flags in each status (snapshot)

	Objective C	Java	Swift	Total
Flags removed	782	284	315	1381
Remaining Flags	1267	2496	2838	6601
Flags Redux.	38.2%	10.2%	10.0%	17.3%
Auto (Del. lines)	11416	4853	3970	20239
Manual (Del. lines)	1869	30337	18551	50757
Percent Auto	85.93%	13.79%	17.63%	28.51%
Percent Manual	14.07%	86.21%	82.37%	71.49%
Auto (Avg)	14.6	17.1	12.6	14.7
Manual (Avg)	2.4	106.8	58.9	36.8
Auto (Median)	11	9	6	10
Manual (Median)	0	12	5	0
Codebase LoC	713K	4.0M	2.97M	7.7M
Codebase Redux.	1.86%	0.88%	0.76%	0.92%

Figure 4: Lines removed automatically and manually

total/mean/median lines deleted, total lines of code across all the codebases, etc. We observe a reduction of 17.3% of the flags across all the codebases. The remaining 6601 flags are not all stale as many of them are still under active use. Overall, we also observe a reduction of the codebase size by 1% approximately.

Automation ability To study the manual effort involved after a diff is created by Piranha, we compute the number of lines removed by Piranha automatically and the subsequent manual effort. Figure 4 presents this data. We observe that the additional manual effort required on PiranhaObjC generated diffs is minimal. Interestingly, for `Java` and `Swift`, Piranha encourages developers to delete a significant portion (> 80%) of the lines manually. We discuss strategies to reduce this manual interference in Section 6.

On average, we observe that Piranha deletes approximately 15 lines of code per diff. We also observe that the count of manually deleted lines per diff is greater than 100 lines for `Java`. Since these numbers were due to a few diffs involving large cleanups, we also compute the median count of deleted lines. We observe that at least 10 lines are removed by Piranha for more than 50% of the generated diffs. The number of lines that needs to be manually removed is at least 12 for 50% of PiranhaJava generated diffs, as compared to 0 and 5 for PiranhaObjC and PiranhaSwift generated diffs respectively. This additional effort involved on a diff can be a barrier in the effective usage of PiranhaJava, which also explains the poor activity data for PiranhaJava in Figure 3.

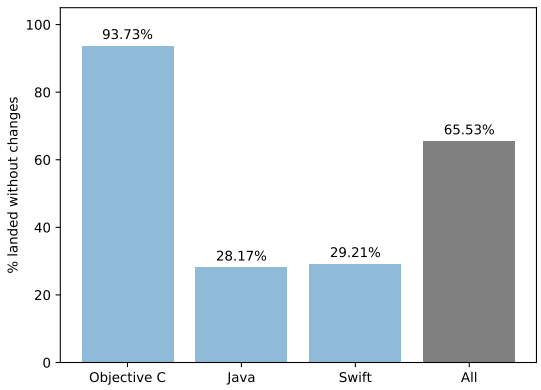

Figure 5: Percentage of flags landed w/o user changes

Figure 5 presents the distribution of flags requiring no manual changes, where 93.73% of the flags cleaned up by PiranhaObjC do not require any manual changes. Consequently, PiranhaObjC is used more frequently (e.g., 782 flags are cleaned up) as compared to the other two variants of Piranha. Not surprisingly, more diffs generated by PiranhaObjC are processed (see Figure 3) because of increased developer confidence.

Due to the technical debt accumulated over the years and the number of stale flags, there is developer overhead in processing the generated diffs for each flag. We can potentially reduce this overhead by cleaning up multiple flags per diff. The effectiveness of PiranhaObjC also lead developers to adopt this process and to clean up multiple flags in batches. Figure 6 shows the distribution of number of lines deleted (including any manual deletions) for 491 flags that are cleaned up in batches. It also demonstrates the efficacy of PiranhaObjC in creating large diffs (> 500 lines deleted) that can be landed without any further changes.

