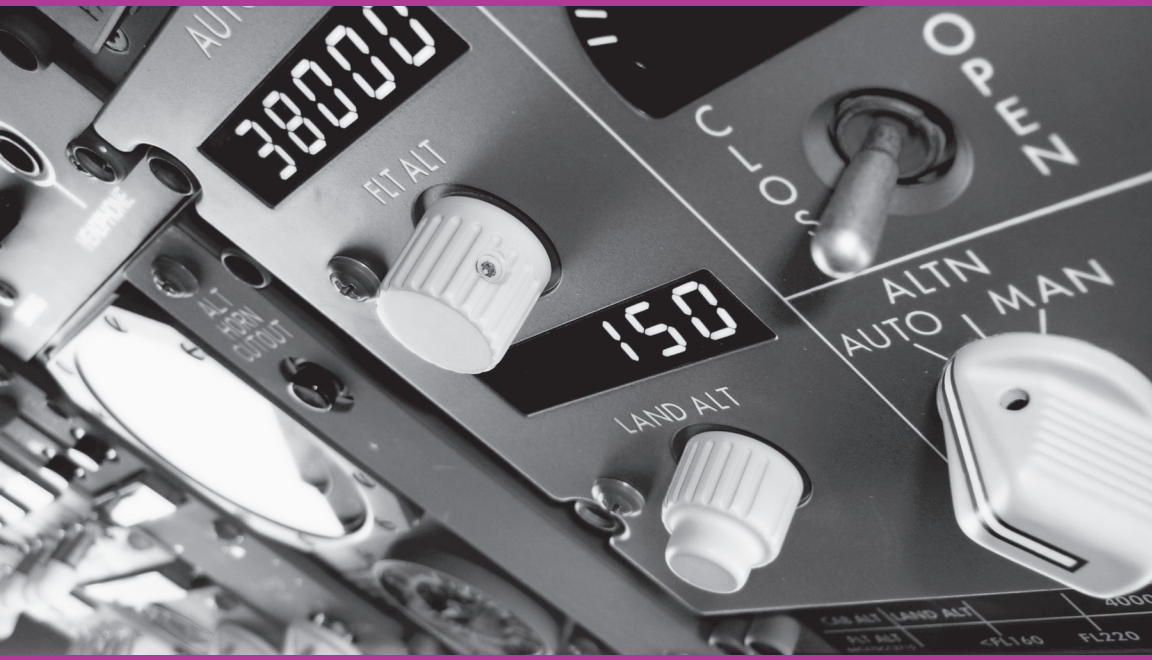
O'REILLY®

Compliments of LaunchDarkly

Effective Feature Management

Releasing and Operating Software in the Age of Continuous Delivery

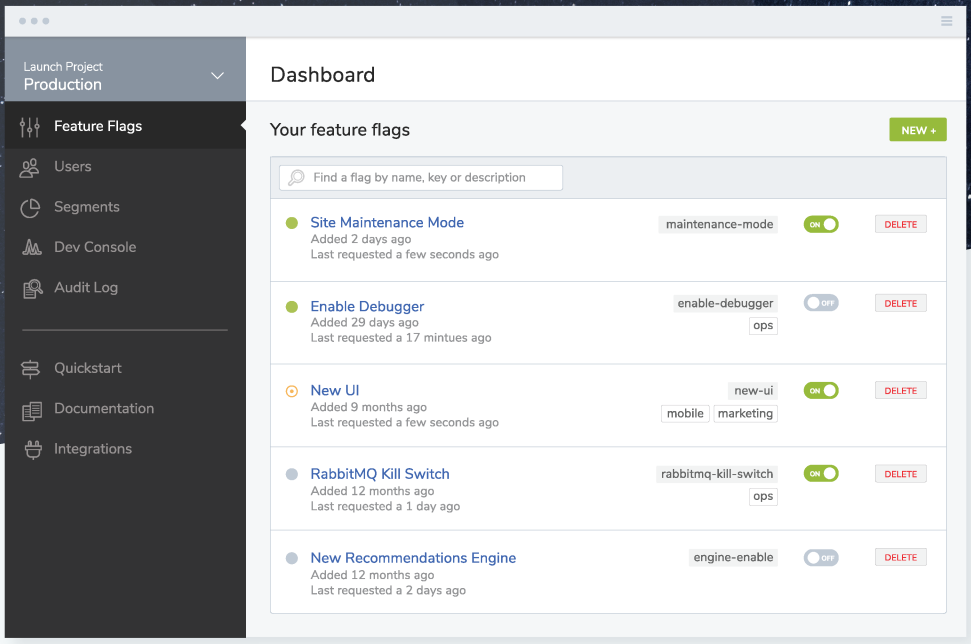


John Kodumal

Deploy More, Risk Less with LaunchDarkly Feature Management

✓ Build better software
✓ Deploy more often
✓ Release with less risk
✓ Operate reliable services

Learn more at go.launchdarkly.com/oreilly



The promise of continuous delivery is now delivered in the languages and frameworks you love.

Java .NET JS php iOS

LaunchDarkly

Effective Feature Management

Releasing and Operating Software in the Age of Continuous Delivery

John Kodumal

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

Effective Feature Management
by John Kodumal

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: Justin Billing
Copyeditor: Octal Publishing, LLC
Proofreader: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

December 2018: First Edition

Revision History for the First Edition
2018-12-21: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492038160* for release details.

978-1-492-03814-6

[LSI]

Table of Contents

Introduction

Since the Agile revolution in software development, teams have been pushing to move faster and faster every year. Over the past decade, the discipline of *continuous delivery* has helped software development teams increase their rate of delivery from months to minutes. Accelerating the pace of change results in huge advantages to both development teams and businesses: making frequent changes, observing feedback, and reacting quickly is a competitive edge, and every business is racing to gain the inside lane on their competitors.

Software development professionals need to recognize this shift and appreciate that achieving true *continuous integration* and continuous delivery (CI/CD) puts software development teams in a fundamentally different world. This new world brings with it enormous benefits but also new challenges. Delivering more changes more frequently can also introduce increased chaos and risk, especially when our toolkits for controlling risk have not adapted as quickly to this new world.

The tools and practices software developers have created to release slowly and safely don't translate to a world in which dozens of changes are released each day. In fact, the very practices that were originally so sensible can become actively harmful.

Software developers need a new set of tools that are specifically designed for a world with continuous delivery. What if you could fully decouple the act of delivering software from the act of releasing features? What if you could deploy any time you want but release only when you're ready?

