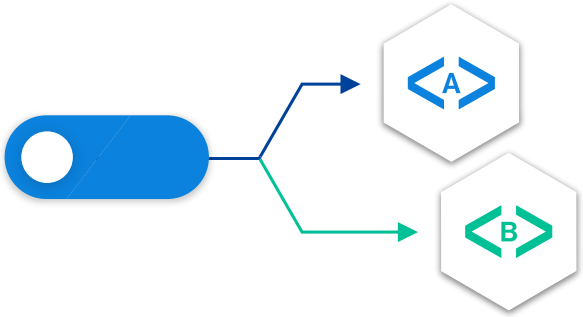
split

Feature Flags: An Essential Guide

Contents

Feature Flagging is a technique that can be used for a range of purposes. Perhaps because of this, the definition of a feature flag can vary quite a bit, depending on who you ask. Nevertheless, whoever you ask, the core concept underlying feature flags remains the same:

A feature flag is a mechanism that allows you to choose between different code paths in your system at runtime.



In this guide, you'll see that this seemingly simple concept can provide a variety of enhancements to your product development process. You'll also explore what sort of functionality a comprehensive feature-flagging system should have, beyond that foundational capability of adjusting your software's behavior on the fly.

Before we go any further, let's take a look at what a feature flag might look like in code. Often it's as simple as a thoughtfully positioned if/else statement:

```
01  function getShippingOptions(){
02    if(FF_OFFER_OVERNIGHT_SHIPPING){
03      return ["standard", "express", "overnight"];
04   }else{
05     return ["standard", "express"];
06      }
07  }
```

You can see here that we only include "overnight" in the list of available shipping options if the `**FF_OFFER_OVERNIGHT_SHIPPING**` flag variable is true. The astute reader might find themselves wondering what makes that flag variable true or false. Good question! How this flagging decision is made will vary depending on what type of feature flag this is, and is an important aspect of a feature flagging system; we'll cover this in detail later in this guide. However, regardless of how the flagging decision is made, the fundamental concept of choosing a code path at runtime remains the same for all uses of feature flagging.

Use Cases

We can get a sense of these different types of feature flags by exploring the different ways a team can use feature flagging. Imagine a product delivery team that's working at an e-commerce company. This team owns the product details page, and they're getting ready to add a big new feature - a "related products" carousel, which they hope will help shoppers find the exact product for their needs.

Decouple Deployment from Release

This is a big new feature, and it will require various other teams to build and release supporting changes before the carousel component itself can go live. The Product Details team wants to avoid tying their deployments up with other teams, though - they don't want to have to pause the release of other work on the page just because another team hasn't finished the changes they need for the related products carousel.

They could solve this by developing the related products feature on a branch, but they've learned through bitter experience that long-lived feature branches can cause a lot of pain and suffering.

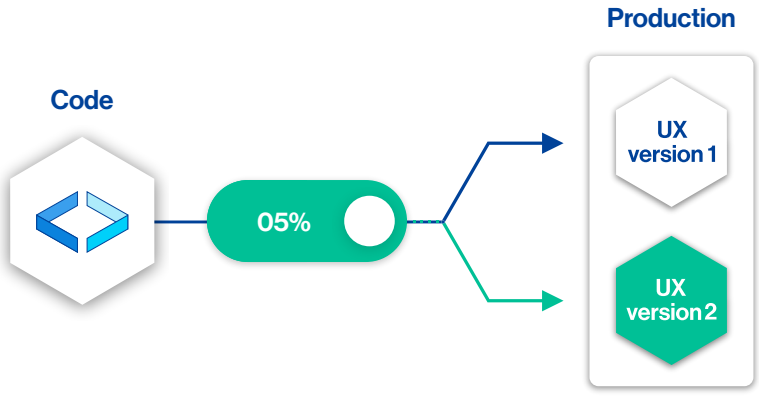
Instead, they opt to use a feature flag instead, placing a flagging decision at the point where the carousel would be generated for the current page:

```
01  function renderRelatedProductsCarousel(){
02    if(!features.enabled('RELATED_PRODUCTS')){
03      return null;
04   }
05
06      // … actual logic to generate related products
07  }
```

When the `**RELATED_PRODUCTS**` flag isn't enabled, we simply skip past the carousel-rendering code entirely. This allows the Product Details team to deploy changes whenever they want - as long as that `**RELATED_PRODUCTS**` flag is disabled, it doesn't matter whether the supporting work from other teams has been deployed yet since the carousel will be hidden. The carousel implementation within the Product Details page itself doesn't need to be tested, or even functional, as long as that flag is going to stay disabled.