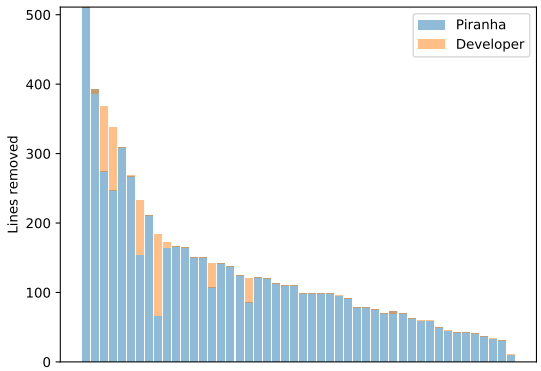

Figure 6: Code deleted in bulk Objective C diffs

Reasons behind different adoption rates PiranhaObjC performs better than the other two variants of Piranha for the

following reasons. Firstly, the coding style related to feature flags is amenable for automated cleanup and follows the coding guidelines discussed in Section 6. Secondly, the data pertaining to staleness of a flag is precise resulting in more diffs being landed. Thirdly, the presence of developers who champion the usage of PiranhaObjC and help improve the tooling by providing quick feedback on potential issues ensure generation of comprehensive diffs which are processed at a higher rate. Finally, the network effects of adoption results in the provision of better features which benefits the developers further forming a virtuous cycle.

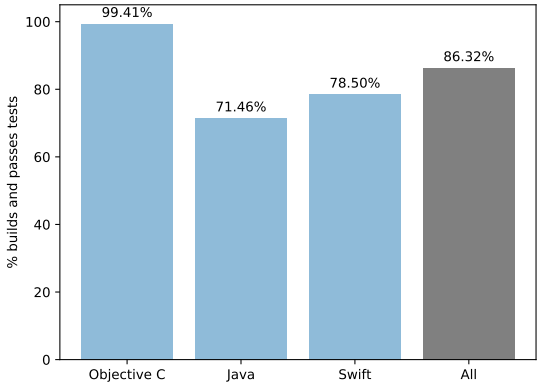

Figure 7: Percentage of generated diffs that pass CI

Accuracy of refactorings We also studied whether the generated diffs resulted in compilation or test failures, which guarantees the need for further manual changes. Figure 7 shows the percentage of generated diffs that successfully compiled and passed tests on our continuous integration (CI) [61] system. Surprisingly, we observe that a significant percentage of diffs generated by PiranhaSwift and PiranhaJava do not have any failures. This is much higher than the percentage of diffs that are landed without any additional changes. Based on a manual perusal of these diffs, the additional changes performed are due to incomplete deletions (e.g., helper functions, redundant tests, etc). Defining a coding style to be used with feature flags while simultaneously improving the analysis underlying Piranha can improve the automated cleanup rates.

Complexity of refactorings Figure 8 shows the distribution of the count of modified files, including developer edits before landing. It shows that 80% of the diffs involve a change to five files or fewer, with 66% involving three or fewer. For such changes where affected code regions are bounded, a more precise goal-directed reachability analysis to delete code may be practical, even if an equivalently precise whole-program analysis would be impractical. For the 4% diffs requiring changes to ten or more files, a common cause is an entire UI workflow gated by the flag becoming unreachable, resulting in the deletion of all supporting classes.

Processing time Figure 9 presents the histogram of time taken to land for diffs generated by Piranha. Recall that in order to land a generated diff, a reviewer needs to review and approve the changes, after making any additional modifications. We observe that

