O'REILLY®

Managing Feature Flags

Deliver Software Faster in Small Increments

Adil Aijaz & Patricio Echagüe

split

The Feature Experimentation Platform for Product Decisions

Split is the leading platform for feature experimentation, empowering businesses of all sizes make smarter product decisions. Companies like Vevo, Twilio, and LendingTree rely on Split to securely release new features, target them to customers, and measure the impact of features on their customer experience metrics. Founded in 2015, Split's team comes from some of the most innovative enterprises in Silicon Valley, including Google, LinkedIn, Salesforce and Databricks. Split is based in Redwood City, California and backed by Accel Partners and Lightspeed Venture Partners. To learn more about Split, contact hello@split.io, or start a 14-day free trial at www.split.io/signup.

www.split.io/oreilly

Managing Feature Flags

Deliver Software Faster in Small Increments

Adil Aijaz and Patricio Echagüe

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Managing Feature Flags
by Adil Aijaz and Pato Echagüe

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Foster
Production Editor: Justin Billing
Copyeditor: Octal Publishing, Inc.
Proofreader: Matthew Burgoyne
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2017: First Edition

Revision History for the First Edition
2017-11-03: First Release

978-1-492-02856-7

[LSI]

Table of Contents

Abstract. v

1. Introduction. 1
The Past, Present, and Future of Feature Flagging 2

2. How Are Feature Flags Commonly Used?. 5
A New Carousel 5
Use Cases 9

3. Succeeding with Feature Flags. 11
The Moving Parts of a Flagging System 11
Implementation Techniques 12
Testing Flagged Systems 14

4. From Continuous Delivery to Continuous Experimentation. 17
Capabilities Your Flagging System Needs 18

5. Conclusion. 21

Abstract

Managing Feature Flags

For almost as long as we've written software programs, we've included ways to control what those programs do at runtime via configuration options or flags. *Feature flags* are a modern application of this concept, focused on accelerating software delivery. What began in the late 2000s as a way for fast-moving software teams to work on half-finished code without disrupting their users has evolved into a standard practice for modern product delivery teams who want to deliver functionality in small increments and learn from their users.

In this book, we'll look at the history of feature flags and, more importantly, learn how teams can successfully apply these techniques. We'll examine different types of feature flags and what makes them different. We'll see some critical code-level techniques to keep our feature flagging code manageable, and we'll explore how to keep the number of flags in our code base to a manageable level.

CHAPTER 1

Introduction

Feature flags (aka toggles, flips, gates, or switches) are a software delivery concept that separates feature release from code deployment. In plain terms, it's a way to deploy a piece of code in production while restricting access—through configuration—to only a subset of users. They offer a powerful way to turn code ideas into immediate outcomes without breaking anything in the meantime.

To illustrate, let's assume that an online retailer is building a new product carousel experience for customers to easily view featured products. Here is a quick example of how it could use a flag to control access to this feature:

```
if (flags.isOn("product-carousel")) {
    renderProductImageCarousel();
} else {
    renderClassicProductImageExperience();
}
```

Here `flags` is an instance of a class that evaluates whether a user has access to a particular feature. The class can make this decision based on something as simple as a file, a database-backed configuration, or a distributed system with a UI.

The Past, Present, and Future of Feature Flagging

The fundamental concept behind feature flags—choosing between two different code paths based on some configuration—has probably been around almost as long as software itself.

In the 1980s, techniques like #ifdef preprocessor macros were commonly used as a way to configure code paths at build time. They were primarily used for supporting compilation to different CPU architectures, but were also commonly used as a way to enable experimental features that would not be present in a default build of the software in question.

Although these preprocessor techniques supported only the selection of code paths at build time, other commands like command-line flags and environment variables have been used for decades to support runtime feature flagging.