Imagine how development practices and software releases could evolve in a world in which code deploys are decoupled from making that code available to end users:

- Imagine that *every* deployment is deliberately, safely ramped up, allowing you to react instantly to the unexpected.
- Imagine that you could *know* how each new feature will perform in production—with real users and real data.
- Imagine being able to instantly turn off a misbehaving feature, protecting the rest of your business.
- Imagine measuring the impact of each and every change so that you know how it will affect users and your business before you proceed.

In this book, I define a new category of software called *feature management* and discuss the underlying technologies, use cases, and best practices for adoption. I also include real-world case studies throughout. Feature management enables businesses to dynamically control the availability of application features to end users. Product teams use feature management to progressively deliver new code and features to end users, run experiments and A/B tests, customize the user experience, and maintain highly reliable applications—all while the application is running and without the need to deploy new code. Wrapping each change in a feature flag and controlling it with a feature management service empowers a variety of people across the organization—from development and operations engineers to product and marketing managers—to move faster with less risk, while delivering better results for your customers.

CHAPTER 1

Feature Management

Feature management solutions enable businesses to dynamically control the availability of application features to end users. In this chapter, I introduce *feature flags*, a fundamental technology in feature management solutions. I then discuss the various feature management use cases, including progressive delivery, infrastructure migrations, experimentation, and testing.

What Is a Feature Flag?

In simple terms, a feature flag is a decision point in your code that can change the behavior of your application. Feature flags—also known as *feature toggles*—have long been used to trigger "hidden" code or behaviors without having to ship new versions of your software.

A feature flag is like a powerful "if" statement:

```
if(enableFeature(one.click.checkout, {...}))
then
    /*show the one-click checkout feature */
else
    /* show the old feature/
```

In the early days of software development, this might have been a command-line argument, an undocumented variable in a configuration file, or a hidden value in a registry. The value of each feature flag was typically set early in the life cycle, at compile-time, deploy-time, or runtime. After the value of the flag was set, the software

would run that way until it was recompiled, redeployed, or restarted. Modern feature management is different.

There are two parts to the seemingly simple `enableFeature` call highlighted in the preceding example that make it special. The second parameter to the function (elided for simplicity in our code snippet) is a *context*. It typically identifies a specific user, but it can contain any arbitrary data. This is important because it makes the behavior of the function *context sensitive*. The code path taken can change based on the context provided; for example, the user's identity, the plan they've paid for, or any other data.

Second, the `enableFeature` function calls out to the feature management platform, which takes the context and a set of rules defined in the feature management system to determine which code path is executed. This makes the behavior of the function *dynamic*—the code path can change whenever the rollout rules are changed, without redeploying a new version of the software.

Feature flags can either be temporary or permanent. Temporary flags are often used to safely deploy changes to your application or to test new behaviors against old ones. After a new behavior is being used by 100% of your users, the flag is intended to be removed. Permanent flags give you a way to control the behavior of your application at any time. You might use a permanent flag to create a kill-switch or to reveal functionality only to specific users.

This simple function is the source of enormous benefit for your team. After you begin to build feature management into your application, you'll develop new superpowers to ship when you're ready, test in production, stay in control, experiment with everything, and customize on the fly.

Ship When You're Ready

With the rise of continuous delivery, software teams ship new code to production all the time. But the rapid pace of change brings new kinds of risk. In the old days, there might have been months of testing and validation for each new deployment. But for a team practicing continuous delivery, new code goes from the development team, through automated tests, and then directly to users in minutes.

Deploy != Release

Software developers attempt to mitigate this risk in two ways. First, when deploying frequently, each change is less risky because it is smaller and more isolated. Secondly, feature flags can be used to control which users can see each change. This decouples the act of deploying from the act of releasing.

When deployment is decoupled from release, feature flags give you the opportunity to create a release strategy. Your team—developers, product managers, DevOps, and marketers—can decide together, at the beginning of the project, how a new change should actually roll out to customers:

- How will the feature be released?
- Should anyone see it now?
- Who will be getting this feature first?
- Who will beta test it?
- Will it be rolled out progressively?
- Do I need to compare it against the old behavior?
- Do I need to hit a particular date for an external event?
- What do I do if something doesn't go right?

Feature flags allow your team to create an agreed-upon process for deployment with success criteria that you can define at each stage. The flags give you the ability to move forward, or backward, based on the success at each critical decision point.

Teams using feature management are no longer constrained by users seeing new code the moment it hits production. Developers can merge their work, even in a partially finished state, deploy it to production, and be confident that no users will experience it until they turn on the feature flag. This also means that you can do *true* continuous delivery, where new code is deployed as it is written. Without some way to decouple deployment from release, teams are stuck in a maze of long-running feature branches.

With feature management, not only are developers able to fully adopt continuous delivery, but the business can now take advantage of *progressive delivery*—the ability to strategically control when individual users or user segments gain access to new features. Develop-

ers can build at the pace of innovation, operations can deliver continuously, product teams can iterate quickly, and marketing can safely release features to the right users at the right time.

Case Study: Progressive Delivery

Atlassian, an enterprise software company (and former employer of mine), planned to launch significant rebranding across its entire product portfolio at its annual user conference. There were dozens of different teams responsible for multiple products and websites that all had to make the change at exactly the same time for maximum effect.

Over the months leading up to the conference, feature and branding changes were built into the company's products but safely hidden with a master feature flag.

While their CEO was on stage during the conference keynote, the team simply switched the master flag, revealing the new branding across all the products and properties at the same time. The changes had already been deployed and tested—the switch merely activated existing code that had been in place for weeks.

This software provider had a flawless rollout of its new brand. The marketing team pulled off a dramatic reveal at the conference. The development team didn't need to push a massive number of coordinating code changes on a tight timeline. Feature management enabled development teams to deploy as soon as the code was ready, and left the moment of release up to the marketing team.

Test in Production

Test in production started out as a joke in the days of Waterfall development because software development schedules almost always slipped. Testing, which always came at the end of the life cycle, was often compressed or skipped entirely. To meet the deadline, teams often sarcastically joked that testing could happen after the feature was released in production.

It sounds risky to test in production; you don't want to expose your users to untested software. That's why software teams have built elaborate patterns for testing and why most developers assume that deployment must always involve difficult, multistage deployments to

testing and staging servers. But there are surprising benefits to skipping the staging servers and testing in production deliberately and safely.

True Test Environments Are Difficult to Create

The software industry has moved toward distributed microservices and an ever-expanding roster of external service providers. The volume of messages, transactions, traffic, and data has increased significantly, to the point that it is often impossible to replicate in another environment.