This category of feature flag is called a Release Toggle. It allows a team to separate the deployment of code from the release of a feature. As we've seen, this is a powerful capability. It helps a team avoid long-lived branches and decouples changes across different systems, reducing the need for synchronized "big bang" code rollouts.

Controlled Rollout



After a period of time, the team is ready to launch the related products carousel. All the dependent changes from other teams are completed and deployed, and they've tested the feature thoroughly. However, it's a significant change and one that involves a lot of moving parts. The team wouldn't mind an extra safety net. Luckily, they still have that `**RELATED_PRODUCTS**` feature flag in place. While it was initially put in place to hide half-finished work, the team can now use that flag to also perform a Controlled Rollout of the related products carousel.

Rather than releasing the carousel to everyone, the team will initially expose the carousel to just 5% of users. They'll watch what happens with that small initial set of requests to the Product Details page, and if things go well, they'll move on to a full release. This is called a Canary Release - the initial 5% of users are "canaries in the coal mine" who help detect any issues in production without exposing our entire user base to them.

With this canary release, we'll want to be consistent in which users see the carousel - it would be a very confusing user experience if we randomly turned the feature for 5% of page views. To make this work, our feature flagging decision will have to take the current user making the request into consideration so that the flagging system can ensure that the same 5% of users have the flag enabled:

```
01  function renderRelatedProductsCarousel(){
02    if (!features.for(currentUser()).enabled('RELATED_PRODUCTS')) {
03      return null;
04    }
05
06    // … actual logic to generate related products
07  }
```

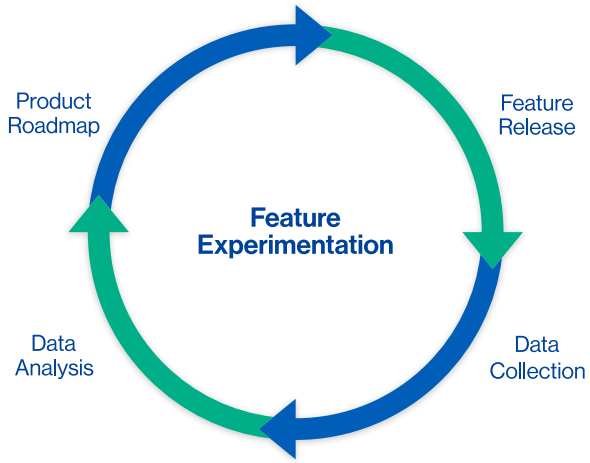
NOTE:

This ability to take context, such as current user, into account when making a flagging decision is crucial for any serious feature flagging system; it unleashes the full potential of feature flags. We'll explore this more in a moment.

After turning the feature on for a small percentage of users, the team watches their system metrics for any issues - a noticeable increase in error rates, or a marked spike in request latency, for example. They don't see anything concerning and feel confident that their new feature doesn't have

any serious issues. However, despite their confidence in the technical implementation of the related product carousel, the team still isn't going to roll the feature out to everyone. Instead, they're going to perform an experiment (specifically, an A/B test) so they can understand how the new related products carousel affects user behavior.

Experimentation



The team's hypothesis is that displaying related products will increase the chance that consumers find their perfect product and make a purchase. The only way to truly validate this hypothesis is by testing the carousel against real users.

The team will do this by performing an A/B test. Their feature flagging system will randomly split their users into two groups, a treatment group, and a control group. The treatment group will be exposed to the new feature - the feature flag will be on for those users. The control group will not be exposed - the feature flag will be off for them. The team will then watch to see whether the treatment group behaves differently than the control group. In this specific case, the team will be watching to see whether the treatment group is statistically more likely to make a purchase (and therefore, whether exposure to the related products carousel makes users more likely to make a purchase). A flag used in this way is called an Experiment Toggle.

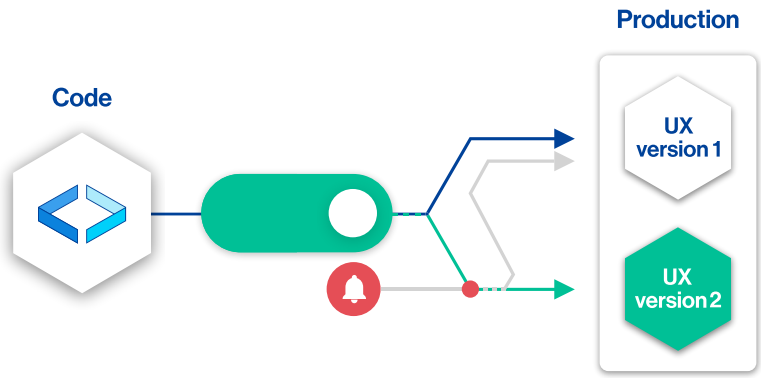
NOTE:

This ability to take context, such as current user, into account when making a flagging decision is crucial for any serious feature flagging system; it unleashes the full potential of feature flags. We'll explore this more in a moment.