Continuous Delivery

Around 2010, a software development philosophy called Continuous Delivery (CD) was beginning to gain traction, centered on the idea that a code base should always be in a state that it could be deployed to production. Enterprise software teams were embracing Agile methodologies and using CD concepts to support incremental deployment into production. Companies that had been releasing software every quarter were beginning to release every two weeks, with huge efficiency gains as a result. Around the same time, startups like IMVU, Flickr, and later, Etsy had been taking this idea to its logical conclusion, deploying to production multiple times a day.

To achieve these incredibly aggressive release cadences, teams were throwing away the rulebook, abandoning concepts like long-lived release branches and moving toward trunk-based development. Feature flagging was a critical enabler for this transition. Teams working on a shared branch needed a way to keep the code base in a stable state while still making changes. They needed a way to prevent a half-finished feature going live to users in the next production deployment. Feature flags provided that capability, and became a standard part of the CD toolbox.

Experimentation

Around the same time as this revolution in software delivery practices, folks like Steve Blank and Eric Ries were sparking a parallel revolution in product management. The Lean Startup methodology brought a heavy focus on rapid iteration (powered by CD, in fact) along with a scientific approach to product management using A/B testing.

Software delivery teams quickly realized that they could use the same feature flagging techniques that they had been using for release management to power their A/B tests. Though this was a great enabler for rapid product development, it also led to some growing pains as product managers suddenly became heavy users of homegrown feature flagging systems that had been built as an internal tool for engineers. These growing pains continue today as the wave of modern product management sweeps through our industry. Later in this book, we discuss how to mitigate some of the issues that arise.

Today, feature flagging systems are seen primarily as a tool for feature management. This is evidenced by the fact that most modern feature flagging systems are oriented around which users are exposed to a given feature. Previously, the primary axis would be around which environment or cluster of servers get a given feature.

What's Next?

For many product delivery organizations, it has become the norm for any change to be managed by feature flags. Product changes are no longer "launched" or "released"—they are incrementally rolled out. As we'll discuss later on, this might involve an initial "champagne brunch" rollout to internal users, followed by a canary release to 5% of regular users, followed by an incremental ramp to 100%.

Rolling out a change involves monitoring the impact of that change. When feature flags were mostly the domain of engineers, the metrics being watched would be technical in nature—CPU load, request latency, database transactions per second. As feature flagging has moved into the domain of product management, the nature of these metrics has shifted toward higher-level business *key performance indicators* (KPIs)—active users, conversion rates, and business transactions per hour.

Today, pioneering organizations are using that feedback loop to automate the rollout (and rollback) of features. This will become more mainstream as more organizations improve their ability to tie feature releases to business metrics, closing and tightening the feedback loop that powers product delivery.

CHAPTER 2

How Are Feature Flags Commonly Used?

To understand how feature flags are used, we'll follow along with a hypothetical team that's just starting with feature flags.

Our software delivery team works at Acme Corporation, a large online retailer. This particular team is responsible for the Product Details page, owning both the user interface for this page and the functionality that powers it. The team has been experiencing a lot of pain recently with branch management and merge conflicts, and has been researching ways to reduce this pain. The tech lead believes that feature flags could be a way to reduce the amount of branching and merging the team needs to do. The team agrees to try this approach for their next big piece of upcoming work.

A New Carousel

The Product Details page includes images of the product, and the team is planning to roll out a new carousel-based user experience for these product images. Typically, they would work on the new UI in a feature branch, perform validation, testing and sign-off against that branch, eventually merging the branch with their shared release branch so that the new functionality can be deployed to production (and into the hands of their users). Instead of that approach, the team plans to experiment with using feature flags as a way to enable *trunk-based development*.

They incrementally develop their new carousel experience on the shared release branch, but add some conditional logic so that it shows the new experience only when appropriate.