Unlike 10 years ago, your software can't be tested realistically on a developer's laptop. And even your testing and staging environments are unlikely to be true representations of your service.

Real users are unpredictable and so is their data. Your test data, even at high volume, will never capture every edge case. And the closer you come to production-level traffic volumes, the more expensive testing becomes, especially for projects that rely on external vendors like Amazon Web Services.

So, kill your staging server! Test in production.

Test in Production

Testing with real, live users on your production environment will always yield better data than any test or staging system. By testing in production, you can gain a much more accurate understanding of your system's behavior.

The question is how to do this testing safely. With a feature management platform, teams use powerful targeting rules to control access to new features and can instantly turn access to any feature off, bringing back the old behavior instantly, without the need to rollback code or redeploy.

Canary releases, *ring deployments*, and *percentage rollouts* are different patterns for safely testing in production by progressively increasing the exposure of any new change. Instead of cutting over

completely, as in a *blue/green deployment*,[1] feature management allows you to roll out a change gradually to an increasing population, monitoring as you go.

How Netflix Tests in Production

Netflix provides content to millions of consumers each day. Because their system is so large and their users interact with it in so many different ways, it is nearly impossible to create a test environment that would yield reliable data. So engineers are not just allowed but encouraged to test new code in production instead of in an artificial staging environment.

This organization has made it safe for its engineers to test on real users by using feature management by default. It uses percentage rollouts, starting with tiny fractions of the user base, to observe the effects of every change. Netflix has strong guidelines on how tests are set up—including which users are exposed to a test at any given time—and are always monitoring so that it can stop experiments before anything has a significant impact on customers.

By testing in production, this team can quickly find and fix problems, without affecting all customers, and still maintain a high level of service.

Percentage Deployments

In a percentage-based rollout, small numbers of users are selected randomly to experience the new feature. That percentage is then increased over time until eventually everyone has access to the new feature. By starting small and gradually increasing the number of users, you have the opportunity to observe the behavior of the system under new conditions, and advance only if the signs are healthy and user feedback is positive.

1 Blue/green deployment is a technique that includes running two identical production environments. Only one of the environments serves production traffic while the other environment is updated with the new software. When ready, the updated environment takes on all production traffic, and the original environment is available for rollback if necessary.

Percentage-based rollouts are useful when there is little variation in your targeted user base or you are more concerned with operational impact of your change.

Companies often automate percentage rollouts to gradually increase unless there is an alert or monitoring alarm.

Case Study: Percentage Deployment

A large social media company wanted to use a new third-party service to provide images and profile information to enhance its users' contacts. However, the application had hundreds of thousands of users, each of whom had hundreds or thousands of unique contacts in the address books they had saved in the system.

The team wanted to ensure that it had enough capacity to successfully cache the information from the third party and to ensure that these new requests didn't adversely affect the rest of the system.

The team started with a group of beta customers that were selected based on geography, which allowed it to see real-world behavior but in small volumes. Through the beta test, the team found ways to refine the caching system to be less expensive.

After the successful beta test, the team deployed the new functionality to its customers in increments of 10% over the next several days, monitoring how the new system performed in the new configuration as usage increased.

Having this level of control over the way in which a new feature was delivered allowed the team to beta test, validate new functionality, and roll out significant new functionality in a way that controlled risk and gave more flexibility to the team.

Ring Deployments

Ring deployments, a term coined by Microsoft, is another method to gradually expose features to different groups of users. The difference is that the groups are selected specifically to manage the risk of deployment. Microsoft deployments often start with small groups of low-risk users and expand through larger, higher-risk populations. This helps identify problems early while limiting the "blast radius" of disruption if something goes wrong.

A typical ring deployment begins by releasing first to internal users. After you have verified that the change has been successful for that set of users, you can expose the next set of users: the *canary* group.

Teams use canaries to measure the reaction from real users in production and look for early indicators of danger or success (similar to how actual canaries were used in coal mines many years ago to test for toxic gas). Even testing with a small number of users can increase your confidence dramatically. If a feature generates an unexpected or negative reaction, you can pause the deployment to address problems or even turn the feature off entirely until you are ready to try again.

Then, you can move on to the third ring: beta testers or early adopters. These are users who are interested in having early access to new features and are more prepared for problems or rough edges.

After you're satisfied with your beta test, you can move on to a general release. But even this you can do in stages. You might start with users of your free product and then move on to paying customers. Or, you might start with small geographies and move on to larger ones. Or, you might start with accounts with small numbers of users or load, and expand to larger-load customers.

Not every ring deployment needs to follow this exact pattern. But the defining characteristic of a ring deployment is using feature management targeting rules to release strategically to users in order to identify problems in the smallest, lowest-risk population.

Case Study: Ring Deployments

A team at a revenue intelligence company developed the pattern of releasing new features first to a subset of its best customers. Engaged and happy customers who had the chance to see new features first felt like they were special and in-the-know, and they provided great feedback and were the most vocal in spreading the word to other users. This canary group of super users wasn't based on the users who deliberately opted in to testing; rather, it was based on users who were using and enjoying the product the most.

This company created its canary group by integrating its Net Promoter Score (NPS) survey solution with its feature management system. The company automatically added NPS promoters into the group. This also meant that if a customer's NPS changed, that cus-

tomer could be added or dropped from the canary group without intervention.

Stay in Control

Creating and deploying new software is risky. Bugs can be introduced accidentally, the software can be delivered badly or to the wrong people, or it can interact in unexpected and unfortunate ways with existing software or hardware. It's important to have safety nets in place when things don't go as planned. Feature management provides the necessary control to reduce, and often eliminate, the risks associated with deploying new software.

Flag Early and Often

I've found that many teams have a too-narrow definition of what they can feature flag. The misconception comes from thinking that a "feature" refers only to customer-visible features. In fact, there are many benefits to feature flagging every significant change, even those that are not visible to the customer, such as new backend improvements or infrastructure changes. As a complement to customer-visible feature flags, these "operational feature flags" give DevOps teams powerful controls that they can use to improve availability and mitigate risk.

Safety Valves and Circuit Breakers

A well-wrapped feature means that you can quickly turn it off if it is performing poorly, which can mean the difference between a public relations disaster and a minor hiccup with minimal impact. Enter the *safety valve*.

Safety valves are permanent feature flags that you can use to quickly disable or limit nonessential parts of your application to keep the rest of the application healthy. On an ecommerce site, for example, increased page load time can dramatically reduce conversion rates. If nonessential features like product recommendations or reviews are found to increase latency, a DevOps team can use a safety valve to quickly disable the feature and reduce latency, all without requiring any change in code.