After running their A/B experiment for a few days, the team determines that their product hypothesis was correct - users who experienced the related products carousel were 4% more likely to purchase a product. They are finally ready to launch the feature to all users!

But, they're still not going to remove that **RELATED_PRODUCTS** feature flag! It has one last job to do.

Kill Switch



It turns out that the core system which powers the new related product functionality is a little... fussy. A few times during testing it seemed to exhibit some sort of memory leak, and at other times it appeared to cause spikes in CPU usage. The operations team is a little nervous about this and would like some way to gracefully degrade the system if they start to see things going sideways. Our feature flag can help here if we repurpose it for use in an emergency. The feature flag is left in place as a Kill Switch - a way for someone to quickly turn off the related products functionality in production (without the need for a code deploy), just in case anything goes awry in the future.

Feature Flag Adoption is a Journey

Reaching the end of the journey for this particular flag, we now understand that feature flags can serve a wide variety of purposes. This journey through different use cases is also representative of the adoption path for feature flags at most organizations. They are initially intended for one use case - release management perhaps, or experimentation - but over time they are adopted for more and more different usage patterns, by a variety of stakeholders.

Additional Use Cases

There are even more feature flagging use cases which we didn't cover in our journey so far:

Testing in Production

Feature flags allow us to deploy a feature into production but only expose that feature to internal testers. This allows us to test a feature in the same environment our users will experience it - production - sidestepping many of the problems associated with testing in pre-production environments - test data, stability issues, version mismatches, environmental drift, and so on.

Permissioning

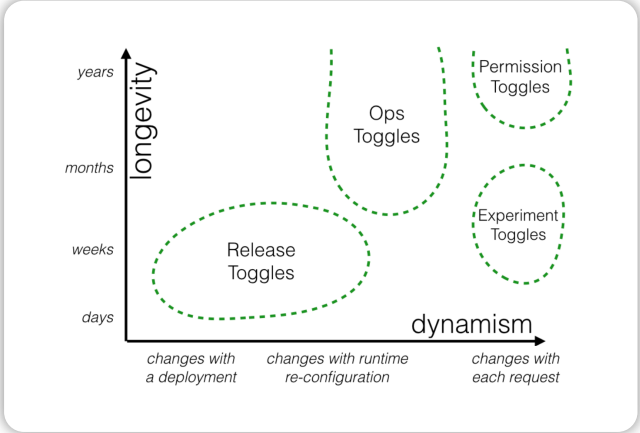
Some organizations use feature flags as a way to implement entitlements - only allowing certain classes of users to access a feature. For example, a feature might only be exposed to administrators, or to users who have a premium subscription. Flags used in this way are sometimes referred to as *Permissions Toggles*.

Caution should be exercised in using feature flagging for this sort of entitlements system. These systems often require capabilities that aren't always available in a feature flagging system - integration with billing, sophisticated group management, audit trails for permission changes, and so on. Often a better approach is to use a separate entitlements system, but establish a shared access pattern, so that the decision to enable a feature can be driven from either your feature flagging system or your entitlements system, without your code having to know the difference.

Architectural Evolution

Feature flags can be used to safely manage large-scale architectural migrations - extracting a chunk of functionality out of a monolith and into a microservice, for example. In these sorts of scenarios we can use branch by abstraction techniques to incrementally build out a new implementation, and use a feature flag to safely manage the transition. This approach is covered in detail in our blog series on managing a monolith.

Categorizing Feature Flags



Because of this wide variety of use cases, it doesn't make sense to treat all feature flags the same. The way that we implement and manage a release toggle should be different from the way we work with a kill switch.

It's helpful to explicitly categorize our feature flags across a few dimensions, and then modify our approach to implementing and managing them based on that categorization.

Longevity

The longevity of a flag refers to how long the decision logic for that flag will be live in a codebase. Some types of flags, such as release toggles, are short-lived - each flag should only be needed in a codebase for a few weeks. In contrast, kill switches and permission toggles are long-lived - they will often stay in the codebase for years.

It's important to be aware of the longevity of a flag when you're implementing the decision logic for that flag. If that logic is for a short-lived flag, and therefore temporary, then it might be acceptable to implement the decision logic using a simple if/else statement. However, if the flag is going to be in place for a long period of time then we should look for more maintainable options, such as a Strategy pattern. We'll revisit this idea later in the guide.

