O'REILLY®

Compliments of split

# Feature Flag Best Practices

## Advanced Tips For Product Delivery Teams

Pete Hodgson & Patricio Echagüe

split

# Split powers your product decisions with a unified solution for feature flagging and experimentation.

Split is the leading product decisions platform for engineering and product teams who want to rapidly – and safely – deliver valuable software to customers. Unlike systems that are homegrown, point solutions, or those not built for engineering and product teams, only Split provides a unified feature flag and experimentation solution that is trusted by engineers and built for teams of any size to make data-driven decisions. Split ingests large volumes of data and performs near real-time statistical analysis to look for the impact of every feature release on metrics that matter to you. Engineering and product teams at Twilio, Salesforce, and JPMorgan Chase & Co. trust Split with their feature management and decisions. To learn more about Split, contact hello@split.io, or get started for free at www.split.io/signup.

# Feature Flag Best Practices

## *Advanced Tips for Product Delivery Teams*

*Pete Hodgson and Patricio Echagüe*

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

***Feature Flag Best Practices***
by Pete Hodgson and Patricio 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.

**Acquisitions Editor:** Nikki McDonald
**Development Editor:** Virginia Wilson
**Production Editor:** Deborah Baker
**Copyeditor:** Octal Publishing, LLC

**Interior Designer:** David Futato
**Cover Designer:** Karen Montgomery
**Illustrator:** Rebecca Demarest

January 2019: First Edition

**Revision History for the First Edition**
2019-01-18: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492050445* for release details.



This work is part of a collaboration between O'Reilly and Split Software. See our *statement of editorial independence*.

978-1-492-05042-1

[LSI]

# Table of Contents

# CHAPTER 1
# Introduction

Feature flags—also known as *feature toggles*, *feature flippers*, or *feature bits*—provide an opportunity for a radical change in the way software engineers deliver software products at a breakneck pace. Feature flags have a long history in software configuration but have since "crossed the chasm," with growing adoption over the past few years as more and more engineering organizations are discovering that feature flags allow faster, safer delivery of features to their users by decoupling code deployment from feature release. Feature flags can be used for operational control, enabling "kill switches" that can dynamically reconfigure a live production system in response to high load or third-party outages. Feature flags also support continuous integration/continuous delivery (CI/CD) practices via simpler merges into the main software branch.

What's more, feature flags enable a culture of continuous experimentation to determine what new features are actually desired by customers. For example, feature flags enable A/B/n testing, showing different experiences to different users and allowing for monitoring to see how those experiences affect their behavior.

In this book, we explain how to implement feature-flagged software successfully. We also offer some tips to developers on how to configure and manage a growing set of feature flags within your product, maintain them over time, manage infrastructure migrations, and more.

CHAPTER 2

# The Moving Parts of a Feature-Flagging System

At its core, feature flagging is about your software being able to choose between two or more different execution paths, based upon a flag configuration, often taking into account runtime context (i.e., which user is making the current web request). A toggle router decides the execution path based on runtime context and flag configuration.

## Creating Separate Code Paths

Let's break this down using a working example. Imagine that we work for an ecommerce site called acmeshopping.com. We want to use our feature-flagging system to perform some A/B testing of our checkout flow. Specifically, we want to see whether a user is more likely to click the "Place your order!" button if we enlarge it, as illustrated in Figure 2-1.

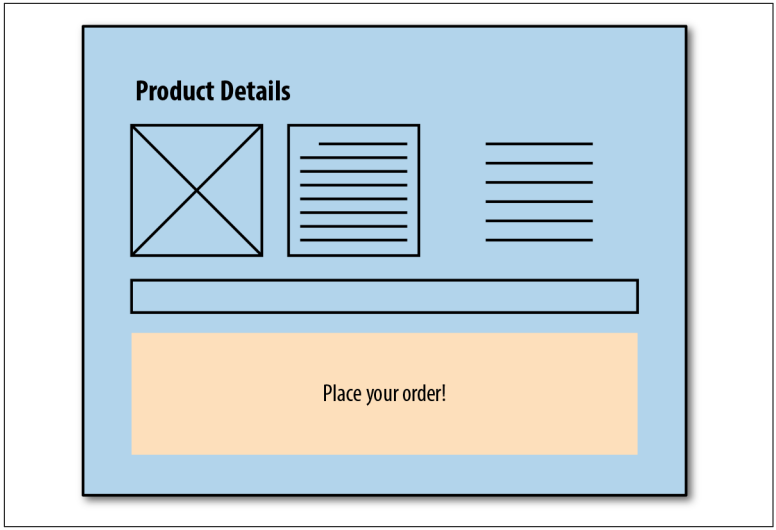


*Figure 2-1. acmeshopping.com A/B testing*

To achieve this, we modify our checkout page rendering code so that there are two different execution paths available at a specific toggle point:

```
renderCheckoutButton(){
  if(
    features
      .for({user:currentUser})
      .isEnabled("showReallyBigCheckoutButton")
  ){
    return renderReallyBigCheckoutButton();
  }else{
    return renderRegularCheckoutButton();
  }
}
```

Every time the checkout page is rendered our software will use that `if` statement (the toggle point) to select an execution path. It does this by asking the feature-flagging system's toggle router whether the `showReallyBigCheckoutButton` feature is enabled for the current user requesting the page (the current user is our runtime context). The toggle router uses that flag's configuration to decide whether to enable that feature for each user.

Let's assume that the configuration says to show the really big checkout button to 10% of users. The router would first *bucket* the user, randomly assigning that individual to one of 100 different buckets.