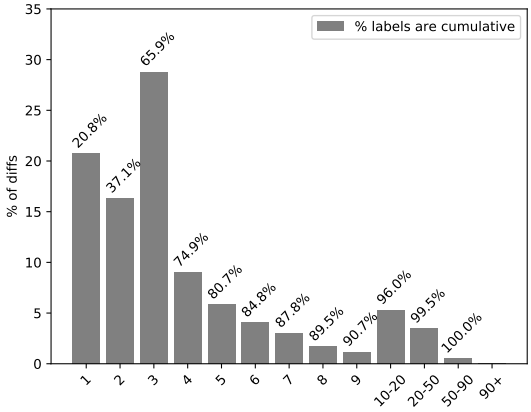

Figure 8: Diffs per number of files touched

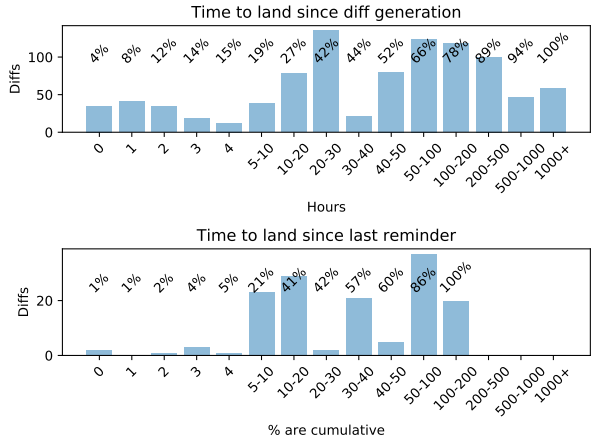

Figure 9: Histogram of time to land for Piranha diffs

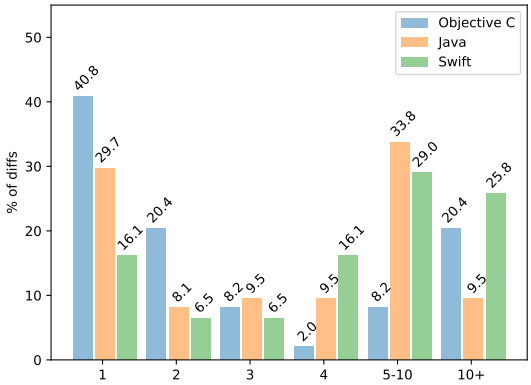

Figure 10: Number of PiranhaTidy reminders per diff+task pair

a significant portion (≥ 66%) of the diffs are processed and landed within a week of their generation (168 hours). Not surprisingly, approximately 40% of the diffs are processed within a day or two of their generation. This is to avoid possible merge conflicts [17] with other changes to the code with the passage of time. The figure (at

the bottom) also shows that adding reminders to diffs usually results in landing 86% of the diffs within five days of the last reminder. The processing of the generated diffs quickly shows that (a) code cleanup related to the stale flags is necessary, and (b) the burden on the developers in terms of the time taken to review and land the generated diffs is acceptable.

Figure 10 shows that a majority of the diffs required multiple reminders before they are processed. Given the high processing rates for diffs, less time taken to land a diff after a reminder, and the high number of reminders, we conclude the following – (a) developers did not clean up the flags earlier because the flags were not stale yet, and (b) reminders enable the developers to process the diffs in a timely manner when the flag becomes stale.

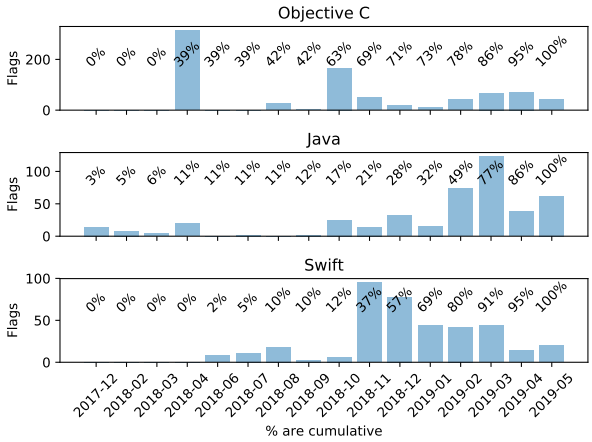

Figure 11: Histogram of flag diff generation

Chronological perspective Figure 11 presents the number of diffs generated by each variant of Piranha since Dec 2017. There are a few interesting takeaways. First, while a large number of flags are cleaned up in Apr 2018 in Objective-C code, the technical debt due to flags built up again in a few months necessitating running PiranhaObjC again in Oct 2018. This also inspired the design and implementation of Piranha pipelines. Second, we observe that there are spikes in cleanup of flags for each of the variants. These correspond to focused efforts (e.g., fixit week [8]) to reduce technical debt by various teams. This also shows the importance of program analysis tools for such focused activities. Finally, we also observe a steady stream of generation of Piranha diffs in the last few months of the study. This, coupled with the processing rates of the diffs (Figure 2), shows that Piranha is useful in practice.