Dynamism

The configuration for some types of flags needs to be more dynamic than for others. The turn-around time for flipping a Release Toggle from off to on can be fairly long. An engineer bumping up the exposure of a controlled rollout would appreciate a reasonably fast feedback loop. An operator wants a kill switch to take effect as quickly as possible.

This dynamism affects the requirements for how flag configuration is propagated, something we'll look at later in this guide when we discuss the need for feature flagging systems to provide a live update capability.

Ownership

Different types of flags are managed by different people. Release Toggles are typically managed by the engineer implementing the feature, or perhaps an engineering manager. Experiment toggles are usually managed by product managers. Kill Switches are managed by operations folk.

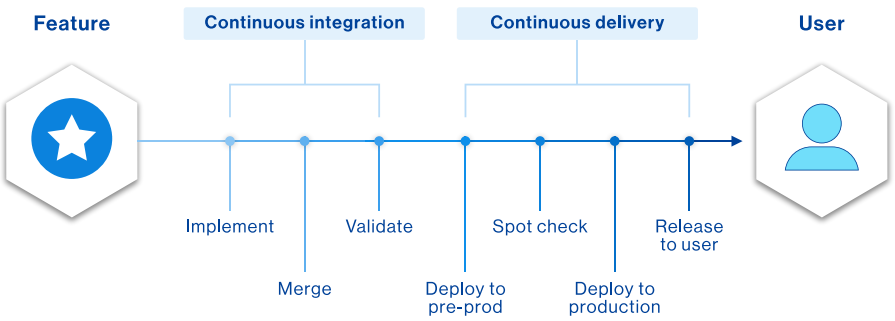
Flag ownership factors into how a flag's configuration is managed. An engineer might be quite happy to manage flag configuration via a source-controlled YAML file, while a Product Manager might prefer a web-based UI.

03 The Value of Feature Flags

We've now seen that flags can be used in a wide variety of ways. Let's take a step back, and summarize the value provided by these different usage scenarios.

Feature Flags Enable Continuous Delivery

Feature flags are an integral part of continuous delivery. They allow engineers to decouple deployment from release and also place control of feature release into the hands of product management. They also allow teams to safely practice trunk-based development, avoiding long-lived feature branches and painful merge issues by integrating code changes very frequently.



Continuous Experimentation

Feature flags allow you to scientifically validate your product ideas. A good feature flagging system allows a product manager to slice and dice their userbase, establishing treatment and control groups for a feature based on various demographics. By then correlating user behavior with feature exposure, we are able to gain insight into the true impact of a feature.

Operational Control

Feature flags provide a very useful control mechanism for people operating a system in production. Adding custom kill switches deep within a system allows operators to degrade a system gracefully when confronted with issues like overwhelming load or 3rd-party outages.

Safer Releases

We can reduce the risks associated with rolling out a new feature using feature flags. We can test in production, deploying a feature's implementation into production but initially turning it on only for internal users. This allows testers to validate the feature in the same environment as our users, with the exact same data and software. After internal testing, feature flags continue to provide more safety by enabling the controlled rollout of that feature. We can release it to a small initial set of users - a canary release - in order to gain confidence, and then incrementally roll it out to more users from there. If anything goes wrong during this rollout process, feature flags provide a fast, effective mitigation strategy - simply turn the feature flag off in production!

04 Capabilities of a Feature Flagging System

We saw at the beginning of this guide that feature flagging is based around a central concept - the ability to choose between different code paths in your system, at runtime. While we can implement this core capability using little more than an environment variable and an if/else statement, we'll need more sophistication if we want our feature flagging system to support the valuable use cases that we've already seen. Let's look at what additional capabilities a good feature flagging system provides, and why they're valuable.

Dynamic, Context-specific Flagging Decisions

Many of the feature flagging use cases we've looked at require some sort of context in order to make a flagging decision. You can't just ask whether a flag is on or off, you have to ask whether it's on in the current context - is the flag on for the currently logged-in user, or for the specific server instance that's servicing the request.

