

Introduction to Feature Toggles and Implementation Best Practices

An introduction to feature toggles, their types, practical implementation experience, and best practices for using them safely.

01 MAR 2018 · 14 MIN READ



Darren Sim

Senior technology & product leader

All content published here is shared in a personal capacity. Nothing on this site should be interpreted as an official statement, endorsement, or communication from my employer.

As software professionals, success is measured by our ability to improve our users' lives.

Unlike other engineering fields which have been around for hundreds of years, the software engineering discipline has existed for barely a hundred years. As an industry, software development is still largely part-art, part-science.

With the ever evolving business and technological landscape, many of our past experiences serve merely as references. Adding to the unknowns, users seldom know what features they want, but they know what they don't want. More often than not, it takes several attempts of trial-and-error before software teams become successful in delivering features that stick.

The ability to continuously deliver working software into the hands of customers, has served as a competitive advantage for leading tech companies like Amazon and Facebook. Not only did this capability allow engineering teams to deploy code into production in small chunks, quickly, and safely, it also shortened the feedback loop – allowing teams to learn fast and respond to changes quickly.

In the practice of Continuous Delivery (CD), it's important that the application is continuously integrated (CI), and deployable at all times. This can be challenging especially in situations where a feature spans multiple commits, and, an urgent fix needs to be released before the development of that feature is complete. This highlights the need for deployment and release to be decoupled.

When a feature is completed and deployed to production, teams also need a way to experiment safely, beta testing with a small group of users and being able to roll back quickly if an undesirable experience is observed.

Feature Toggles offers options to overcome to these challenges.

In this article, we will explore: **1) what feature toggles are, 2) how we can use them**, before **3) discussing best practices**.

What are Feature Toggles?

Feature Toggling is not a new concept and was a common practice more than 20 years ago, in the absence of reliable and easy-to-use concurrent versioning systems (CVS). In recent years, Feature Toggle experienced a resurgence with the popularisation of DevOps practices and mindsets within software development teams.

Also known as “Feature Flags”, “Config Flags”, “Flippers”, “Feature Bits” and “Conditional Feature”, Feature Toggling is a set of patterns that help teams deliver new functionality to users rapidly but safely, separating deployment from release, allowing teams to modify system behavior without changing code. Implemented by simply wrapping an if-statement around a code block that we are trying to isolate, the condition of this if-statement can be a simple boolean or complex decision trees with multiple paths.

The basic idea here is that we have a configuration system that allows the development team to toggle features on or off. When the application is executed, it will use these toggles to decide what features to display. The actual implementation will vary – ranging from a hard-coded configuration value within the application where changing the toggle configurations require a redeployment of the application, to reading toggle states from a text file, or a web service, allowing teams to modify toggle configurations at runtime.

Some of the benefits that Feature Toggles bring to the table include helping teams eliminate nasty merge conflicts, reduce deployment risks and achieving a shorter feedback loop. However, there's always two side of a coin. Apart from introducing technical debt the moment we use Feature Toggles in our code, Feature Toggles also results in code being more fragile, harder to test, harder to maintain, and harder to support. Hence, Feature Toggles should always be short-lived, and used on a need-be basis rather than it being the silver-bullet to software development.

One of the most common misunderstandings of Feature Toggles is that it can double as a licensing and authorization service. This results in tight coupling of responsibility and should be avoided.

Types of Feature Toggles

In his thought-leading article [Feature Toggles \(aka Feature Flags\)](#), [Pete Hodgson](#) extended [Martin Fowler](#)'s original post "[FeatureToggle](#)", nicely categorizing Feature Toggles into **FOUR** categories:

Short-Lived Toggles

Dynamism	
RELEASE TOGGLE	Largely static, changing release toggle decision usually done in code; requiring redeployment.
EXPERIMENTAL TOGGLE	Highly dynamic; changes with each request
Longevity	
RELEASE TOGGLE	Transitory by nature; should not stick around longer than 1 weeks.
EXPERIMENTAL TOGGLE	Long enough to generate statically significant results, usually kept around for hours to weeks.
Benefits	
RELEASE TOGGLE	Allows incomplete and untested code-paths to be deployed to production.
EXPERIMENTAL TOGGLE	Provides development team with means to make data-driven decisions.
Disadvantages	
RELEASE TOGGLE	Results in dead code that will not add value to anyone.
EXPERIMENTAL TOGGLE	Highly dynamic in nature, requiring a third-party services; a conscious technical debt.
Implementation	
RELEASE TOGGLE	By hardcoding the condition in the if-statement to false.
EXPERIMENTAL TOGGLE	Usually by means of an external feature toggle web service.
When to Use	

RELEASE TOGGLE

When developing code that is not ready for consumption and has a high risk of breaking the application.