Initially, they keep the decision as to whether to show the new experience a hardcoded constant:

```
function renderProductDetailsPage() {
  const showNewProductImageCarousel = false;
  // ... lots of rendering code...

  if (showNewProductImageCarousel) {
     renderProductImageCarousel();
  } else {
    renderClassicProductImageExperience();
  }

  // ... more rendering code...
}
```

This hardwired approach is sufficient for a short while. It allows development of the new carousel code to proceed on a shared branch without affecting other developers (or end users). Developers working on the new functionality can test it out locally by flipping the hardcoded constant to `true`. However, this approach is not sustainable. It's only a matter of time before someone actually checks in a change, which flips the constant! Aside from that, there is no way to run automated tests against the new functionality because it is hardwired off. It's also not possible for a nondeveloper to test out the carousel, for the same reason.

The team needs to allow the decision to show the carousel to be a runtime decision.

Dynamic Decisions

The implementation evolves. Here's what it looks like now:

```
function renderProductDetailsPage() {
  // ... lots of rendering code...

  if (flags.isOn(&ldquo;product-carousel&rdquo;)) {
     renderProductImageCarousel();
  } else {
    renderClassicProductImageExperience();
  }

  // ... more rendering code...
}
```

By deciding to show the new feature as a runtime decision, encapsulated within a `flags` object, you can specify the state of this feature in the context of an automated test. More generally, it allows the user to configure the state of features outside of the code itself, and allows for control over the decision to show the carousel or not via configuration. The potential to show the carousel is always present, but the code path is not always exposed. It is “dark” or “latent” code.

Release Management

Now that flagging decisions are being made via configuration, the team has the option of making that flag configuration environment specific. This means having the ability to turn the new carousel on for a staging environment for testing while still not exposing it to end users in a production environment.

When the team is confident that the new carousel is ready for prime time, it can push a configuration change to production that turns the feature on for users. If something goes wrong, the team can use another simple configuration change to roll back the feature.

Canary Releases

The team has now been using feature flags in this manner for a while. It’s comfortable with using a configuration change to roll out a latent feature to users, but many team members aren’t satisfied that a feature release is an all-or-nothing affair—either the feature is exposed to no users or all users. The team wants to move to a more sophisticated *Canary Release* in which a feature is initially exposed to 5% of users and then rolled out to more and more users if it is performing as expected.

To enable Canary Releasing, the team modifies the feature flagging decision logic to take the current user into account when making a toggling decision:

```
if (flags.isOn("our-new-feature", {user: request.user})) {
   showOurNewFeature();
}
```

Rather than configuring a feature as Off or On, it now configures a rollout percentage (e.g., 5%). The flagging system consistently buckets any given user into a release range from 0% to 100% and uses that position, together with the currently configured rollout percentage, to decide whether to show the current user the given feature. The team can now incrementally roll out a feature by gradually reconfiguring the rollout percentage for the feature in production.

Up to this point, feature configuration has been fairly static. The team has been incorporating the configuration into its code deployment, which means that every feature flag configuration change has required a redeploy. With the addition of Canary Releasing, the team's configuration has become a lot more dynamic, and it decides to move feature flag configuration out to a separate system that allows dynamic reconfiguration on the fly, without a deployment or process restart.

Experiments

The team's product manager notices that it now has the ability to expose a feature to half of its user base. Wouldn't this be adequate for A/B testing? The team agrees that the basic capabilities are there. What's needed is integration with its analytics platform so that it can correlate a user's experimental cohort with their behavior. The team also needs to begin thinking about how to ensure its A/B tests are statistically valid, including making sure that a rollout percentage doesn't change in the middle of an experiment.

Recap

This team's journey is a fairly typical (if somewhat compressed) example of how an engineering organization's usage of feature flags can evolve and expand over time. The team initially intended to use feature flags for a fairly narrow purpose (i.e., avoiding merge conflicts). As time went on, the team became more comfortable with the technique and saw a broader applicability.

We skipped some additional feature flag variants—operations folks wanting to use a feature flag to turn off expensive subsystems when under heavy traffic, and product managers wanting to expose some features to only premium customers.