Rich Context

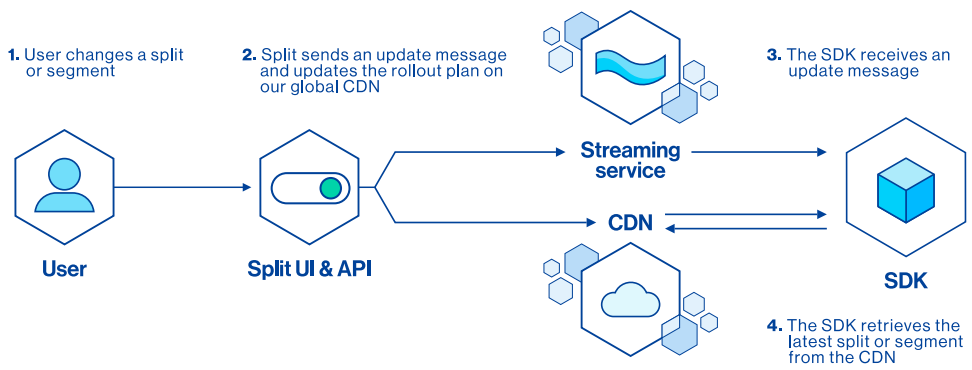
A good feature flagging system provides mechanisms for taking additional context into account when making a flagging decision. At a minimum, this context includes the current user (since flagging decisions are frequently based on user identity), but can also include additional attributes about that user - things like what organization the user belongs to, or what payment plan they are in. Besides user context, we might also want to take into account context such as what server instance is handling a request, or what geographic market the request is related to.

Powerful Decision Rules

When this rich contextual information is available, a feature flagging system can then provide more powerful ways to configure a flagging decision. Rather than saying "I want to expose this feature to 5% of our customer base", I might instead say "I want to expose this feature to 100% of our internal users, 50% of our beta user group, but not to any other users", or "I want to A/B test this feature with 50% of the Atlanta market".

Without this ability to make detailed, context-specific flagging decisions you can't really use your feature flagging system for any of the more advanced use cases - A/B testing, controlled rollout, permissioning, and so on.

Streaming Updates



Any non-trivial feature flagging system needs to manage flag configuration - whether a flag is enabled, or disabled, or enabled for 5% of users, or a certain demographic, or what have you. A good feature flagging system supports streaming updates to that configuration - the ability for any configuration changes to rapidly propagate to the running processes where flagging decisions are being made, without the need to reload or restart anything.

This capability is closely related to how dynamic you need your feature flags to be, and as we discussed earlier this varies depending on the type of flag you are managing.

For very static flags - a release toggle, for example - you might be able to get by with managing flag configuration within a YAML file in your codebase, and propagating flag configuration changes via a code deployment (assuming you have a mature delivery pipeline which can get changes to that configuration out in a timely manner).

However, it's very unlikely that code-based flag configuration would be an acceptable solution for something like a kill switch - for that type of flag you'd want configuration changes to propagate as rapidly as possible.

Ideally, your flag management system allows configuration changes to propagate within a few seconds, not even requiring processes in an environment to be restarted in order to pick up the change.

Rich Flag State

The most common type of feature flag has a boolean state - it is either disabled or enabled. However, it can be useful to have flags with richer state. It's much easier to model an A/B/n test, where there are multiple experimental treatments plus a control, if your feature flag is an enumeration rather than a boolean, having a value of either "treatment_a", "treatment_b", or "control" (for example).

Taken to an extreme, you can use a feature flagging system as a more general mechanism to specify dynamic configuration based on some context. You might want to implement an experiment where you show a 5% discount to users in Atlanta, a 7% discount to users in Dallas, and no discount elsewhere. If your flagging system is providing the discount amount as a configuration value then you can use the context and decision rules that are already in place to implement this experiment. Caution is advised here - this is a powerful mechanism that essentially allows people to make live changes to the configuration of production systems. This great power comes with great responsibility, and should be handled respectfully!

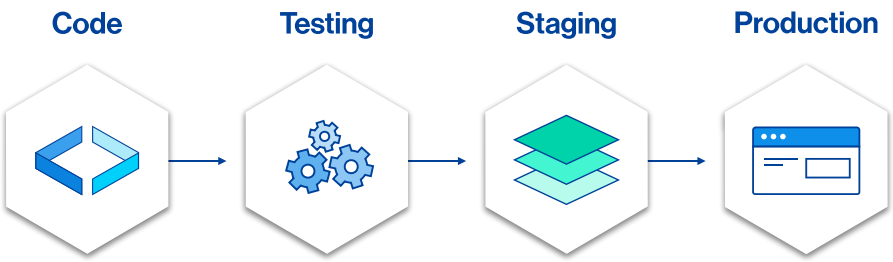
Feature Flag Configuration Management

As a feature flagging system sees more and more types of usage from a wider variety of constituents, managing the configuration of flags becomes crucial.