Safety valves are powerful tools that you can design into a system from the outset, or introduce later, after a need has been identified. It's important to properly document these flags. Because they're permanent, you must note them as such so that they're not accidentally removed. You also should add them to service runbooks, with a clear description of the symptoms that should trigger the use of the flag as well as the impact to end users. Going a step further, if a safety valve disables user-visible functionality, you should bake it into incident management processes so that status page updates, emails to impacted customers, and other relevant communications are triggered.

Simple safety valves are first managed manually, but the next logical step is to automate them, triggering changes to the flag's rollout rules based on error monitoring or other metrics. This kind of automation is powerful, but you must use it with caution—a false positive caused by faulty error monitoring could cause unnecessary customer impact.

Case Study: Kill Switch

A consumer retail company with a large online presence experienced a complete outage the morning of Black Friday. When the site tried to restore automatically, all the fresh requests for pages caused the servers to overload again, resulting in hours of downtime and millions of dollars in lost revenue.

After the company finally was able to restore service, the team realized that it could have restored access much more quickly if it had been able to turn off noncritical features. The more resource-intensive parts of the team's page weren't necessary for customers to check out, and serving a less feature-rich version of the site would have still allowed the customers to purchase.

Atlassian uses feature flags heavily in its development processes. With a combination of release flags and safety valves, it is able to recover from 90% of its incidents within seconds. Only a small fraction of incidents require redeploys or code reverts.

Infrastructure Migrations

Infrastructure migrations, like database schema changes, are some of the riskiest changes in software systems. They often involve complex interactions between different services and backing stores. They can be difficult to roll back, and in modern applications, they must usually be completed with no downtime.

With some advanced planning, you can use feature flags to remove the risk from most infrastructure migrations. The key idea is to decompose the problem into multiple steps, with each step controlled by a feature flag that you can roll back. At each step, you use correctness checks to validate that it's safe to proceed to the next step. This approach works well for database migrations, but you also can apply it to other infrastructure changes (e.g., changing cloud service providers, vertically scaling a database, or swapping the implementation of a specific algorithm).

Case Study: Infrastructure Migration

LaunchDarkly relied heavily on Elasticsearch. When the version it was using was deprecated, the company needed to switch to the latest version immediately.

LaunchDarkly used feature flags to complete this transition quickly and safely. Engineers created new classes for the new search syntax and then put the new classes behind feature flags.

When the DevOps team started the Elasticsearch upgrade, it deployed the upgrade to production on an index-by-index basis. Throughout the rollout, the team was able to test the upgraded search—with production data—and ensure that search results were correct.

Using feature management, even for a "feature" that was invisible to customers, gave the team control over the rollout process and a safe space to test. This meant that any issues would be discovered early, and the team could halt the migration and address those issues before continuing with the upgrade for all indexes.

Experiment with Everything

Feature flags can serve different features or different feature versions to different people, based on rules that you create. This is a powerful new tool that allows you to experiment with different behavior in your applications. You can perform extremely detailed and granular targeting of your users based on their self-reported characteristics, historical usage patterns, or any attribute you specify.

By defining a goal and comparing which behavior in your application leads to more successful completions of that goal, you can validate your hypotheses about product improvements and protect yourself against unexpected changes.

Monitoring/Baselines

Every change that you make to your software is a kind of experiment, even if you don't explicitly design them as such. When you release a new change using a feature flag, you have a built-in capability to measure the impact of the change.

You begin by establishing important baselines for your product, such as the number of signups, dollars in sales, or searches performed. Then, as you progressively roll out, you can compare user activity in the old cohort with the behavior in the new one to ensure you are affecting your baseline metrics positively. Your application is a complex system, and it isn't always easy to predict the way users will react to changes, even changes that might seem unrelated. Measuring your new features against existing baselines helps to reduce the risk of change.

Case Study: Optimizing and Measuring the User Experience

Atlassian has more than a dozen products in its portfolio and wanted to roll out a large-scale UI change for one of its most popular products. Though the team was excited to deliver a new and improved UI, it was also sensitive to how its customers would receive the changes. The team wanted to carefully control how it rolled out new features to users, but it wanted to test new functionality with real customers before releasing to its entire user base.

Using a feature management platform with custom targeting, the team identified its most frequent and demanding users. It was able to target these users specifically to test its new UI and confirm that it was developing new features these customers would find valuable. The team compared the behavior of power users in using the new and the old UI, looked at how often.

When it was time to roll out functionality to the entire user base, the team was careful to treat various cohorts differently. New customers saw only the new UI. Existing customers, however, received different treatment. Power users saw the new functionality first, and less technical users were given more time to adjust. Using powerful targeting rules, they were careful to give all users within the same account the same experience to minimize confusion.

A/B Testing

A/B testing, or experimentation, is a commonly used method of validating new ideas. This kind of experiment tests one thing against another—variation A versus variation B, or new versus old. Multivariate feature flags allow you to serve multiple variations of a feature to different user segments: testing variation A versus B versus C, or more. Most people use experimentation to test UI features, such as placement of buttons, design layout, or word choice. But it's equally useful for testing backend functionality, such as evaluating the effect of different content delivery networks (CDNs) or different site search providers.

User Feedback

The ability to release changes to a limited set of users makes it much easier to gather feedback about your product. You can create a beta group of users and target feature flags specifically to that group who have offered to provide you feedback. Testing new features with a subset of users allows developers to find and address bugs as well as glean valuable feedback about the features they've built.

As your team continuously delivers new features of your application, feedback is continuously flowing back to you. You have the

opportunity to harvest this feedback and develop your product to meet the needs of real users.

Case Study: User Feedback

Honeycomb provides enterprise customers with powerful debugging, monitoring, and observability tools.

The company's process for adding new customers to beta programs was cumbersome and slow. After an interested customer was identified, the product team had to make a formal request to the development team to change code in order to give the identified customer access to the beta. Only after the request was approved could the code be committed and deployed. Depending on how busy the team was, the customer might need to wait days or weeks.

Besides being a long, unpredictable process that delayed getting customer feedback, this back-and-forth also created unnecessary work for the development team. The team was already using home-grown feature flags, but there was no way for nontechnical team members to make changes to features.

The team replaced its home-grown feature flag system with a feature management platform and employed user-specific targeting to grant access to beta features. Now, any product manager can enable beta features for a specific customer in a matter of seconds without the need to wait for a developer to change and deploy code.