During this evolution in usage, the code where feature flagging decisions were needed—the *toggle point*—remained in a fairly consistent shape: if feature X is On, execute code path A; otherwise, execute code path B. However, the code making the flagging decision—the *toggle router*—evolved quite dramatically, becoming configurable and starting to perform cohorting and bucketing of users. Likewise, feature flag configuration became both more complex and more dynamic.

Use Cases

At a high level, feature flags accelerate development by powering the following use cases:

Continuous Delivery

Central to Continuous Delivery (CD) is the idea that your product is always in a releasable state. Teams practicing CD also often practice trunk-based development, in which all development happens on a single shared branch (i.e., trunk, master, and head) with feature branches being short lived (i.e., going only a few days before being merged). Trunk-based development helps teams to move faster by integrating their changes constantly and avoiding the "merge hell" caused by long-lived branches. The only way to ensure that a shared branch is always releasable is to hide unfinished features behind flags that are turned off by default. Otherwise, CD can turn into continuous failure.

Testing in production

New feature releases are preceded by functional QA and performance testing. This testing requires the creation of a User Acceptance Test (UAT) or staging environment with a sample of production data. This sampling and copying of production data is problematic. For starters, it's a red flag for data privacy and security-conscious teams. Second, these staging environments are not a faithful replica of production infrastructure, so you need to take any performance evaluations with a grain of salt.

Feature flags allow teams to perform functional and performance tests directly on production with a subset of customers. This is a secure and performant way of understanding how a new feature will scale with customers.

Kill a feature

By having a feature behind a flag, you can not only roll it out to subsets of customers, but you also can remove it from all customers if it is causing problems to customer experience. This idea of "killing" a feature is better than having to do an emergency fix or a code rollback.

Ops toggles are a generic term for the idea of having certain features reside permanently behind a flag that you can kill or turn off, maintaining a minimum viable functionality of the product under high load or exceptional circumstances

Migrating to microservices
: Microservices is the practice of breaking up a huge, monolithic release into many discrete services that you can roll out in independent release schedules. Like any large architectural shift, a monolith breakup is best tackled as a series of small steps that incrementally moves the system toward the desired state. By taking advantage of the capabilities of a feature flagging framework, you can make this transition safe, and in a controlled manner.

Paywalls
: You can use feature flags to permanently tie features to subscription types. For instance, a feature could be available to every customer as part of a free trial, but is gated afterward by the customer buying a premium subscription.

CHAPTER 3

Succeeding with Feature Flags

As highlighted in previous chapters, adding feature flagging capabilities to the code base can provide a broad range of benefits. However, feature flags can also add complexity to the code base and reduce internal quality. It's not uncommon for teams who have recently embraced feature flagging to feel that they have added some tax to their software system. Code can become littered with conditional flag checks, and it can seem that every part of the code base (and every test) has a dependency on the feature flagging infrastructure.

In this chapter, we look at some specific techniques that you can use to ensure that feature-flagged code is readable, maintainable, and testable. Most of the techniques we discuss are really just good general software design principles applied in the context of feature flagging code. Similar to test code, feature flagging code seems to be treated as second-class code that doesn't need the same level of thought as "regular" code. This is not the case, as you'll see.

The Moving Parts of a Flagging System

Let's define the various moving parts involved in a feature flagging decision, based on the following example decision:

```
if (flags.isOn("product-images-carousel", {user: request.user})) {
  renderProductImagesCarousel();
} else {
  renderClassicProductImages();
}
```

Toggle Point

The toggle point is the place in the code base where you choose a code path based on a feature flag. In this example it's the `if` conditional statement in which we call `flags.isOn(…)`.

Toggle Router

The toggle router is the code that actually decides the current state (or "treatment") of a given feature and provides that treatment to the toggle point. In our example, `flags` is an instance of a toggle router.

Toggle Context

Toggle context is contextual information that is passed from the toggle point to the toggle router. The toggle router can use this context when deciding what treatment to return to the toggle point. In a typical web application, the toggle context is based on the current request being processed. In a mobile application, the toggle context might be based on the device running the application. In our example here, the toggle context consists of the user being serviced by the current request.