EXPERIMENTAL TOGGLE

When performing multivariate testing (e.g. AB Testing)

Long-Lived Toggles

Dynamism	
OPERATION TOGGLE	Relatively static; changes be done with runtime reconfiguration.
PERMISSION TOGGLE	Highly dynamic in nature; changes may happen between requests.
Longevity	
OPERATION TOGGLE	Usually from weeks to years.
PERMISSION TOGGLE	Usually spanning months to years.
Benefits	
OPERATION TOGGLE	Allows system operators to disable / degrade a low performing feature quickly without having to perform a redeployment of the code.
PERMISSION TOGGLE	Perform targeted experiments and learnings.
Disadvantages	
OPERATION TOGGLE	Tends to stick around forever; leading to increased system complexity.
PERMISSION TOGGLE	Tends to stick around forever; leading to increased system complexity.
Implementation	
OPERATION TOGGLE	By means of a config file on a CDN or other means of data cache services.
PERMISSION TOGGLE	Usually by means of an external feature toggle web service.

When to Use

OPERATION TOGGLE

When rolling out a new feature with unclear performance impact.

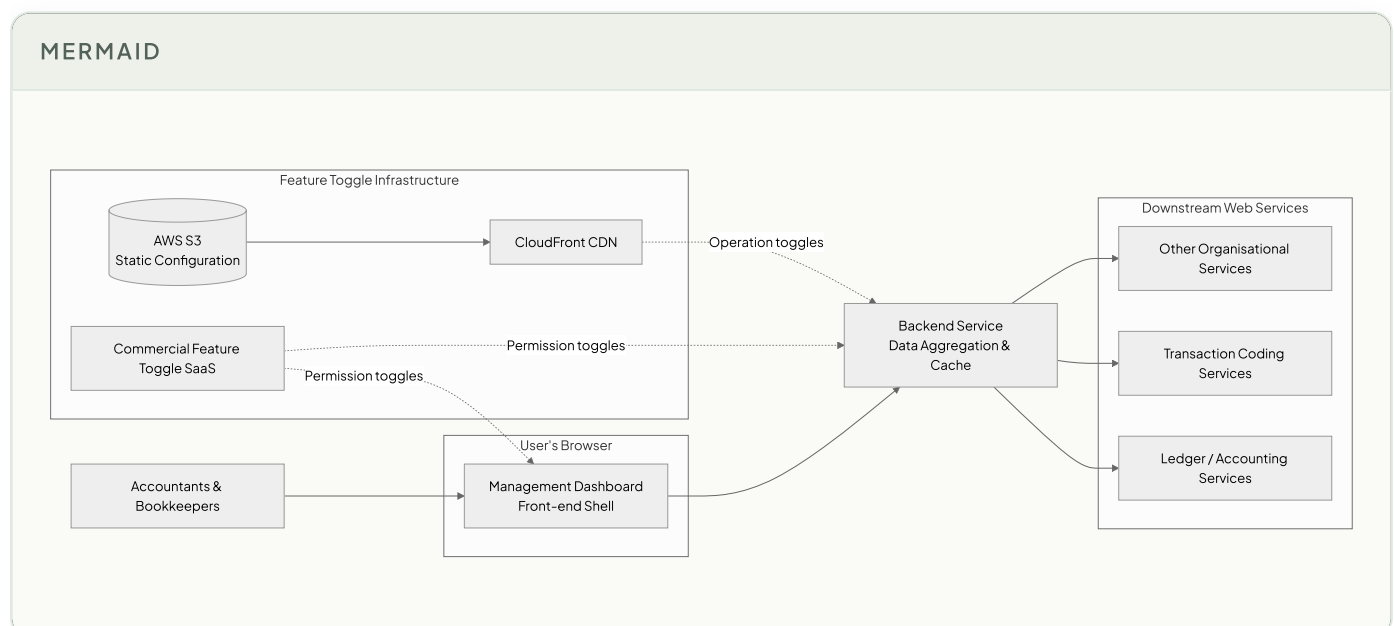
PERMISSION TOGGLE

Used in cases of dog-fooding, alpha and beta programs.