The router would then report that the feature is `enabled` if the current user has landed in buckets 0 through 9, but `disabled` if they'd landed in any of the remaining buckets (10 through 99).

When using a feature-flagging system, we often want to control which users see a feature. We might want to initially limit rollout of a new feature to a set of beta users, or expose a new functionality to only paying customers, or to only 10% of traffic. Most feature-flagging systems allow you to configure a feature to support these different targeting scenarios based on a few different strategies, such as canary release, dark launching, targeting by demographic, account, or license level. When the benefits of feature flags are proven, additional use cases emerge and usage quickly grows, so it's a good idea to establish best practices from the start.

CHAPTER 3

# Best Practice #1: Maintain Flag Consistency

Our hypothetical ecommerce company, acmeshopping.com, has a landing page in one of the main sections of the web portal where customers can search and see search results, as they look for items they would like to buy. Sometimes, Acme's customers search without buying. Often, users return and search again for the same items, and eventually some of them do a checkout. Acme is trying to reduce the time it takes users to find those frequently searched items by adding a new header section showing these items. Let's call them "Previously Seen Products."

This new section is gated by a feature flag named `new_search_relevance`. So, when `new_search_relevance` is enabled, the web portal will first display the "Previously Seen Products" section at the top of the search result part of the page.

You, as the person in charge of the rollout, set up this new feature to initially be seen by 10% of the user population. Exposing the feature from zero to a subset of the population is often called *ramping up a feature*. Here, a feature was ramped to 10%.

As demonstrated in Figure 3-1, the expectation is that if user A visits your site and your feature-flagging system decides that user A should see this feature (variation "on" of the feature), user A should then continue to see this same variation of the feature no matter how many times the flag is evaluated, assuming no external changes have occurred (for example, the flag definition changed).

Increasing the exposure to a broader user population should not affect the current exposure of variations to users—if a user experienced a feature when it ramped to 40%, that user should continue to see it as it ramps to 60%, 80%, and so on. In others, existing allocations should remain intact.

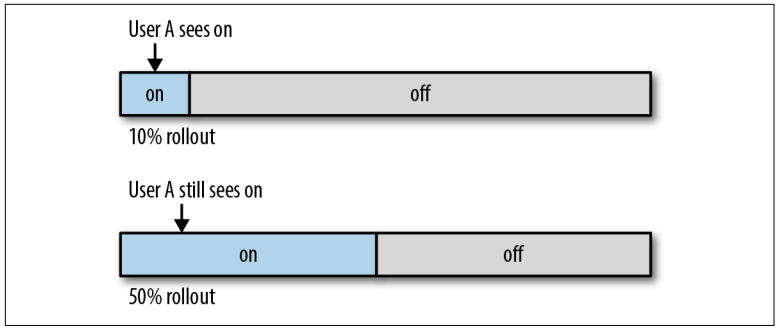


*Figure 3-1. Flag consistency during feature ramping*

A particular case occurs if you were to "de-ramp" (reduce exposure of) the feature; for example, reducing exposure from 10% to 5%, as in Figure 3-2. We know that user A was part of the "on" group in the 10% sample. Unless your feature-flagging system has the notion of "memory" to remember the prior allocation of A, there is little you can do to maintain user A in the "on" group when reducing exposure, just because we don't know *a priori* whether user A will be in the "on" or "off" group.

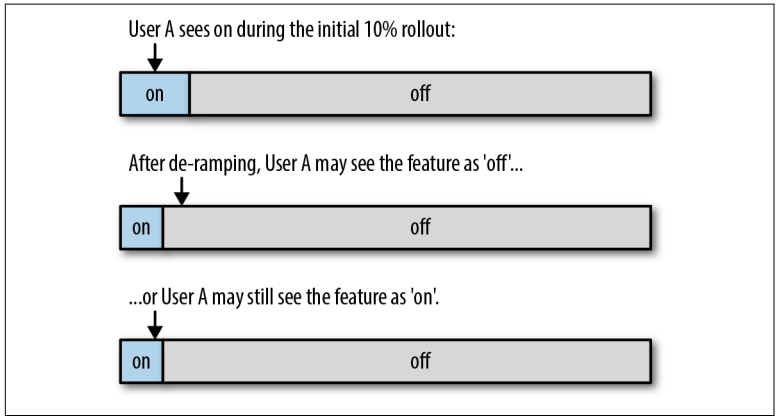


*Figure 3-2. Flag consistency during feature de-ramp*

CHAPTER 4

# Best Practice #2: Bridge the "Anonymous" to "Logged-In" Transition

When the same user crosses from an "anonymous" to a "logged-in" user, deciding whether to maintain a consistent feature treatment can be a challenge. The need to maintain the same feature set as a user switches from anonymous to logged in is encountered more often in organizations that serve consumers (B2C) than in enterprise organizations. In B2C software, consumers generally start as visitor or anonymous, perform some actions like adding elements to a shopping cart, and then later log in to complete the purchase. There are several strategy options, and you will need to choose what is right for your user's experience.

Our sample company, acmeshopping.com, will have this problem as well. An acmeshopping.com customer generally enters the site as an anonymous user and is assigned a visitor ID as a cookie. The user might later complete a login sequence.

When dealing with an anonymous user, you first need to decide whether it's important to maintain feature-flag consistency during the transition from visitor ID to user ID.

For example, if your application is more transactional in nature, such as a collaboration or networking site, perhaps maintaining feature-flag consistency from session to session will not be as important. However, if your test involves different pricing options, main-

taining consistency will be important because you'll want to show the same price at every session.