Toggle Config

Toggle config is the set of rules that the toggle router uses to decide the treatment of a given feature. For instance, the config in the previous example might range from the simple "only for user X" to the complex "10% of users in California." When beginning with flagging, teams configure toggle routers via config files or database records. As the scope extends to include product managers and the experimentation use case, configurations increase in complexity and often require a friendly UI for editing purposes.

Implementation Techniques

Here are some specific techniques that can be used to ensure that feature-flagged code is readable, maintainable, and testable.

Keep Decision Point Abstracted from Decision Reason

It is important to keep the toggle config abstracted behind the toggle router for two key reasons.

First, it makes it easier to increase the complexity in a toggle config without affecting the rest of the code base. For instance, in the beginning, the Acme Corporation team might want to show the product image carousel only to employees for dogfooding the product. As part of the launch, the team might want to show the carousel to only 20% of visitors from California who use a mobile device. By abstracting the toggle config, the complexity of the configuration can be increased without affecting the rest of the code.

Second, abstraction helps with testing the code. We cover this subject later in this chapter, but suffice it to say that abstraction helps with mocking the toggle router and lets us try different configurations in a preproduction environment than we would use in a production environment.

Avoid Multiple Layers of Flags

A common mistake teams new to feature flagging make is to add toggle points for the same feature at all levels of the stack. They put a feature flag in the UI layer, in the mobile application, and in the backend code. At best, this causes confusion. At worst, it can lead to a feature being turned on in the UI layer but off in the backend.

One way to avoid this mistake is to follow the principle of *highest common access point*. In simple terms, place the toggle point in the highest layer of the stack that is common to all user traffic. If a feature is accessible both on the web and in the mobile application, the highest common point for traffic from these two clients is the backend, which is the ideal location for placing this toggle point. On the other hand, if the feature is available only in the UI, place the toggle point in the UI.

This approach avoids duplication of toggle points and the pitfalls that come with it.

Retire Features

Although feature flags reduce the pain of integrating long-lived branches and the resulting bugs due to bad merges, they come with their own set of challenges. Instead of a large number of feature branches, a feature-flagged code base has a large number of toggle points. The conditional logic of these toggle points is a form of code smell; the longer it stays in the code, the more difficult it becomes to test or debug that code.

A best practice to avoid these problems is to have a process around retiring and removing feature flags from code. Processes are best enforced via code, not meetings, so it is best to automate the process of tallying the age of feature flags and opening tickets against engineering owners to clean up the flags.

Testing Flagged Systems

Feature flags throw a wrench into traditional QA techniques. Traditionally, there is one release branch to be certified. The branch might have a number of features added to an existing product which are available to all customers the moment the release is deployed. However, there is only one version of the product to test.

In a feature-flagged world, each one of the new features may be "on" or "off," depending on the customer. Instead of a single product to test, there are hundreds or thousands of versions of the product depending on the combination of features that are turned on or off for the customer. It is an unreasonable expectation for the QA team to certify all possible combinations of features before the release.

Test Individual Features, Not Combinations

Testing the combinatorial explosion of multiple feature flags sounds good in theory, but in practice, it is neither good engineering nor needed. Instead, it is best to test each feature in isolation for all states of the feature (e.g., on and off).

This is because most features do not interact with another. The product carousel is likely independent of the new preferences tab that is being rolled out. It is this assumption of independence that makes it possible for you to test features in isolation.

However, when you cannot assume independence, you should test all possible combinations. As an example, let's look at two features: one controlling a new home page design, and another controlling the product carousel redesigned for the new home page. These two features are not independent. In fact, the toggle config for the latter can take into account the state of the former. In this scenario, the QA team should test all possible combinations of the two features.

How to Automate Testing?

Automation is central to the Agile philosophy. The abstraction of a toggle router is key to making it easier to automate testing for feature-flagged code.