Developer interaction We also note that 197 developers interacted with the diffs generated by Piranha. Figure 12 shows the distribution of developers based on the number of flag processed by them. It shows that there are four developers (2%) who have processed more than 30 flag cleanups, with one of them processing a total of 71 flag diffs. Also, the majority of participating developers (127, 64%) have processed at least two diffs, and a quarter of developers interacting with Piranha have landed 5+ diffs. Piranha pipelines are designed to reach out to such developers. The usage of

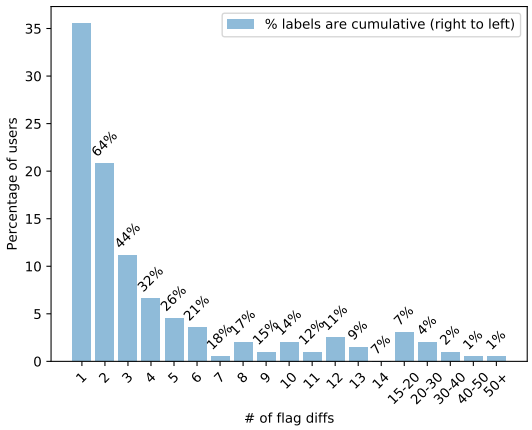

Figure 12: Flag deletion diffs landed per user

Piranha across a wide developer base also demonstrates the ability of novel developer tools that improve productivity to seamlessly become part of routine workflows.

In summary, the above data demonstrates the usefulness of Piranha in reducing feature flag debt at Uber. However, there are many areas where Piranha can potentially be improved such that the processing rates of diffs are closer to 100% and manual changes to diffs are closer to 0%. We discuss the challenges in achieving this goal and our recommendations in the following sections.

6 RECOMMENDATIONS

In this section, based on our experiences with the use of Piranha, we present our recommendations pertaining to feature flag management and the coding guidelines related to use of flags.

Precise maintenance of flag data: When a feature flag is created initially, the owner of the flag should also define an expiry date for the flag. This clear intent enables automated cleanup tools to generate diffs without confusing developers. Further, the owner for each flag should be tracked precisely. For example, reassigning flag ownership when a developer exits a team/organization should become a standard task.

Coding guidelines: We propose the following coding guidelines to eliminate manual intervention completely in terms of performing additional cleanup. We classify the guidelines for application (production) code and testing code separately. For application code, we recommend the following:

(1) Use feature flag APIs that return boolean values only.
(2) Invoke the APIs from within a conditional expression, which can be a compound expression that contains multiple operators (and, or, negation) can ease the cleanup analysis. If that is not possible, avoid store and re-use across multiple classes.
(3) If flag API invocations are stored in a variable or wrapped in a function, use these variables or wrapper functions as part of a conditional expression without performing further re-assignments or wrapping.
(4) Implement the feature related code immediately as part of the conditional statement corresponding to the feature flag API within

the same method. Otherwise, if there are helper method implementations that are specific to the flag treatment (treated/control), then use annotations to denote helper method implementations appropriately. For example, consider the following example:

```
@helper(f1=treated,f2=control,f3=treated)
helperMethodImpl(...) { ... }
```

Here, the `helperMethodImpl` is used when flags `f1` and `f3` are treated and `f2` is in control. When these flags and the specified behaviors are eliminated, the entire `helperMethodImpl` can then be eliminated.

For test code, we recommend the following:

(1) Use names for tests that do not necessarily tie them to treatment or control behavior. Instead, the test names should be associated with the functionality.
(2) Annotate unit tests with flag name and treatment behavior. For example,

```
@flagtest(f1=treated)
unitTestForSomeFeature(...) { ... }

@flagtest(f1=control)
unitTestForAnotherFeature(...) { ... }
```

Here, the two tests are run where the state of feature flag `f1` is treated and control respectively.

Developer workflows: Cleanup of feature flag related code needs to be performed periodically. In general, the Piranha pipeline discussed in Section 4 can form the template for workflows involving code cleanup tools. Further, information pertaining to stale code and corresponding diffs needs to be part of the developer workflow. For example, the information on outstanding stale flag diffs can be included in the IDE [23] to enable developers to process the corresponding diffs immediately.