If you decide that maintaining a consistent feature-flag treatment is important, the technique to achieve consistency is to track the user's visitor ID as a cookie, as shown in Figure 4-1, and then associate it to the user ID immediately after login when the user is created. We recommend setting the cookie expiration time to be semi-permanent to ensure that the user is served a consistent experience over the life of any feature flags.

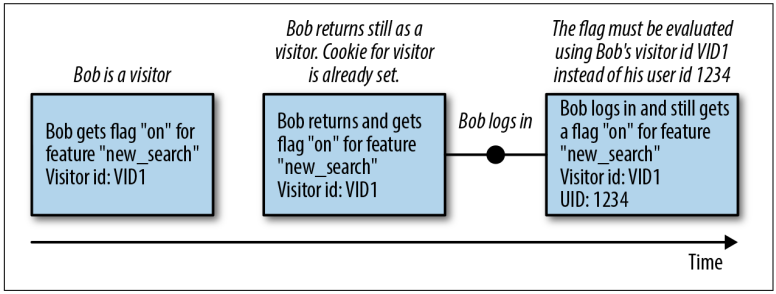


*Figure 4-1. Crossing the "anonymous" to "logged-in" barrier*

CHAPTER 5

# Best Practice #3: Make Flagging Decisions on the Server

In addition to logic implemented on the server side, modern web applications often contain rich client-side logic. When applying feature flagging, we usually have a choice between making our toggling decision client side or server side, and there are trade-offs to consider either way.

## Performance

By moving flagging decisions to the server side, you gain user-perceived performance. Single-page applications are already making a server-side call to render the data needed for the UI. At this time, you could also make a call to a feature-flag service, so one network call fetches all feature-flag evaluations with the server-side data.

## Configuration Lag

One way in which engineers improve performance of an application is to cache data locally, thereby reducing network latency. This has an impact on where the feature flag decision should be made. You could opt to proactively request all flagging decisions for a specific runtime context (i.e., current user, browser, and geolocation) from the server. Or, you could just request the current feature-flagging configuration and make flagging decisions using a client-side toggle router. In both approaches, you are at risk of *Configuration Lag*.

When a Site Reliability Engineer hits a kill switch to disable a feature that's going sideways, how does your client-side feature-flagging system find out? Do you poll for updates on a regular basis? Maybe there is a server-side push system that informs you of a flag configuration change. What if your client-side code doesn't have network connectivity when that push goes out? These are all variants of cache invalidation challenges—one of the famously difficult problems in computer science. Keeping your decisions on the server side helps to reduce these challenges.

In addition, the UI often won't have access to a lot of dimensional data about the user for security purposes. For example, there might not be history or behavioral data on a mobile application that is needed to roll out your features. This data is on the server side and is another reason to keep feature-flag evaluations on the server side.

## Security

Whenever you move a feature-flagging decision to the client, you're exposing information about the existence of those decisions—anyone who's able to log in to your product can potentially see what product features are under active management and can also manipulate which variants they experience. If you're concerned about industrial espionage or particularly nosy tech journalists, this might be relevant, but that's unlikely to be the case for the typical feature-flag practitioner.

## Implementation Complexity

Most delivery teams working with feature flags need the ability to make a server-side toggling decision. If a team also begins making toggling decisions on the client side, it significantly increases the complexity of its feature-flagging system. There will now be two parallel implementations, which are likely to be implemented in multiple languages (unless you've opted to implement your backend in JavaScript, in addition to your frontend). These parallel implementations need to remain synchronized and make consistent toggling decisions. And, as discussed earlier, if you begin adding client-side caching into the mix, things can get quite complicated.

Given these performance and complexity concerns, we recommend keeping feature-flagging decisions on the server side.

CHAPTER 6

# Best Practice #4: Incremental, Backward-Compatible Database Changes

Whenever we make code changes to a production system, we need to take existing database data—and, more generally, any shared persistent state—into account. The database schema in place needs to be compatible with the expectations of any newly deployed code; sometimes that means applying a migration to our database schema, as illustrated in Figure 6-1.

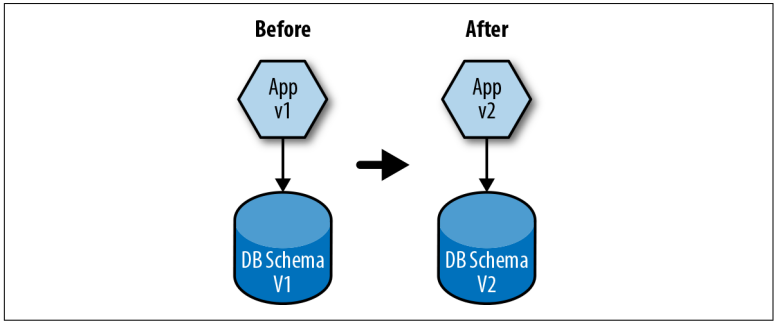


*Figure 6-1. Database schema before and after migration*

We can orchestrate a code deployment and its corresponding database migration in a few ways.

## Code First

We can perform the code deployment first, shown in Figure 6-2, making sure that the new version of our code is backward-compatible with the existing database schema.