Customize on the Fly

Customizing how users experience your application can be a powerful way to improve engagement and increase conversions, sales, and other business objectives. Historically, customization has required significant investments in developer resources. With feature management, nontechnical teams can now manage customization initiatives without requiring direct support from developer resources.

Entitlements and Plan Management

There are many kinds of features that are better controlled with a permanent flag. Often, features in your code might need to change because the status of a user or organization changes; for example, when a user upgrades from a free to a premium service or when a

specific organization needs a custom feature. In many cases, these changes can and should be managed by sales and support, not the development organization.

Case Study: Customization with Feature Flags

A logistics and fulfillment company created its own system to turn features on and off, but only developers could access it. Every time support needed to change a user's account status, the support engineer needed to file a request with the development team and wait, sometimes days.

Customers were frustrated that support could not help them immediately, support felt like it didn't have the power to do its job, and development was irritated because its engineers felt like support kept bugging them about trivial yet time-consuming changes.

This organization decided that a feature management system that all teams could use would alleviate frustrations on all sides. The team designed a system with access controls and audit logs that the development team trusted other teams to use independently. Now, support, sales, and marketing all have the ability to adjust product access as needed for each customer, and development can focus on building what's next.

Dynamic Configuration Management

Configuration parameters might need to be changed over time. Adding feature flags to those settings allows you to change the way your software operates without having to update or redeploy code. This results in much quicker response times to changing conditions.

Case Study: Dynamic Configuration

A home automation company had thousands of voice-activated internet of things (IoT) devices distributed in customers' homes. The company began getting reports from customers that the devices were being triggered by television content. The devices would hear the activation word from TV dialog and misinterpret the command as coming from the homeowners.

It would have taken the company weeks to push full firmware or software updates to the large number of IoT devices distributed

across many geographies in environments the provider doesn't control. Luckily, the team had built in a feature flag to its software that could disable that particular initiation word. The change took a matter of seconds and did not require the devices to reboot or install updates, and the support line stopped receiving calls about phantom activation.

Summary

There are a many uses for feature management that benefit different departments in an organization. Your feature management solution will empower your software development, technical operations, and business teams to deliver value to customers faster, with less risk. But how do you bring feature management to your team? In the next chapter, you'll find best practices for adopting feature management across your organization.

CHAPTER 2

Advice from the Front Lines

After your team has decided to take advantage of feature management, it's easy to get started in small, simple ways. There are certain techniques that you'll want to keep in mind as you expand your usage and incorporate feature management into your development process to ensure that you realize the maximum benefit.

First Steps with Feature Flags

One of the great things about feature flags is that the cost to try them is extremely low. There's no need to completely shift development methodologies to incorporate feature flags into your workflow. You can begin with a single flag, demonstrate its value, and introduce flags into your workflow slowly, use case by use case.

Your team can benefit from feature flags regardless of how far along you are in adopting Agile methods. You can use feature flags to release more frequently, even as you are working to modernize the rest of your development methodology. Feature flags don't require you to have a distributed version control system (DVCS), continuous integration and continuous delivery (CI/CD), Kubernetes, service meshes, containerization, serverless, or any other specific technology in place in advance. (Though flags work great in tandem with many of those!) You can begin with feature flags now, no matter what your architecture or deployment cadence is.

If your team is working on adopting modern development practices, feature management can act as a beachhead—a critical first step taken down that path.

Your First Feature Flag

A *kill switch* is a technique for quickly turning off a feature or routing users to a previous version of a feature. Kill switches are the simplest use case for feature flags and make for a great first flag in your software. I've found that teams derive immediate value by introducing kill switches around a new feature. It's also best to start with a small feature; your first flag shouldn't be part of a large, multimonth delivery effort. Find quick wins to begin building the muscle memory in your team for flagging.

It's also important to follow these first flags through the entire life cycle, including removing them after they're no longer needed. After this process has been completed successfully a few times, you can begin introducing feature flags into your everyday workflow.

Feature Flags in Your Workflow

Feature management is valuable to many teams in an organization, not just the development team. Depending on the need, features might be managed by product managers, marketing, support, or DevOps teams. When you're starting a new project, you should include feature management in the planning process. In the kickoff meeting, it's useful to ask the following questions to help define your flagging strategy:

- What's the release strategy for this feature? How risky is it? Are there segments of users who should see the feature first?

  Consider whether there should be a beta, canary release, or a ring deployment for this feature.

- Are you gating access to the feature based on pricing plans or entitlements?

  Consider adding a permanent flag and specifying rules to determine who should have access to the feature.

- Are there any circuit breakers or operational controls that you should flag? What will you do if something goes wrong?

Consider including operational flags to adjust the configuration of your new feature.

- Is this feature nonessential and possibly resource intensive?

  Consider adding a kill switch.

- What does success mean for this feature? Are there any experiments that you should perform, baselines you should measure, or metrics to track?

  Consider introducing a flag that compares new behavior against the current behavior of a control group.

Based on your answers, you can define the flags that the development team should introduce. You'll know which flags are intended to be temporary, exactly how long they should live, and whether any permanent flags will be introduced. It's much easier to do this early, before you've begun to build your new feature.

Flags and Branching Strategies

You can use feature flags with any branching strategy. They are an important step in moving toward trunk-based development because the ability to flag code and keep it hidden allows you to merge aggressively, even before the branched code is ready for release.

Feature flags also complement feature-branch–based development and DVCS. Teams that incorporate flags in conjunction with feature branches can eliminate the need for long-lived branches, along with the headaches associated with integrating long-lived branches back into master.

Testing with Feature Flags

There's no one-size-fits-all approach to testing with feature flags, mostly because there's no one-size-fits-all approach to testing in general. However, it's important to think through the impact of feature flags on your testing strategy. With planning, feature flags can work well with almost any approach.

Should I Test All Combinations of Flags?

It isn't necessary (or even possible) to test every combination of feature flags. Because each feature flag has at least two variations, you'd

have an exponential number of test cases to write. Testing each variation of a flag in isolation (using default values for the other flags) is usually enough, unless there's some known interaction between certain flags. But you should identify combinations of feature flags that are known to create user issues in test cases and guard against using them in your code.