7 RELATED WORK

Feature-driven software development [11, 37] has become the defacto standard in large software companies [53] such as Google [22, 33], Facebook [28], Netflix [45], and Flickr [30]. Feature flags enable timely releases in continuous integration and delivery systems [27]. A comprehensive description of various categories of feature flags and their integration with different feature-flagged systems is described at [31]. Rahman *et al* [52] report the wide-spread use of feature flags, and discuss the strengths and drawbacks of feature flags in continuous integration and delivery systems. They also highlight the introduction of technical debt and additional maintenance overhead for developers due to feature flags.

Feature flags can also be seen as defining a family of programs from the same codebase, one for each valid set of flag assignments. This is the view taken in the Software Product Line (SPL) literature (see e.g. [16, 24, 48, 58]). Dynamic Software Product Lines (DSPLs), in particular, are SPLs where the instantiation of the configuration happens at runtime (see e.g. [14, 19, 34, 35]). Thus, feature flags are a specific DSPL implementation strategy [39], and removal of stale configurations has been studied at least in the SPL case. Closest to Piranha is the work in [12, 13], which removes code guarded by stale C preprocessor conditionals. Also related are approaches like [40] and [2], which seek to transform features guarded by conditionals into some other form of SPL configuration, such as aspects or extension modules. Apart from proposing techniques to refactor code related to stale flags, this paper also shows that stale flag removal can be integrated with developer's ongoing CI workflows, and adopted enthusiastically at scale to clean up large codebases with MLoC and spanning multiple languages.

This automated generation of diffs, and thus code branches, bears some relation to the extensive literature [15, 18, 20, 55] quantifying, predicting, and helping manage the pain of merge conflicts. Our evaluation of Piranha provides insight into both the incidence of CI issues and the 'shelf-time' (before diffs become too old to merge) for a specific set of automatically generated code changes.

The issue of stale feature flags introducing dead code into a codebase has been studied before [29, 57]. Since this code is dead only based on the value returned by API calls (which might be querying a remote configuration), standard dead-code elimination used in optimizing compilers [6, 25, 59] cannot remove them. On the other hand, Piranha assists the developers in removing this dead code, leveraging information about known stale flags.

8 CONCLUSIONS

In this paper, we take a deep dive into analyzing the prevalence and impact of stale feature flags in the large scale codebases at Uber across multiple languages and platforms. We present a tool, Piranha, to automatically cleanup code related to stale feature flags for Java, Swift and Objective-C applications. Furthermore, the observations and insights around stale feature flags have been distilled into a simple set of guidelines that developers can use to minimize feature flag maintenance overhead.

ACKNOWLEDGMENTS

We thank Vasileios Kalintiris, Nick Lauer, Gautam Korlam, Martin Patfield, Scott Graham, Richard Howell, Edward Jiang, Vikram Bodicherla, Tho Nguyen, AJ Ribeiro, Harman Dhaliwal and Ali-Reza Adl-Tabatabai for their support in operationalizing Piranha at Uber. We also thank the anonymous reviewers for their detailed feedback that helped improve the presentation.

REFERENCES

[1] Pekka Abrahamsson, Juhani Warsta, Mikko T. Siponen, and Jussi Ronkainen. 2003. New Directions on Agile Methods: A Comparative Analysis. In *Proc. of the 25th Intl. Conf. on Software Engineering* (Portland, Oregon) *(ICSE '03)*. 244–254.
[2] Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed E. Hassan. 2009. Can we refactor conditional compilation into aspects?. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development, AOSD 2009, Charlottesville, Virginia, USA, March 2-6, 2009*. 243–254.
[3] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*. 14–23.
[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[5] Eric Allman. 2012. Managing Technical Debt. *Commun. ACM* 55, 5 (May 2012), 50–55.
[6] AndroidDeadCode 2018. Shrink, obfuscate, and optimize your app. https://developer.android.com/studio/build/shrink-code. Accessed: 2019-01-29.

[7] Mikio Aoyama. 1998. Agile Software Process and Its Experience. In *Proceedings of the 20th International Conference on Software Engineering* (Kyoto, Japan) *(ICSE '98)*. IEEE Computer Society, Washington, DC, USA, 3–12.