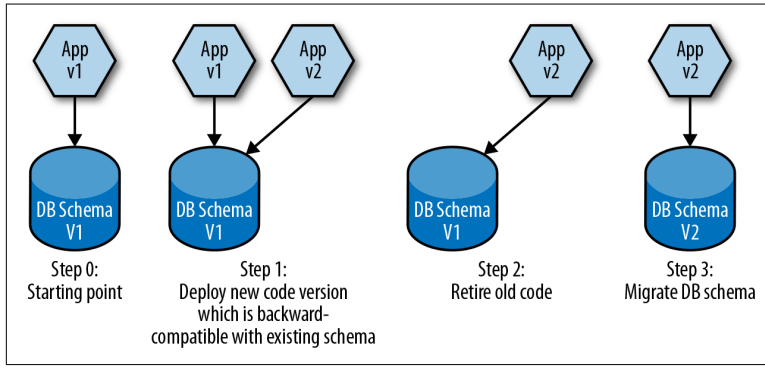


*Figure 6-2. Code-first approach*

## Data First

Alternatively, we can perform our database migration first, as shown in Figure 6-3, which means that we must ensure that the new schema is backward-compatible with the existing code.

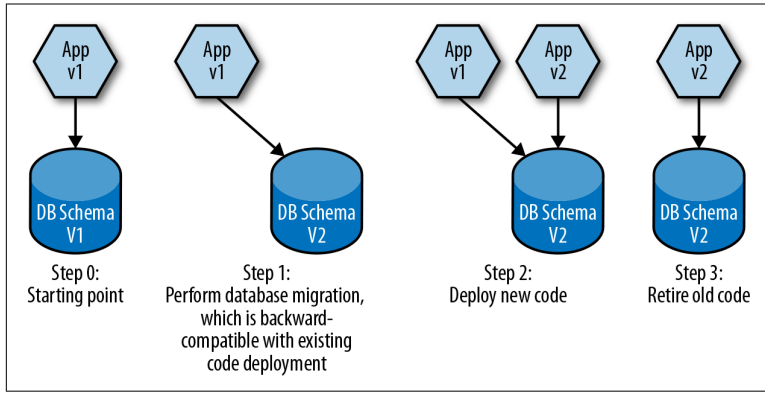


*Figure 6-3. Data-first approach*

## Big Bang

In simple systems, there's a third option (Figure 6-4): update data and code simultaneously in a lockstep deployment in which you stop your system, update your data to support your code change, and then restart the system with your new code.

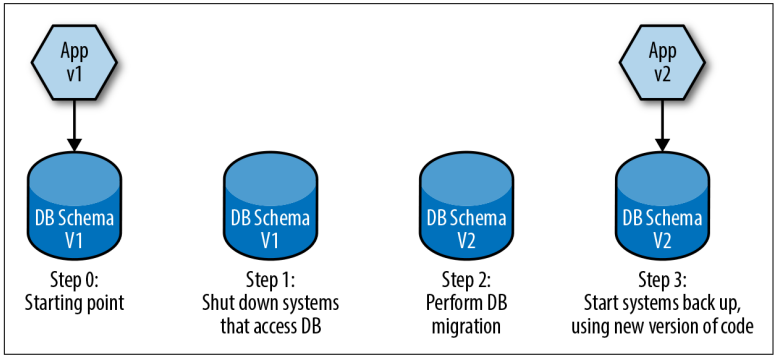


*Figure 6-4. Big-Bang approach*

With the Big-Bang approach, you don't need to worry about backward or forward compatibility, because you're updating both data and code in concert. New code will never see old data, and old code will never see new data.

Feature flagging brings an additional challenge in this area. Code paths that are managed by an active feature flag exist in a sort of quantum superposition, in which any given execution might go down one side of the code path or the other. This means that your data must be compatible with *both* code paths for the duration that the managing feature flag is active. This precludes the option of performing a lockstep migration of both data and code, because after that migration, your data schema will not support the old code path that could still be selected by your flagging system.

## Expand-Contract Migrations

When a feature-flagged code change requires a corresponding data schema migration, this migration must be performed as a series of backward- or forward-compatible changes, sometimes referred to as an *Expand-Contract migration*, or a *Parallel Change*. The technique is called Expand-Contract because the series of changes will consist of an initial data-first change that "expands" the schema to accom-

modate your code change, followed by a code-first change that "contracts" the schema to remove aspects that are no longer needed.

Let's see how one of these Expand-Contract migrations might work in practice. At acmeshopping.com, the shipping address for an order previously was stored as a set of columns within the Orders table. We want to normalize how addresses are stored, extracting the shipping address out of the Orders table into a ShippingAddress table, referenced by a foreign key in the Orders table, as depicted in Figure 6-5.

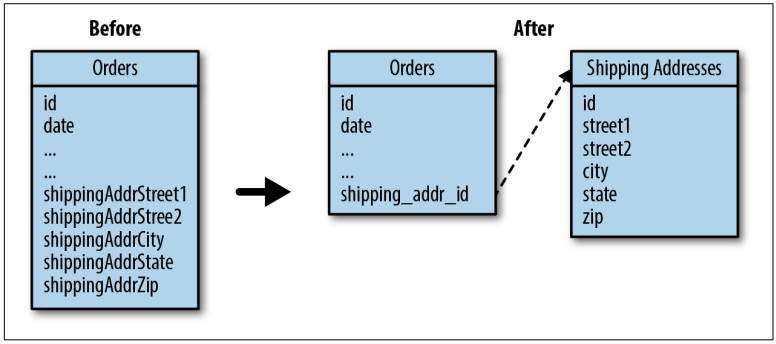


*Figure 6-5. Expand-Contract migrations*

We want to perform this change safely, so we use a feature flag to manage the change, and we perform the migration using the following series of Expand-Contract changes:

1. We Expand the schema by performing a data-first change, which adds the new ShippingAddress table, as well as a nullable `shipping_addr_id` foreign key to the Order table. We don't make any code changes at this point, because this change is backward-compatible with existing code.
2. We start a code-first change, rolling out a code change that will write to *both* the old shipping address columns in the Order table and the columns in the new ShippingAddress table.
3. We perform a one-time data migration, which backfills the ShippingAddress table, creating a row for each existing Order, linked back to the Order via the `shipping_addr_id` foreign key. We also remove the nullability from `shipping_addr_id`, given that we can be sure that it will now always be set.