From a unit-testing perspective, you can mock the toggle router to generate the state of the features to be tested.

When looking at regression tests, a toggle router can be backed by a configuration file that hardcodes the state of every feature for a specific series of regression tests. To run the entire suite, you can run tests with the router initialized with the appropriate configuration file.

CHAPTER 4

From Continuous Delivery to Continuous Experimentation

In the introduction of this book, we touched upon the convergence of continuous delivery (CD) and experimentation driving "lean product development," with feature flags being the foundational element powering this convergence.

Let's explore the trends and practices that are driving this convergence.

CD became a well-defined strategy among forward-thinking engineering teams, and stemmed from the need for businesses to rapidly iterate on ideas. At the same time, product management teams were adopting lean product development concepts, such as customer feedback loops and A/B testing. They were motivated by a simple problem: up to 90% of the ideas they took to market failed to make a difference to the business. Given this glaring statistic, the only way to be an effective product management organization was to iterate fast and to let customer feedback inform investment decisions in ideas.

Common elements began to emerge, connecting both these trends in day-to-day software development and delivery. These elements included the need for rapid iteration, safe rollouts through gradual exposure of features, and telemetry to measure the impact of these features on customer experience. The resulting outcome is that modern product development teams are beginning to treat CD and experimentation (i.e., a more generic term for A/B testing) as two sides of the same coin. Core to both of these practices is the foundational technology of feature flags.

We can further illustrate this convergence through real-world examples of how teams at LinkedIn, Facebook, and Airbnb release every feature as an experiment and how every experiment is released through a flag. These teams have shown that the future of CD is to continuously experiment.

Furthermore, this convergence is now creating a need for tooling that can support this new paradigm of continuous feature experimentation.

Capabilities Your Flagging System Needs

In this section, we will cover some ways a feature flagging system can evolve to support experimentation.

Statistical Analysis of KPIs

A significant step in the evolution of a feature flagging system into an experimentation system is to tie feature flags to Key Performance Indicators (KPIs). This involves tracking user activity, building data ingestion pipelines, and investing in statistical analysis capabilities to measure KPIs within the treatment and control groups of an experiment (on or off for a feature flag). Statistically significant differences between the groups can be used to decide whether an experiment was successful and should continue ramping toward 100% of customers.

The anticipated outcome then becomes: ideas turned to products with speed from feature flags, and products turned to outcomes with analytics from experimentation.

Multivariate Flags

Feature flagging is a binary concept: a flag is either on or off. Similarly, experimentation is a binary concept. There is a treatment and a control. Treatment is the change we are testing, and control is the baseline for comparison. However, it is common for an experiment to compare multiple treatments against a control. For example, Facebook might want to experiment with multiple versions of its newsfeed ranking algorithm. You can enhance a feature flagging system to support this experimentation need by changing its interface from

```
if (flags.isOn("newsfeed-algorithm")) {
   // show the feature
} else {
   // do not show the feature
}
```

to:

```
treatment = flags.getTreatment("newsfeed-algorithm");
if (treatment == "v1") {
   // show v1 of newsfeed algorithm
} else if (treatment == "v2") {
   // show v2 of newsfeed algorithm
} else {
  // show control for newsfeed algorithm
}
```

Targeting

A simple feature flag is global in nature—it is either on or off for all users. However, experimentation requires more granular capabilities for targeting and ramping. On the targeting side, an experiment might need to be defined for a segment of customers for whom the feature will be turned on. Using the example of Facebook's newsfeed, Facebook might want to experiment on a ranking algorithm for a particular group of users in a specific geographic location. To accommodate this need, a flagging system can evolve to accept customer targeting dimensions at runtime. This pseudocode will clarify:

```
treatment = flags.getTreatment("newsfeed-algorithm",
  {user: request.user, age:
"35", locale: "U.S"})
```

In an ideal implementation, the flagging system should abstract the details of the dimensions away from the developer so that the developer simply has to call the following:

```
treatment = flags.getTreatment("newsfeed-algorithm",
  {user: request.user})
```