[8] Nathaniel Ayewah and William Pugh. 2010. The Google FindBugs Fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. ACM, New York, NY, USA, 241–252.

[9] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: practical type-based null safety for Java. In *Proc. of the ACM European Software Engineering Conf. and Symp. on the Foundations of Software Engineering, FSE '19*. 740–750.

[10] Rajkishore Barik, Manu Sridharan, Murali Krishna Ramanathan, and Milind Chabbi. 2019. Optimization of Swift Protocols. In *Object Oriented Programming Systems Languages and Applications* (Athens, Greece) *(OOPSLA '19)*.

[11] Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A Software Architect's Perspective*. Addison-Wesley, New York. http://my.safaribooksonline.com/9780134049847

[12] Ira D. Baxter and Michael Mehlich. 2001. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proc. of the 8th Working Conf. on Reverse Engineering, WCRE '01*. 281–290.

[13] Ira D. Baxter, Christopher W. Pidgeon, and Michael Mehlich. 2004. DMS: Program Transformations for Practical Scalable Software Evolution. In *26th Intl. Conf. on Software Engineering (ICSE 2004)*. 625–634.

[14] Nelly Bencomo, Svein O. Hallsteinsen, and Eduardo Santana de Almeida. 2012. A View of the Dynamic Software Product Line Landscape. *IEEE Computer* 45, 10 (2012), 36–41.

[15] Christian Bird and Thomas Zimmermann. 2012. Assessing the Value of Branches with What-if Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. ACM, New York, NY, USA, Article 45, 11 pages.

[16] Jan Bosch. 2000. *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education.

[17] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, New York, NY, USA, 168–178.

[18] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2013. Early Detection of Collaboration Conflicts and Risks. *IEEE Trans. Softw. Eng.* 39, 10 (Oct. 2013), 1358–1375.

[19] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz Cortés, and Mike Hinchey. 2014. An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. *Journal of Systems and Software* 91 (2014), 3–23.

[20] M. Cataldo and J. D. Herbsleb. 2011. Factors leading to integration failures in global feature-oriented development: an empirical analysis. In *2011 33rd International Conference on Software Engineering (ICSE)*. 161–170.

[21] Nanyu Chen, Min Liu, and Ya Xu. 2019. How A/B Tests Could Go Wrong: Automatic Diagnosis of Invalid Online Experiments. In *Proceedings of the 12th ACM Intl. Conference on Web Search and Data Mining (WSDM '19)*. 501–509.

[22] ChromeRelease 2012. Chrome Release Cycle. https://www.slideshare.net/Jolicloud/chrome-release-cycle. Accessed: 2019-01-29.

[23] Jürgen Cito, Philipp Leitner, Martin Rinard, and Harald C. Gall. 2019. Interactive Production Performance Feedback in the IDE. In *Proc. of the 41st Intl. Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. 971–981.

[24] Paul Clements and Linda Northrop. 2002. *Software product lines*. Addison-Wesley.

[25] DeadCode 1981. A survey of data flow analysis techniques. https://www.clear.rice.edu/comp512/Lectures/Papers/1981-kennedy-survey-scan.pdf. Accessed: 2019-01-29.

[26] Differential Revision 2019. Differential User Guide. https://secure.phabricator.com/book/phabricator/article/differential/. Accessed: 2019-09-20.

[27] Paul Duvall, Stephen M. Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional.

[28] FBRelease 2014. Native mobile app releases (Move fast and ship mobile things). https://www.youtube.com/watch?v=Nffzkkdq7GM. Accessed: 2019-01-29.

[29] FeatureBits 1981. Feature Bits: Enabling Flow Within and Across Teams. https://www.infoq.com/presentations/Feature-Bits/. Accessed: 2019-01-29.

[30] FlickrRelease 2009. 10+ Deploys Der Day: Dev and Ops Cooperation at Flickr. https://www.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr. Accessed: 2019-01-29.

[31] Fowler 2010. Feature Toggles (aka Feature Flags). https://martinfowler.com/articles/feature-toggles.html. Accessed: 2019-01-29.