4. Now that all existing data is in the new table, we can contract the schema by performing a code-first change. We roll out a code change to read only from the ShippingAddress table. At this point, no code is referencing the old `shipping_addr` columns in Order, so we can drop those columns from the database.

## Duplicate Writes and Dark Reads

When performing these Expand-Contract migrations, there is a middle phase during which our system needs to support both the old and the new schemas. We achieve this by doing Duplicate Writes—whenever we need to create or update a shipping address, we do that in the old fields *and* in the new table.

Because we have data duplicated in two places, we also need to decide from where we should read data. For a complex migration, it's advisable to perform *Dark Reads*, in which we read from both sources and compare the results to make sure that everything matches. This allows us to gain confidence in our change, safe in the knowledge that if things don't seem to be correct, we can easily roll back to the previous schema (using a feature flag) while we debug the problem.

## Working with Databases in a Feature-Flagged World

We've seen that feature flagging precludes some migration techniques, such as making Big-Bang changes. Instead, we recommend using patterns like Expand-Contract and Double Writing to perform a safe, incremental migration of your data. For complex migrations, techniques like Dark Reads will help gain confidence and mitigate the risk of data inconsistencies. Also note that some database changes are just too big and still need to be scheduled and change-controlled in a more traditional fashion.

CHAPTER 7

# Best Practice #5: Implement Flagging Decisions Close to Business Logic

At acmeshopping.com we want to experiment with the idea of offering free shipping on all orders that total more than $50. We want to manage the rollout of this feature using a feature flag. An obvious first question to ask is where to implement that flagging decision. Let's take a look at the architecture of acmeshopping.com's backend systems, shown in Figure 7-1, so that we can evaluate our options.

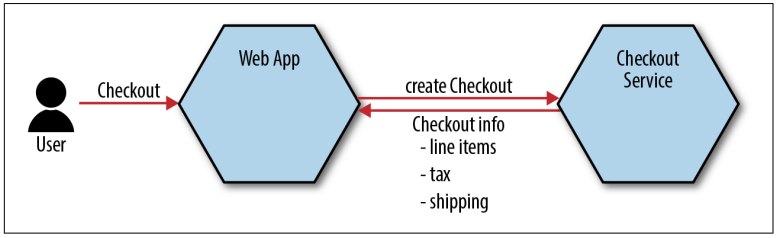


*Figure 7-1. Backend systems involved in creating a new checkout*

At acmeshopping.com, all user-facing web interactions go through the cunningly named Web App.

When a user is done filling their cart and is ready to check out, they make a request to the Web App, which in turn asks the Checkout service to create a new checkout. The Checkout service does this and returns information about the checkout back to the Web App,

including details about the shipping costs associated with the items in this checkout (at Acme, shipping costs are not calculated until the user finalizes the order and checks out).

The Checkout service calculates shipping costs, so it seems obvious that the flagging decision for the "free shipping" feature should reside within that service, as depicted in Figure 7-2.

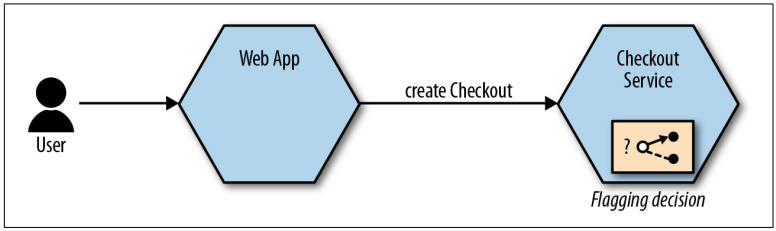


*Figure 7-2. Placing the "free shipping" flag decision in the Checkout service*

But is that really the best place to make this flagging decision? Suppose that we initially want to roll out free shipping for internal testing (an example of the virtual UAT technique that we discussed earlier). This means that we'd need to take into account which user is requesting the checkout when deciding whether to allow free shipping. The user would need to be part of the *runtime context* for the flagging decision. However, our Checkout service doesn't know anything about users—its sole responsibility is creating and managing checkouts.

One way to solve this would be to have the Web App pass the necessary runtime context through to the Checkout service, as demonstrated in Figure 7-3, by telling it which user is requesting to check out. This would allow the Checkout service to make the flagging decision locally.

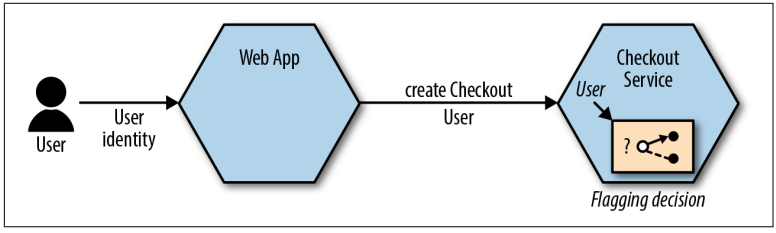


*Figure 7-3. Web App passes the necessary runtime context through to the Checkout service*

However, with this approach, we'd be forcing the Checkout service to become aware of the concept of users, which is beyond the scope of that service. We'd like to keep the Checkout service entirely focused just on creating and managing checkouts.