Flags and Libraries

Another decision that affects testing is whether you should use feature flags in reusable library code. I think the answer is no—flags are an application-level concern, not a library concern. Let's illustrate this with a small example. Imagine you have a search library that is backed by Elasticsearch, with an API like the following:

```
function searchES(query, lastId) {
  ...
}
```

Imagine that you're feature-flagging between two different search systems, using a feature flag called `use.algolia`. You could make the feature flag the library's concern, which would require us to pass the user context to the library, like so:

```
function flaggedSearch(query, lastId, user) {
   if (enableFeature("use.algolia", user)) {
     return searchAlgolia(query, lastId);
   }
   else {
     return searchES(query, lastId);
   }
}
```

This approach has a few problems. First, it introduces into the library a dependency on the feature management system, making it potentially more difficult to reuse and test. Second, it ties the application's data model (the user context) into the library. The user context isn't relevant to search; it's relevant only to the feature flag. A better way to structure this is to move the feature flag to the uses of the search function and add a parameter to the search function indicating which engine to use:

```
enum SearchEngine {
  ALGOLIA,
  ES
}
function search(query, lastId, searchEngine) {
```

```
  if (searchEngine == ALGOLIA) {
    return searchAlgolia(query, lastId);
  }
  else (searchEngine == ES) {
    return searchES(query, lastId);
  }
}
```

Now, feature flagging is an application-level concern. You check the flag at the call sites to the search function:

```
 if (enableFeature("use.algolia", user)) {
   engine = ALGOLIA
  }
  else {
   engine = ES
  }
  results = search(query, lastId, engine);
```

Using this method, you've separated the flagging concern from search and avoided introducing dependencies on the feature management system and the application-level user object, which simplifies testing considerably.

Unit Testing

Even after structuring your code to keep feature flags in the application level, you're still going to need to unit-test in the presence of feature flags. The easiest way to write unit tests for code containing feature flags is to mock-out the feature management service. Mocking is a simple way to avoid interactions with the external feature management service, which is important for unit testing.

Integration Testing

In most cases, it's still best to eliminate external dependencies in end-to-end integration tests. You can accomplish this either by using a mock object library or by having the feature management service return default values or hardcoded values from a file or other local source.

If external dependencies are not an issue for your tests, and you want to modify feature flags for testing purposes, a feature management service with an API is a viable approach. The API should allow you to programmatically create flags and change rollout rules. Ide-

ally, it should allow you to also create and destroy environments so that tests can start from a clean slate with each run.

Managing Technical Debt

Temporary feature flags introduce technical debt. Old flags that should have been removed can litter code with conditional statements and prevent dead code from being removed. They also clutter your feature flag dashboard, making it more difficult to manage active flags.

Cleaning up flags aggressively is the key to preventing technical debt from building up. There's no royal road to flag cleanup, but there are some processes that make it manageable.

Appoint a Maintainer

The developer who introduces a flag should be responsible for cleaning it up. To enforce this, appoint a maintainer for the flag. This person is responsible for recording the purpose of the flag, ensuring that it has a well-defined rollout plan, and eventually cleaning it up.

Set Expiration Dates

When I create a flag, I usually have some idea of how long it's intended to remain in the code. It's common to have temporary flags that can't be removed quickly; for example, when running a weeks-long A/B test or creating a flag that hides a months-long work in progress. These flags are the most likely to slip through the cleanup cracks. A strategy here is to give feature flags expiration dates at the time of creation. When a flag reaches its expiration date, you can be notified that it's time to delete that flag. Some teams file issues to track this, some teams use tags or naming conventions, and some teams even create pull requests to remove the feature flags in advance.

Flag Removal Branches and Pull Requests

One of the biggest barriers to deleting temporary flags is remembering what the flag did, where it needs to be removed, and what dead code can be deleted as a result. Too many *stale flags* are a form of technical debt and an antipattern that you should avoid. Surpris-

ingly, this amnesia kicks in quickly—even if a developer circles back only a week later to delete a flag, it's time-consuming to remember the original context and locate all the places where the flag is referenced. One way to help overcome this challenge is to create a branch and pull request (PR) to remove the flag at the same time the flag is introduced. This extra branch or PR is kept unmerged until it is time to delete the flag. After the flag has served its purpose, cleanup consists of merging the PR, deploying the change, and deleting the (now-inactive) flag from the flag management dashboard.

Finding Flag References

Eventually, you'll run into a flag that nobody knows anything about. Perhaps the maintainer has left the company or the flag itself wasn't documented well. You might not even know which codebase (or codebases) reference the flag. In that situation, you'll need a good strategy for finding references to the flag in code. Some best practices make this easier:

- Use a naming convention that makes feature flag strings stand out. For example, if feature flags always have the sentinel prefix `fflag.`, a simple grep search is much more likely to identify flags without many false positives.
- Name feature flags in commit messages. When you introduce a flag, add a commit message like "Introduces feature flag flag-key." Similarly, when you delete a flag, add a commit message like "Removes feature flag flag-key." By doing this, you can quickly find commits that affect feature flags with a Git log.
- If you need to do a comprehensive search, use your repository hosting tool's code search feature. You can also search Git history with a command like `git rev-list --all | xargs git grep flag-key`.

Scaling to Large Teams

I've covered how to introduce feature flags to your team's development workflow, but how do you roll out feature flags across an entire development organization? The more developers are working on an application, the greater the challenge of delivering code together, and the more you need to protect your changes with fea-

ture flags. But a new set of challenges arise when you consider feature flagging across hundreds or thousands of developers.

Permissions and Role-Based Access Controls

Feature management is a practice that can benefit your entire organization. Your feature management console is the source of truth, and it's useful for anyone in your organization to be able see the current, accurate state of the world for your application for any given customer.

You can more tightly control just who can *modify* a feature flag using role-based access controls (RBACs). You can limit changing specific flags, any flag in a given project, or environment (like your production environment). You might want to limit write permissions to your DevOps team or to just the engineers who built a given feature.

That said, I would encourage you to be as open as possible in providing access to feature management throughout your team and across all disciplines from engineering to marketing to support. Feature flags are valuable as much for the collaboration they enable as the simple technical capability of enabling or disabling something, and those benefits should be shared as widely as possible.

Collaboration

After your team is using feature management, there are a handful of practices to observe that will ensure smooth collaborating:

Document changes
: It's good practice to maintain a log of flag updates. It's even more helpful to leave a comment with every change. When something is going unexpectedly wrong, being able to quickly see if anything has changed recently (and why it did) is an invaluable asset.

Make updates visible
: Likewise, it's good practice to make sure that your team knows whenever changes are made. You can try sending notifications to Slack or a similar collaboration tool so that the team knows about flag changes in real time.

Integrate
: After you begin using feature management, answering the question "is this done yet" becomes a bit more complicated. The code might be written and it might be deployed, but only a small fraction of users might be able see it. So, it's good practice to connect your feature management platform—which is the source of truth for feature deployment—to your other collaboration tools, like issue trackers or code repositories. It's super helpful to see the current deployed state of a flag directly from an issue, ticket, or PR.