Randomized Sampling

To infer causality between a feature experiment and changes in KPIs, we need a treatment and a control group. Treatment is the group exposed to the new feature or behavior; control is the group seeing baseline behavior. The only difference between these groups should be the feature itself. This concept is called *control for biases*. Using our recent Facebook example again, the treatment and control algorithms should both include teenagers from the United States. If the treatment algorithm is given to Australian teenagers while the control is given to men in the United States in their 30s, you cannot infer causality between the new algorithm and KPI changes because of the demographic differences between treatment and control.

You can use a feature flag to serve this need by adding the ability to randomly give a feature to a percentage of customers. As an example, Facebook would update its feature flag to serve the new algorithm to 50% of randomly selected users of the target age group and geographic location, and the control algorithm to the remaining 50%. This percentage rollout is called *randomized sampling*.

The key point here is randomization. If two different Facebook experiments are both at 50/50 exposure across the same segment of users, the 50 percent of users seeing the treatment for one experiment should not overlap—except by chance—with the remaining 50% seeing the treatment for the other experiment. This is possible only through randomization.

Without randomization, one experiment can bias the results of the other, nullifying any causality between the feature and changes in KPIs.

Version History

A feature flag's historical state is usually unimportant. What matters is that the flag is either on or off at any given moment. Version history, however, is important for experiments. Let's assume that we run a 30/70 experiment across all Facebook users (i.e., 30% in treatment and 70% in control). If we change the experiment to 50/50, any KPI impact measured in the 30/70 state is statistically invalid for the 50/50 state.

This means that a feature flagging system should keep a versioned history of changes to the configuration of a flag, so statistical analysis of the KPI impact can respect version boundaries. Practically speaking, you can achieve this can by pushing the version history of feature flags into the analytics system serving experimentation needs.

CHAPTER 5

Conclusion

Agility is a driving force in modern product development. Businesses that develop better products faster are able to out-innovate their competition. The software industry has developed many techniques to serve this need for agility: from trunk-based development to Continuous Delivery to microservices.

In this book, we covered a simple, yet powerful primitive that is central to many of these techniques—the feature flag. Its power lies in breaking down a product into a set of features that can be dynamically targeted to customers without redeploying code. It takes companies on the journey from Continuous Integration to Continuous Delivery, and finally, to Continuous Experimentation.

The ultimate benefit for companies adopting feature flags is not only an increase in speed and quality, but also a significant reduction in risk to their product development practices, which leads to superior customer experiences.

About the Authors

Adil Aijaz is CEO and cofounder at Split Software. Adil brings more than 10 years of engineering and technical experience, having worked as a software engineer and technical specialist at some of the most innovative enterprise companies, such as LinkedIn, Yahoo!, and most recently, RelateIQ (acquired by Salesforce). Prior to founding Split in 2015, Adil's tenure at these companies helped build the foundation for the startup, giving him the needed experience in solving data-driven challenges and delivering data infrastructure. Adil holds a Bachelor of Science degree in computer science and engineering from UCLA, and a Master of Engineering degree in computer science from Cornell University.

Patricio "Pato" Echagüe is the CTO and cofounder at Split Software, bringing more than 13 years of software engineering experience to the company. Prior to Split, Pato was most recently at RelateIQ (acquired by Salesforce) where he joined as one of the first three engineers leading the majority of the company's data infrastructure efforts. Prior to RelateIQ, Pato was an early employee at Datastax (the creators of the Apache Cassandra project) where he was one of the lead committers for the open source Java client Hector, creating the first enterprise offering and coauthoring the Cassandra Filesystem (CFS) that was used to replace the HDFS layer from Hadoop to Cassandra. Other professional experiences include software engineering roles at IBM, VW, and Google. Pato holds a Master of Information Systems in Software Engineering from the Universidad Tecnológica Nacional, Argentina.