On the other hand, the Web App is already deeply aware of the concept of users and already has the context of which user is requesting a checkout. The Web App is actually in a better position to make our "free shipping" flagging decision, as shown in Figure 7-4.

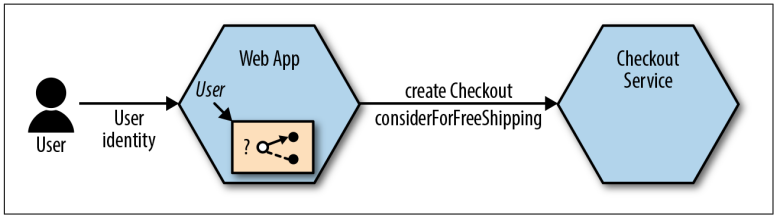


*Figure 7-4. Placing the "free shipping" feature decision in the Web App*

Now, every time a user proceeds to checkout, the web app will use the feature-flagging system to decide whether this user should receive free shipping. That decision is then passed to the Checkout service as an extra parameter when creating the checkout. Note that the Checkout service still has the responsibility of determining whether the order is eligible for free shipping; that is, if the total value of the order is more than $50. We've avoided having to change the scope of our various components just to support a flagging decision, and we've kept the underlying business logic that's powering the feature in the right place.

## A Rule of Thumb for Placing Flagging Decisions

We can extract a general rule of thumb from this example: make your flagging decision as close to the business logic powering the flag while still retaining sufficient runtime context to make the flagging decision.

This ensures that flagging decisions are encapsulated as much as possible while avoiding the need for every part of your codebase to be aware of the concepts like User, Account, Product Tier, and so on, which often help drive flagging decisions.

Following this rule of thumb is sometimes a balancing act. We usually have the most context about an operation at the edge—where the operation enters our system. In modern systems, however, core business logic is often broken apart into many small services. As a result, we need to make a flagging decision near the edge of the system where a request first arrives (i.e., in a web tier) and then pass that flagging decision on to core services when making the API calls that will fulfill that request.

CHAPTER 8

# Best Practice #6: Scope Each Flag to a Single Team

In our previous example, acmeshopping.com was experimenting with offering free shipping for some orders via a feature-flag managing rollout. Marketing executives want to promote this new feature heavily and plan to put a banner at the top of the home page. Of course, we don't want to show that banner to users who won't be eligible for free shipping—we could end up with some grumpy customers that way! We need to place that banner behind a feature flag.

We could create a new feature flag called "free shipping banner," and use that to manage the display of the banner ad. But we'd need to make sure that this banner wasn't ever on when the other "free shipping" feature was off; otherwise, we'd be back to grumpy customers unhappy that they will not be getting the free shipping.

To avoid this problem, some feature-flagging systems allow you to "link" different flags together, marking one flag as a dependency of the other. However, most of the time there's a much simpler solution: just use one flag to control both the feature and the banner promoting the feature!

Although this might seem like an obvious best practice, it's sometimes not so obvious when flags are grouped by team, and the team implementing the shipping calculation code is very disconnected from the team implementing the banner ad. Ideally, your product delivery teams are already oriented around product features (i.e., the product detail page team, the search team, the home page team)

rather than technology or projects (i.e., the frontend team, the performance optimization team). This reduces the number of features that require cross-team collaboration, but does not eliminate it. Some features will always require changes across multiple teams. When that happens, it's OK to bend the rules and have a flag that's being used by multiple teams. You should, however, still always aim to have each flag owned by a clearly identified team. That team is responsible for driving rollout of the feature, monitoring performance, and so on.

CHAPTER 9

# Best Practice #7: Consider Testability

We write unit tests to validate parts of business logic, automate user interaction to maintain an error-free user experience, and avoid introducing bugs as the codebase evolves. In this section, we discuss the implications of using feature flags in combination with known practices of continuous integration.

When using feature flags in your application, you are creating a conditional alternative execution path. Imagine that you have a typical three-tier microservice composed of an API layer, a controller layer, and a Data Access Layer (DAL). Your engineering team set up a feature flag at the controller layer to gate the access to a new storage layer that won't have any visible impact for the user.

In a typical CI/CD environment, a new code change is pushed to the source code repository, where it is compiled and all appropriate test cases are run (unit tests, integration tests, end-to-end tests, and so on). If successful, the new code will become part of the main branch.

There are two approaches here. On one end we have high-level testing, often called end-to-end testing or black-box testing. This approach tests the functionality from the outermost layer and ensures that the expected behavior is working without caring for how the underlying components operate. When using feature flags, high-level testing must assure that the application will produce the expected behavior when the feature is turned on.

On the other hand, when pursuing lower-level unit testing, you should try to isolate the functionality the flag is gating and write tests to target both behaviors; testing the functionality for when the flag is on and when the flag is off.

Last, it's easy to fall into the temptation of writing tests for all possible combinations of flags. We advise reducing the scope of the test components to span only a handful of flags or isolate the test so that the tester can test the main target flag with the rest of the flags turned on.

CHAPTER 10

# Best Practice #8: Have a Plan for Working with Flags at Scale

As engineers from different teams create, change, and roll out feature flags across the application stack, tracking and cleanup can get out of hand over time. Maintaining the who/what/why of feature flags and establishing a basic process for identification and tracking will reduce complexity down the road.