Organizing Flags

As feature management spreads across a larger organization, one of the challenges is simply how to deal with the sheer number of flags. Here are a couple of methods that you can employ:

Separate concerns
: The most important step is to divide your flags into smaller, related chunks of functionality. You might choose to divide by microservice, by application tier, or by related functionality. But separating your flags into projects or buckets can avoid the need for developers to understand every flag in the system.

Name your flags well
: It's also important to help your team understand what flags are for as easily as possible. So, adopt a naming convention that makes it clear at first glance what a flag is for, what part of the system it affects, and what it does. The more consistent you can be across teams, the smoother your collaboration will be.

Feature Flags Versus Blue/Green Deploys

Blue/green deployments are a technique that involves running two identical production environments ("blue" and "green"). At any point in time, one environment is "live" (e.g., blue), while the other (green) is "idle." To prepare a release, new code is pushed to the green environment. Gradually, traffic is pointed to the green environment until eventually the blue environment becomes idle. At any point during the release, you can direct traffic back to the blue environment if problems occur. You can manage the deploy/ramp up/rollback steps of a blue/green deployment workflow by using a continuous delivery system such as Spinnaker.

Blue/green deployments have some of the same benefits as feature flags: they help reduce the risk of deploying new changes, and both provide the ability to instantaneously roll back a problematic change. Thus, feature flags and blue/green deploys are complementary techniques. Although there are areas of overlap, each approach has distinct benefits, and the best strategy is to use both. Following is a comparison between the major characteristics of feature flags and blue/green deployments:

Granularity
: Feature flags work at the code level, and flags can protect extremely fine-grained changes, down to individual code paths. In contrast, blue/green deploys work by routing traffic to entirely different versions of a service. Usually, this means that blue/green deploys end up protecting services from a set of commits packaged into a binary and deployed together to the idle environment.

Applicability
: Blue/green deployments can protect against *any* change, including infrastructure changes, configuration changes, and code changes. Feature flags, on the other hand, are most easily applied to code changes.

Ability to route users
: Feature flags can change values for users based on arbitrary user attributes and rules. This makes them extremely flexible. On the other hand, blue/green deployments operate at the router level. Routers typically don't have access to user attributes, and most are limited to simple percentage rollouts.

Intended users
: Because blue/green deployments operate on the version level, the primary users are generally DevOps engineers. Feature management tools are used throughout the organization by developers, product managers, and nontechnical users.

Lifetime
: Because there are typically only two environments (blue and green), and additional environments are expensive to keep in operation, blue/green deploys are usually very short lived. Feature flags, on the other hand, can live across many changes, or even be permanent, depending on your need.

When the infrastructure for blue/green deployments is in place, every deploy should follow that pattern. This provides a catch-all rollback plan if anything goes wrong with a deploy, including changes that can't easily be flagged, like configuration changes.

However, if a new problem is traced to a new feature that's flagged, you can turn it off directly without forcing a rollback of any other change that was shipped on that release. In addition, you should use flags for other use cases (like operational flags, A/B tests, and entitlements) that aren't addressed by blue/green deploys.

Feature Flags Versus Configuration Management

A service's *configuration* is a set of parameters that is likely to change between deployment environments (staging, production, development, etc.). This includes information such as the following:

- Credentials to access external services (e.g., Stripe, AWS)
- Settings for backing stores (e.g., pool sizes, host, and port info)
- Deploy values such as the externally facing hostname for the environment

Configuration parameters are typically stored in files, environment variables, or services like Consul or Redis. As services become more complex, *configuration management* becomes a real concern. Tasks like versioning configuration data, rolling back changes, and promoting configuration changes across environments become cumbersome and error prone. More than one company has experienced an outage caused by deploying a bad configuration change to a production environment.

For teams using feature flags successfully at scale, it's tempting to think about moving all of this configuration data into a feature management platform. Feature management platforms solve many of these change management problems, but I still do not recommend moving configuration data into feature flags. Because feature flags are dynamic (they can change at runtime) and context sensitive (they can change depending on the user context), it is more difficult to get a single, simple view of the configuration data. For most configuration data, there's little benefit to be gained by making the data

change based on the user context. In this case, simplicity trumps flexibility.

Rather than migrate all configuration data into feature flags, I recommend introducing feature flags selectively on top of whatever configuration management mechanism is in place (files, environment variables, etc.). These flags should be introduced only on an as-needed basis. For example, imagine that you're trying to manage a database migration via feature flags. You might introduce a simple Boolean flag called `read-from-new-database`. You can then modify your configuration file as shown in the following example:

```
[database]
host = localhost
password = redacted
port = 1234
[new-database]
host = localhost
password = redacted
port = 5678
```

Now, you can use the `read-from-new-database` flag to connect to the new database if true. Otherwise, the original database configuration is used. The configuration data is kept where it belongs, in the configuration management system, but you've gained the benefits of feature flagging, including the ability to do a controlled rollout. With this approach, you can also easily clean up the flag and the configuration file when the migration is completed.

If you had managed your migration by moving the entire database configuration into a feature flag, perhaps by creating a multivariate database-configuration flag, you'd need to keep the flag in place permanently. There's little long-term value to this because you're unlikely to use different database configurations for different users. You'd also be storing your database credentials in the flag management platform, which is also not a best practice.

Summary

Armed with these ideas and tools, you now have a playbook for getting started with feature management. Remember to start small and extend your usage of feature management after you've had a few small wins. Before you get started with your first project, read the

next chapter to learn what you need to consider when selecting your feature management platform.

CHAPTER 3

Selecting a Feature Management Platform

Feature flags are not a new idea in software development. However, the increasing pace of delivery has shifted the technique from a rarely used tool to a requirement in modern applications. With the more frequent use of flags and the sheer number of flags used in software increasing, teams need a scalable, enterprise-grade feature management platform. In this chapter, I discuss important requirements and considerations for your feature management system. Whether you decide to build your own or opt for a third-party feature management service, you should ensure that it is well designed.

Feature Management

When teams embark on a journey of feature management, they often go through similar stages of development:

Stage 0: Config file
: This is typically where teams start with feature flags. Developers create a flat file of values that is read at initialization time to provide configuration values for an application. However, values often can't be changed after application startup and are static, meaning it's not possible to offer different settings to different users. Flag values proliferate until they become difficult to manage in a single file, and the files run the risk of losing synchronization across different deployments.

Stage 1: Database
: Developers often decide to move flag values into an application-wide database. A database can be queried during runtime and updates can be read without the need to reinitialize the system. Simple database systems still usually lack the ability to customize behavior on a per-user basis.