Ownership and Metadata

The number of feature flags under management has a tendency to grow over time, leading to "flag bloat" - a challenge which we'll discuss later in this guide. The best way to keep this in check is proactive management, a big part of which is tracking metadata for each flag, such as who is responsible for the flag, when the flag was created, when it should expect to be retired, and so on.

Environment-aware configuration



A small flag configuration change can have a big impact on a production system. There is a real risk associated with enabling a feature for users before it's been tested, for example. Because of this, it makes sense to treat a flag configuration change similarly to a code change - with appropriate processes in place for change control and validation. With code-based flag configuration, you get these processes for free - a flag configuration change flows through the same delivery pipeline as a code change. When flag configuration is managed more dynamically, outside of source code, then that system should replicate some of those delivery pipeline concepts.

Specifically, your flag management system should have a first-class concept of deployment environments, and support the promotion of a flag configuration change from one environment to the next. This reduces the need for "swivel chair" synchronization of a configuration change across environments (with the associated risk of human error), and can reduce configuration drift between environments in general.

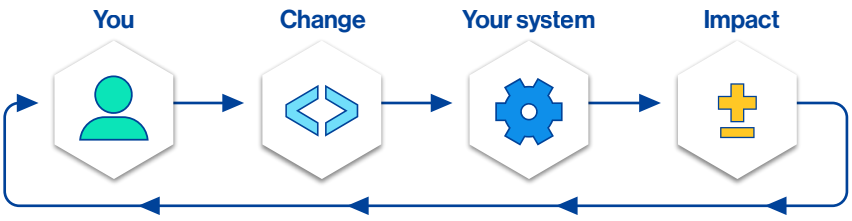
Change Control and Auditability

Because flag configuration changes do carry risk, enterprise organizations often need to restrict who has the ability to make these changes. Regulatory compliance may also require that an audit trail of changes be maintained. This is particularly true when a flag management system is used by a varied constituency of users from different parts of the enterprise, or when the flags are being used in a regulated environment such as finance or healthcare.

Change control is another reason why it's valuable for a flag management system to have a first-class awareness of deployment environments. Tighter restrictions can be put in place around changes to production environments, while still allowing looser access in lower, pre-production environments.

Some flag management systems can even model change control flows directly. Configuration changes are created in a draft state and then submitted for approval by the appropriate stakeholders. Once approved, these changes then flow into the appropriate environments.

Close the Loop - Analytics Integration



Several feature flagging use cases require the ability to understand the impact of a flagged feature. Does it increase conversion rates? Increase API error rates? Increase request latency? To answer these questions we need some way of closing the loop - correlating flag state with the metrics we care about. This is particularly important for experiment toggles - the entire purpose of these flags is to measure the impact of a feature flag - but this ability to correlate flag state with metrics is also very useful for other use cases, such as controlled rollout, and testing in production.

There are two main approaches to correlating flag state with metrics. We can add metadata about flag state to the metrics that we're recording in a dedicated analytics system, allowing us to see how flag state affects those metrics. Alternatively, we can push key metrics into the feature flagging system itself, performing correlation and causal analysis there. Each approach has its pros and cons - the best choice depends upon the capabilities of each system, what kinds of instrumentation are already in place, and upon which stakeholders are using each system.

Challenges with Feature Flags

While feature flagging delivers a ton of value, it doesn't come without its challenges. The implementation of flagging decision points can introduce some cruft into a codebase, which can become quite unpleasant if allowed to accumulate. More generally, a useful feature flagging system can become somewhat a victim of its own success. Flag management can become unwieldy as the number of flags within the system grows, along with the constituency of users.

Let's examine these challenges in more detail, along with some ways to mitigate them.

Messy Code

Implementing the decision points for a feature flag can lead to messy code. Conditional statements scattered throughout your codebase make it harder to understand the flow of the program, and multiple runtime code paths can make it harder to reconstruct the cause of an issue when debugging.

The key to overcoming this challenge is understanding that feature flagging code is still production code - you don't get a free pass to create a byzantine control flow just because it's for a feature flag. This is doubly important for flags that have longevity - the control logic for a kill switch or a permissioning toggle will likely stay in your codebase in perpetuity, so you'd better make that code easy to maintain.

Apply Good Software Design

For long-lived flags, consider using software design patterns that replace conditional logic with polymorphism. The strategy pattern is a great fit for many long-lived feature flags, for example.

Take advantage of your language's type system, where you can. Identifying feature flags via a centralized set of strongly-typed enumerations, rather than a "stringly-typed" magic identifier, will make it easier for you to find usages of that flag when it comes time to remove it.