## Naming Your Flags

Defining a pattern or naming convention for naming your feature flags is always a good practice for keeping things organized. Your peers will quickly identify from the name what the feature is about and be able to recognize what areas of the application for which the feature is being used. We won't take an opinionated view on a specific naming convention; instead, we point out useful examples of naming structures and demonstrate the practical benefits of adopting one.

The feature name example that follows has three parts. First, we present the name of the section the feature is gating. In this example, the feature is gating functionality in the admin section. The second part indicates what the feature does, a self-explanatory naming: make the new invite flow visible to users. And, last, where in the stack the feature is located: here it belongs to the backend layer of the application.

This naming pattern looks like this:

```
section_featurepurpose_layer
```

Here's a specific example that uses the aforementioned template:

```
admin_panel_new_invite_flow_back_end
```

Here are a couple of other examples of the same feature but in different layers of the stack:

```
admin_panel_new_invite_flow_front_end
admin_panel_new_invite_flow_batch
```

An alternative naming structure can include the team that created and owns the flag; for instance, Data Science, Growth, Data Infrastructure. You might also want to include the name of the service for which the flag is used: web app, router, data writer, and so on.

We recommend adopting a naming convention that makes sense within the organization's preestablished style code; or, if one doesn't exist, taking this opportunity to engage with different stakeholders of the feature-flagging system to define one.

# Managing Your Flags

Feature flags are a useful tool for folks in a variety of product delivery roles. It's not uncommon to see rapid, organic adoption of feature flagging after it's introduced into an organization, sometimes leading to a phase in which feature flags become a victim of their own success. The number of flags within a product can grow to become overwhelming. No one is entirely sure who's responsible for which flags or which flags are stale—left permanently on for the foreseeable future, cluttering up your codebase with conditional statements and contributing an incremental drag on delivery. Also, a misguided change to the flag configuration of your production systems can have severe negative consequences.

To ensure a successful long-term adoption, it's essential to have a plan in place that will keep the number of flags in check and help you manage the ones you have.

## Establishing a retirement plan

Delivery teams are asked to add flags for a variety of reasons—release management, experimentation, operational control—but aren't often asked to remove a flag after it has served its purpose.

Teams need to put processes in place to ensure that flags are eventually retired.

One useful technique is to add a flag retirement task to the team's work backlog whenever a flag is created. However, these tasks can have a nasty tendency of being perpetually deprioritized, always one or two weeks away from being tackled.

Assigning an exact expiry date to every flag when it is created can help break through this tendency to prioritize the urgent over the important. Your feature-flagging system should have some way to communicate this information, ideally with a highly visible warning if a flag has expired.

Some teams even go so far as creating "time bombs," whereby a system will refuse to boot if it notices any flags that have passed their expiry date. Less extreme versions of this approach would be for the system to complain loudly in logs or perhaps fail your CI build when an expired flag is detected.

You can also opt to place a limit on the number of active flags a given team has under management. This incentivizes the removal of old flags to make room for a new flag.

Of these various techniques, we recommend getting started by placing a limit on the number of active flags and ensuring that you always add a flag-removal task to your backlog whenever you create a flag.

## Finding zombie flags

You might be faced with a situation in which you already have a large number of flags in your system that don't have any of these retirement plans in place. A good feature-flagging system can help you by showing you flags that have been either 100% rolled out or 0% rolled out for an extended period of time, or flags that haven't had their configuration modified in a long time. You could even have your flagging system identify flags that are in your flagging system but your production systems aren't using.

## Ownership and change control

As an organization increases its adoption of feature flagging, the usage patterns for flagging tend to broaden. A system that was initially used primarily by engineers begins to be used by product own-

ers, marketers, production operations folks, and others. At the same time, the number of product delivery teams using feature flagging can also grow. Initially, a flagging system might have been used by only one team or product, but over time the number of teams taking advantage of the capability can grow.

As this spread of adoption plays out, it becomes important to be able to track who is responsible for each flag in your system. In the early days, this was easy—it was always the tech lead for the mobile team, or always the product owner for the shopping cart team—and so many homegrown feature-flagging systems don't initially have this capability.

Having the ability to assign ownership of a flag—either to a specific individual or to a team—allows your flagging system to answer questions like "which flags my team owns and are within one week of expiring," or "which teams have active flags that have been at 100% for more than a month."

Feature flags offer an extremely powerful way to directly modify the behavior of production systems; flag configuration should have the same type of change control as production code deployments. Having explicit ownership associated with a flag also allows you to apply change control policies. For example, you can prevent individuals from making a change to the configuration of a flag that they don't own, or require approval before a feature-flag configuration change is made in production.

## Flagging metadata

Attaching information like expiration dates and ownership to a flag are specific examples of a more general capability: the ability to associate *metadata* to your flag.

A handy generalized form of this capability is *tagging* (also sometimes referred to as *labeling*)—the ability to add arbitrary key:value pairs to a feature flag. This idea is used to great effect by other infrastructure systems such as Kubernetes and Amazon Web Services APIs. It is an extremely flexible way to allow users of the system to attach semantic metadata to entities in the system. You can use tags to track the creation date and when it should expire, and to indicate the life-cycle status of a flag, what type of flag it is, which team owns it, who should be able to modify it, which area of the architecture it affects, and so on.

CHAPTER 11

# Best Practice #9: Build a Feedback Loop

Feature flags allow us to make controlled changes to our system. We can then observe the impact of these changes and make adjustments as necessary. If a new feature causes business growth metrics such as conversion rates to increase by 20% (with statistical significance), we keep the change and roll it out to our entire user base. Conversely, if a new feature is causing an engineering metric such as request latency to spike by 200%, we want to roll the change back—quickly! Put another way, when working with feature flags we operate within a feedback loop. We make changes, observe the effects, and use those observations to decide what change to make next, as illustrated in Figure 11-1.