Stage 2: Database with context
: As teams scale, they typically add more context about their flags—who is the owner/maintainer of a flag, what part of the code it is used for, and so on. Often, a rudimentary UI is added to give team members visibility to what exists in the system. Many teams investigate open source tools or dedicate engineering time at this stage to make the system more useful and reliable.

Stage 3: Feature management service
: Feature flags become such an important part of your team's process that it requires a dedicated service to manage them. Scale is typically the greatest driver, both the number of flags being managed and the number of times they must be evaluated each day. This is the stage at which the service goes from a developer tool to a mission-critical business service. A robust feature management platform will solve problems like the following:

  - Distributing information globally and propagating flag rule updates quickly
  - Guaranteeing system redundancy and being able to survive failures with a predictable outcome
  - Ensuring that the appropriate set of people have access to manage flags, and maintaining an audit log of changes
  - Separating different teams, projects, and environments and supporting multiple programming languages and frameworks
  - Targeting specific users or segments with customizable rules

Feature Management Is Mission Critical

Most homegrown systems never mature past stage 2 and can quickly become a liability instead of an asset. For those organizations that are interested in integrating feature management deeply into their

development process, I have a few recommendations to help you be successful.

Design for Scale

Teams designing a feature management system need to consider how to maintain the source of truth for flags, how that information is delivered quickly to where it's needed (to servers and even to end users' devices scattered around the world), and how that information is updated when states change.

It's critical that whenever the application is evaluating a flag that it always receives the same answer regardless of the server, datacenter, or even the continent where the application resides. If one request is served false and another one true for the same user in the same session, users get a confusing and inconsistent experience.

Polling Versus Streaming

In any networked system there are two methods to distribute information. *Polling* is the method by which the endpoints (clients or servers) periodically ask for updates. *Streaming*, the second method, is when the central authority pushes the new values to all the endpoints as they change.

Both options have pros and cons. However, in a poll-based system you are faced with an unattractive trade-off: either you poll infrequently and run the risk of different parts of your application having different flag states, or you poll very frequently and shoulder high costs in system load, network bandwidth, and the necessary infrastructure to support the high demands.

A streaming architecture, on the other hand, offers speed advantages and consistency guarantees. Streaming is a better fit for large-scale and distributed systems. In this design, each client maintains a long-running connection to the feature management system, which instantly sends down any changes as they occur to all clients.

Polling

Pros	Cons
Simple	Inefficient. All clients need to connect momentarily, regardless of whether there is a change.
Easily cached	Changes require roughly twice the polling interval to propagate to all clients.
	Because of long polling intervals, the system could create a "split brain" situation, in which both new flag and old flag states exist at the same time.

Streaming

Pros	Cons
Efficient at scale. Each client receives messages only when necessary.	Requires the central service to maintain connections for every client.
Fast Propagation. Changes can be pushed out to clients in real time.	Assumes a reliable network.

Design for Failure

Feature management systems have become a mission-critical component in the production application stack. In many ways, they act like the central nervous system of your application. Businesses now rely on feature flags to maintain the state of applications and control which features (or feature versions) users will experience. If they are not designed properly, failures in the feature management system can be catastrophic. If it fails (for whatever reason), your application should be designed such that it continues to function.

In practice, this means designing multiple layers of redundancy. When you write code you must consider what should happen if the feature flag system fails to respond. Most feature flag APIs include the ability to specify a default option—what is to be served if no other information is available. Ensure that you have a default option and that your defaults are safe and sane.

Your system should be resilient to momentary interruptions, be able to reestablish a connection to your platform, and resynchronize to the true state of the world, all while the application is running.

Design for Collaboration

The "Mythical Man-Month" is real. The larger the team working on a software project, the greater the communication overhead. It's true when building software, and it's equally true when operating a service. When a large team incorporates feature management into its process, there are techniques that teams can use to work together smoothly.

First, just as it is helpful to separate large codebases into smaller units, you can separate your flags into different projects and help the developers avoid the mental overhead of considering hundreds of flags that they don't need to care about.

When flags are created, they should be assigned an owner or maintainer: someone who understands the context and the purposes of the flag, and, even more important, when that flag is no longer useful and able to be removed. That developer is responsible for the life cycle of the flag, including cleaning it up when it's no longer needed.

The information contained in flags is valuable. It helps to describe the behavior of the system and diagnose what users are actually experiencing. And it's a window into the development process and helps everyone to know where a given feature is in the release cycle. Generally, you'll want to let everyone *view* the state of a flag. But often, you will want to limit who can *change* the state of a flag. You can use role-based access to ensure that the people have appropriate access to the right flags, or even limit permissions per environment so that only certain people can change the state of production. As earlier examples have made clear, giving nondevelopment teams access to the feature management system can have significant benefits to the business and to the development team. With that said, it's up to every team to design the right set of rules for your needs.

It's also important to keep an audit trail of changes made to each flag. Track and make visible every change that is made to a flag, whether it's turning the flag on and off, changing the targeting rules, or updating variations. Record who makes the change, when, and ideally why (comments are great for this). You can also use audit log entries as notifications for the rest of the team via Slack or email.

Adoptable

Most modern Software as a Service (SaaS) applications are composed of many different programming technologies, and multiple languages are used to build the end-to-end application experience. You might implement backend code in Java, Python, or Go, whereas the web frontend is likely JavaScript based, and native mobile applications are built for Android and iOS.

Your users don't recognize that distinction; to them it's one application. Thus, it's important that your feature flags work consistently across all of your applications. When a feature is enabled for a user it must be available across platforms, whether that's in a browser or a native mobile app.

Look for a feature management platform that supports all components of the application, with simple SDKs that present a similar API for your developers and a consistent experience for you users.

Summary

Feature flags have become a mission-critical component of the modern application. Your feature management system must scale and perform to meet the demands of your business. The requirements and considerations outlined in this chapter give you a head start for designing your own or evaluating third-party feature management systems.

About the Author

John Kodumal is CTO and cofounder of LaunchDarkly, the leading feature management platform. John was a development manager at Atlassian, where he led engineering for the Atlassian Marketplace. Prior to that, he was an architect and advanced technology researcher at Coverity, where he worked on static and dynamic analysis algorithms. He has a Ph.D. from UC Berkeley in programming languages and type systems, and a BS from Harvey Mudd College. All in, he has more than 15 years' experience building tools for developers. In his spare time, John climbs rocks, ice, small boulders, and the occasional building, and he enjoys juggling startup and family life.