Decouple Decision Points from Decision Rules

More broadly, you should try to avoid feature-flagging concepts permeating your codebase. You can't avoid the need for multiple code paths if you want the ability to switch between them at runtime, but that doesn't mean that every decision point needs to be thinking about these decisions in terms of feature flags. For long-lived feature flags, it is often better to distinguish between the specific question you're asking in code and the feature flag that will drive that decision. For example, take the following feature-flagged logic:

```
function handleExportRequest() {
   const canExport = features.enabled(currentUser,'FF_PREMIUM_USER')
     || features.enabled(currentUser,'FF_ADMIN_USER');
  if (!canExport) {
    throw Error('you are not allowed to export');
     }
     //…
   }
```

This seems like a reasonable approach, but it needlessly couples the question being asked - can the current user export things - from the logic behind the decision - do they have a specific feature flag, or are they an admin. This will make that logic harder to refactor and harder to test and lead to a slew of references to your feature flagging system strewn throughout the code. Here's a better approach:

```
function handleExportRequest() {
   if (!currentUser.canExport()) {
     throw Error('you are not allowed to export');
  }
  //…
}
```

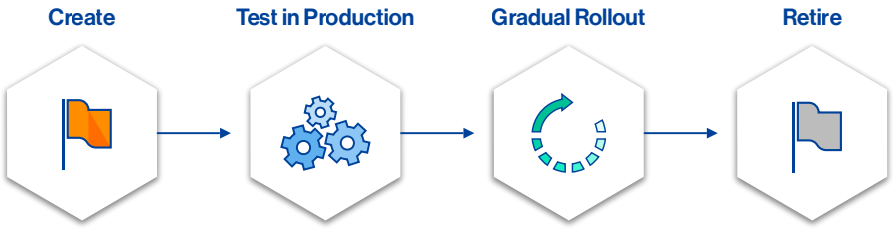
Our `canExport` method will still be implemented using the feature flag API, but that is now an implementation detail. We've removed the direct reference to our feature flagging system, making the code easier to read, easier to test, and more open to change in the future. If these sort of permission checks are very common in our system, we could take this further and create a Permissions object which encapsulates the various permissions that a user may have. That Permissions object might be powered by our feature flagging system, or perhaps switched to some other implementation in the future. The code that uses it won't care, and won't have to change. This general approach of abstracting over the feature-flagging specifics pays dividends. A lightweight adapter layer that makes flagging decisions look natural in the frameworks that are being used can smooth off the rough edges, making it easier to add flags, test their behavior, and clean them up when you're done with them.

Feature Flag Observability

By definition, there is more than one execution path through a feature-flagged codebase. This can make debugging harder since it's harder to understand exactly what code brought the system into its current state.

One helpful approach is to ensure that feature flag state is observable: record relevant flag state in logs, or include flag state metadata in instrumentation. This allows us to reconstruct which code paths were involved in the invocation that we're debugging long after the flagging decisions have been made.

Flag Debt



Even after reducing the ongoing cost of feature flagging code using the techniques I just described, that cost is still non-zero. In addition, overall feature flag management imposes an overhead. The more feature flags we accumulate, the higher these costs become.

Cleaning up flags you no longer need is the best way to keep these costs in check - the cheapest flag is the flag you have removed from your system. However, identifying and removing stale flags is a surprisingly thorny problem. As flag adoption grows, many organizations find it challenging to understand which flags are still active. It can also be tricky to allocate sufficient time to removing stale flags. Let's look at ways to overcome these two challenges.

Identify Stale Flags

When a number of teams use a single flagging system to manage various types of flags, it can be challenging to identify which of the many flags in the system could be retired and removed.

Metadata about each flag - information like ownership, flag type, expected expiration date, and so - can help a lot here. Rather than asking people to audit the entire set of flags in the system, we can group flags by owner, and then ask each owner to assess just the flags they own - a much less

intimidating task. We can also focus on flags of a type that should be short-lived - release toggles, for example - but which have been in the system for a long time. Requiring short-lived flags to have an expected expiration date added to the flag when it is created can make this analysis even easier.

Suppose your feature flagging system also has an audit capability. In that case, it can be used to identify flags that have not been modified in a long time, since these are potentially flags that are no longer under active management. Similarly, if you have closed the loop with good feature flag analytics then this can be used to identify "dead" flag states. However, this also illustrates why metadata around flag type is important - even if a kill switch hasn't been used for 9 months, we still don't want to remove it!

Ensure Flags are Removed

Having identified stale flags, many teams still find it hard to carve out time to actually remove these flags, along with the implementation in their codebase. Like a lot of tech debt tasks, it's easy to succumb to the temptation to defer the work. While everyone agrees that it's important, it's not urgent, and thus tempting to de-prioritize.