[32] GateKeeper 2012. Building and testing at Facebook. https://www.facebook.com/notes/facebook-engineering/building-and-testing-at-facebook/10151004157328920/. Accessed: 2019-01-29.

[33] GoogleRelease 2014. Testing Engineering@Google The Release Process for Google's Chrome for iOS. https://www.youtube.com/watch?v=p9bEc6oC6vw. Accessed: 2019-01-29.

[34] Svein O. Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic Software Product Lines. *IEEE Computer* 41, 4 (2008), 93–95.

[35] Mike Hinchey, Sooyong Park, and Klaus Schmid. 2012. Building Dynamic Software Product Lines. *IEEE Computer* 45, 10 (2012), 22–26.

[36] Rashina Hoda and James Noble. 2017. Becoming Agile: A Grounded Theory of Agile Transitions in Practice. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. 141–151.

[37] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* (1st ed.). Addison-Wesley Professional.

[38] Neil Jones, Carsten Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*.

[39] Christian Kastner. 2019. Feature Flags vs Configuration Options – Same Difference? https://www.cs.cmu.edu/~ckaestne/featureflags/. Accessed: 2020-01-09.

[40] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2009. A model of refactoring physically and virtually separated features. In *Generative Programming and Component Engineering, 8th Intl. Conf., GPCE 2009*. 157–166.

[41] Knight Capital 2013. SEC Charges Knight Capital With Violations of Market Access Rule. https://www.sec.gov/news/press-release/2013-222. Accessed: 2019-09-20.

[42] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46.

[43] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *Proc. of the 41st Intl. Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP 19)*. 91–100.

[44] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proc. of the 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 194–208.

[45] NetFlixRelease 2013. Deploying the Netflix API. https://medium.com/netflix-techblog/deploying-the-netflix-api-79b6176cc3f0. Accessed: 2019-01-29.

[46] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

[47] Phabricator Task 2019. Phabricator/Project management. https://www.mediawiki.org/wiki/Phabricator/Project_management. Accessed: 2019-09-20.

[48] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

[49] Problem Details 2018. The Rise and Fall of Knight Capital: Buy High, Sell Low. Rinse and Repeat. https://medium.com/dataseries/the-rise-and-fall-of-knight-capital-buy-high-sell-low-rinse-and-repeat-ae17fae780f6. Accessed: 2019-09-20.

[50] Pull Request 2019. About pull requests. https://help.github.com/en/articles/about-pull-requests. Accessed: 2019-09-20.

[51] Akond Ashfaque Ur Rahman, Eric Helms, Laurie Williams, and Chris Parnin. 2015. Synthesizing Continuous Deployment Practices Used in Software Development. In *Proceedings of the 2015 Agile Conference (AGILE '15)*. IEEE Computer Society, Washington, DC, USA, 1–10.

[52] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. 2016. Feature Toggles: Practitioner Practices and a Case Study. In *Proc. of the 13th Intl. Conf. on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. 201–211.

[53] ReleaseEng 2015. Guest Editors' Introduction: The Practice and Future of Release Engineering. https://www.youtube.com/watch?v=O3cJQTZXAI8. Accessed: 2019-01-29.

[54] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. ACM, New York, NY, USA, 49–61.

[55] E. Shihab, C. Bird, and T. Zimmermann. 2012. The effect of branching strategies on software quality. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. 301–310.

[56] SwiftSyntax 2019. Github link to SwiftSyntax. https://github.com/apple/swift-syntax. Accessed: 2019-09-20.

[57] TechDebt 2014. Feature Toggles are one of the Worst kinds of Technical Debt. https://dzone.com/articles/feature-toggles-are-one-worst. Accessed: 2019-01-29.

[58] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1 (2014), 6:1–6:45.

[59] Linda Torczon and Keith Cooper. 2011. *Engineering A Compiler* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[60] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The Impact of Continuous Integration on Other Software Development Practices: A Large-scale Empirical Study. In *Proceedings of the 32Nd IEEE/ACM Intl. Conf. on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. 60–71.

[61] Celal Ziftci and Jim Reardon. 2017. Who Broke the Build?: Automatically Identifying Changes That Induce Test Failures in Continuous Integration at Google Scale. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. 113–122.