We cannot overstate how effective a mature feedback mechanism is in unlocking the maximum value of a feature-flagging practice. Making a change without being able to see the effect of that change easily is like driving a car with a fogged-up windshield.

Despite the value of this feedback loop, a surprising number of feature-flagging implementations start life with no or very limited integration to the analytics and instrumentation systems that exist in most modern product delivery organizations and provide a rich mechanism for feedback and iteration.

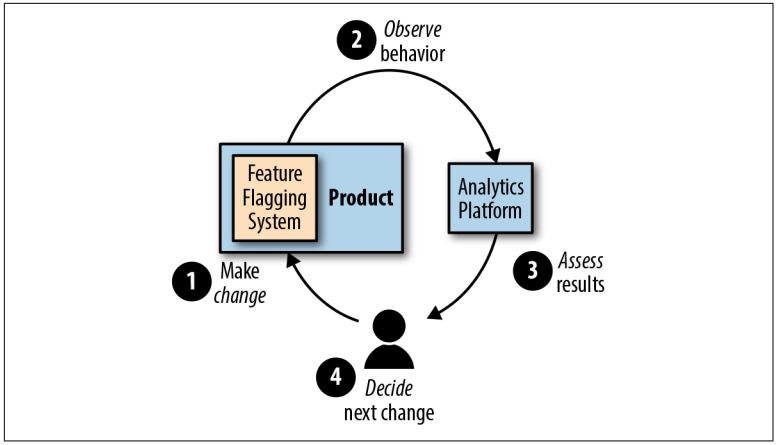


*Figure 11-1. Feedback loop for feature iteration*

# Correlating Changes with Effects

To observe the effects of a feature-flag change, we need to be able to correlate the change with its effects, closing the feedback loop. Correlation is the key. If we apply an A/B test and see 50% of users converting more but don't know which users saw which treatment, we are unable to make sound decisions. Continuous Delivery essentially produces micro-launches all the time as engineers constantly push new features into production. The impact of small changes is difficult to measure if the metrics are not directly tied to the feature.

You can correlate the impact of changes made by tying a measurement to the feature flag or by pushing information about the live state of the feature flags into the instrumentation and analytics systems. This information can be either inferred or statistically analyzed to determine causality.

## Inference

If we publish changes to our feature-flagging configuration as a separate stream of instrumentation events, we can use the timing of these changes as a way to correlate a feature-flag change with its effect. For example, if we see 10% of servers having an increase in CPU utilization at the same time as a feature flag change that rolled out to 10% of servers, we can pretty easily infer a correlation.

This approach enables correlation in most simple scenarios but has some drawbacks. In the case of a 50% feature roll out, it will be difficult to know which effect is caused by the flag being on and which by it being off. It's also more difficult to draw correlations when the impact of a feature flag takes some time; for example, a change that causes a slow memory leak, or a change that affects a user's behavior in later stages of a conversion funnel. The fundamental issue is that we're inferring the correlation between a feature change and its effects.

### Causality

A more sophisticated approach is to include contextual information about feature-flag state within an analytics event. This approach, most commonly described as experimentation, ties metrics to a feature flag to measure the specific impact of each change. For example, the analytics event that reports whether a user clicked a button can also include metadata about the current state of your feature flags. This allows a much richer correlation, as you can easily segment your analytics based on flag state to detect significant changes in behavior. Conversely, given some change in behavior, you can look for any statistically significant correlation to the state of flags. Experimentation establishes a feedback loop from the end user back to development and test teams for feature iteration and quality control.

## Categories of Feedback

You might already have noticed from the discussion so far that a feature change can have a broad variety of effects. We might observe an impact on technical metrics like CPU usage, memory utilization, or request latency. Alternatively, we might also be looking for changes in higher-level business metrics like bounce rate, conversion rate, or average order value.

These two categories of metrics might initially seem unrelated—and they're almost always tracked in different places, by different systems—but when it comes to feedback from a feature change, we are interested in analyzing change from as many places as possible. It's not unheard of for a change in a technical flag to have a surprise impact on business KPIs, or for a new user-facing feature to cause issues in things like CPU load or database query volume.

Ideally, we will look across both categories when looking for the impact of a feature change. One thing to note when looking to correlate a feature change with its effects is that business analytics is typically oriented around the user, at least for a typical B2C product, whereas technical metrics usually focus on things like servers and processes. You'll likely want to use different contextual information for these different categories of feedback.

# CHAPTER 12
# Summary

Feature flags help modern product delivery teams reduce risk by separating code deploy from feature release to safely increase release velocity at scale. Feature flags provide a mechanism for feedback and iteration by linking features to changes in engineering KPIs and product metrics.

In this book, we've offered advanced users of feature flags some best practices for working with feature flags. Following tips such as maintaining flag consistency in different scenarios, development and testing with feature flags, and working with feature flags at scale, will help you to manage a growing feature-flag practice.

## About the Authors

**Pete Hodgson** is an independent software delivery consultant based in the San Francisco Bay area. He specializes in helping startup engineering teams discover how to deliver maintainable software at a rapid but sustainable pace, by leveling up their engineering practices and technical architecture. Pete previously spent several years as a consultant with ThoughtWorks, leading technical practices for their West Coast business, in addition to several stints as a tech lead at various San Francisco startups.

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