Here are a few tricks you can use to make sure flags do eventually get cleaned up:

- Put a clean-up task into your backlog at the time you create the flag.
- Define an expiration date for the flag when you create it. You can then use that metadata to find stale flags (as described earlier). You can also get more aggressive, and emit logging, fire error events, or even refuse to boot your application if an expired flag is detected.
- Make flag debt visible by tracking the number of flags that each team owns as a KPI and encouraging teams to keep this number below a threshold.
- Take the KPI measurement further, and make it a hard Work in Progress limit on how many short-lived feature flags your team is willing to manage at any one time. If you've reached your limit, you need to remove an existing flag before you're allowed to create a new flag.
- Institute regular "flag bash" events - a work day set aside for engineers to focus on ripping out stale flags.

Test Your Feature-flagged Code

Testing a feature-flagging system can seem an intimidating prospect. Do we have to test everything twice - once with a flag off, and then again with it on? What about multiple flags? We can't possibly test every combination of flag states! What's more, feature flagging systems provide an additional vector for production changes, which may not have the same change controls as a code change would. How do we make sure that these production changes are tested and safe?

The testing overhead isn't as bad as it might sound. Yes, for active flags, you should test the system's behavior with the flag off and with it on, but this isn't much different from testing the system before and after a code change is made. When it comes to the combinatorics of multiple flags, as long as different flagged behavior doesn't interact, you don't need to test the various combinations of those flags. For example, if you have one feature flag that affects your login screen and another that affects your user profile screen, then you can likely consider these flags as independent and don't need to test all four combinations of the two flags.

The testing burden of feature flags can be kept in check by having a strategy for identifying which flag states currently need testing. The concept of "active" flags is essential here - that is, the set of flags that could feasibly be in both enabled and disabled states in production in the near future. Examples would be a release toggle for something that's going to be going live soon, an active experiment toggle, and any sort of long-lived permissioning toggles or kill switches. You can also institute policies within your flag management system that reduce the risk of bad production changes. For example, for any flag that goes active in production, someone must first have tested it appropriately in a pre-production environment.

Suppose you do have flags that are intertwined. In that case, it does make sense to analyze which possible combinations of flags you'd see in production and test those combinations - this is an expensive but unavoidable cost to interdependent flags. There are further costs to these types of flag dependencies beyond the testing burden - they also lead to confusing, complex coding patterns. As such, it's advisable to keep these sort of interdependent flags to an absolute minimum. If you must have them, you should explicitly define a small subset of states that such flags are allowed to be in.

One final tip when it comes to testing: it is worth investing some engineering effort in making your tests "flag aware". For example, you can create extensions to your testing frameworks that let you annotate tests with declarations of which flag states should be verified. Your testing framework can then take care of setting that flag state for the test's duration, performing

multiple runs with different flag states, if appropriate. You can provide similar facilities for manual testing too - exposing dev-only tooling that allows a tester to override flag state in order to test a specific scenario, for example.

Flag Management Sprawl

It's important to note that because feature flags provide this wide variety of valuable capabilities, they're useful to a wide variety of people. Engineers value trunk-based delivery. Product managers value experimentation and controlling when features go live. Operators value kill switches. QA folks value the ability to test in production and perform controlled rollouts.

This variety of use cases and stakeholders can lead to "flag management sprawl", where an org has multiple feature flagging systems in use, each focussed on a different stakeholder group. For example, one system might be used for release toggles and controlled rollout, another for experimentation, and a third for kill switches. In this scenario, we should limit the impact of the sprawl. Multiple management systems are bad, but even worse is a code base where all the flagging decisions have to consider configuration from each system. If you do find yourself with multiple feature flagging systems, it's worth introducing a unifying abstraction at the code level.

06 TL;DR - Feature Flagging Should be Simple and Drive Efficiency

In this guide we've discovered that feature flagging takes a simple concept - choosing between different code paths at runtime - and uses it to provide a wide array of benefits. Feature flagging enables engineers to work more effectively by practicing Continuous Delivery, empowers product managers to take a rigorous, data-driven approach based on experimentation, and reduces risk by providing capabilities like testing in production, controlled rollout, and kill switches.

However, feature flags are not a free ride - the benefits that they deliver come with an attendant set of costs. Proactive flag management and thoughtful implementation techniques can keep these costs in check.

Teams can get started with feature flagging quite simply, but a comprehensive feature flag management system is required in order to maximize the benefits of this powerful technique while minimizing the cost.