While this categorisation is by no means all-encompassing, it serves as a good reference when making decisions on designing Feature Toggles implementations.

My Team's Experience with Feature Toggles

My team has been working with feature Toggles for the last 18 months and have gone from using an internally developed, maintained and hosted Feature Toggling service, to a commercial operated SaaS-based solution.



Our bread and butters as a team is to build and operate a management dashboard for accountants and book keepers. Through this dashboard, our users will be able to get a birds' eye view of all their client's accounting books (also known as "ledgers") and categories transactions (also known as "transaction coding") where required. Data is fed to this management dashboard through a plethora of web services that are maintained by various teams within the organization.

The management dashboard comprises of a front-end shell that deals with everything that runs on the browser, as well as a backend service that acts as a data aggregation and caching service, used exclusively by the front-end.

In our team, we have adopted practices such as Continuous Integration (CI), Continuous Deployment (CD), Trunk-based Development and Test-Driven Development to help us deliver working software quickly and safely.

Given that we practice elephant carpaccio during story slicing and work mainly in a mob or as pairs, we found Release Toggles to be less useful. Instead, we adopted the Feature Branching approach when developing new features. With Feature Branching, we are able to make multiple commits while working on the feature, without generating excessive noise in GIT. As we squash our commits with links back to the original feature branch, GitHub allow us the ability to rid the feature branch, while still being able to look back at those micro-commits.

The two toggles that we used were **Operations Toggles**, and **Permission Toggles**.

As a downstream service that depends on many other services for data, the ball is seldom in our courts. Each of these services that we depend on for data, is usually built and operated by different teams, and has varying performance thresholds.

Operation Toggles has been especially effective; offering us the capability to gracefully degrade or disable calls to these dependent services in the event that the business dashboard or any of it downstream services experience a traffic surge resulting in undesirable system performances and/or behaviors.

Given that this services doesn't have to be highly dynamic, we developed a simple setup that required little to no maintenance. Backed by an AWS S3 bucket to store a static configuration and a CloudFront Distribution to deliver the configuration payload, this infrastructure cost us close to nothing to operate.

In order to better understand if features were developed and delivered in a way that actually increases the productivity and effectiveness of our users, we have also introduced beta programs to get early feedback before a feature is released to the entire user base.

Given that this service needs to be able to form toggle configurations dynamically based on a user's profile, locality and many other factors, a highly dynamic service is required. Leading vendors that offers such services include [FeatureAvailability.IO](#), Launch Darkly, and [Split.IO](#). Given that these services usually offer very similar feature offerings and pricing, decisions usually boils down to product roadmaps, and quality of support. Flagr and Petri are two popular open source options that can be downloaded and self-hosted.

Commercially operated solutions typically charge based on a per-request model; hence implementing on stateless functions (e.g. AWS Lamda) can be costly given that caching of this state is not possible. Feature Toggle states are usually cached for up to a minute within an application to reduce the chattiness between the application and the Feature Toggle service.

Feature Toggle Best Practices

Feature Toggling is a useful technique that helps team avoid painful merge conflicts, experiment and learn fast; however not without risks. Here are some of my lessons learnt and best practices:

Business Metric and Traffic Monitoring Dashboard

A key benefit of Feature Toggling is the ability to perform experiments and learn from feedback, quickly and safely. Hence having a business metric dashboard is of high importance, allowing us to understand the business impact of a new feature. Consider an e-commerce system; if the introduction of a new feature results in massive decrease in sales, that is something that is probably a useful feedback that is worth looking into in greater detail.

Avoid Using Release Toggles

As Martin Fowler has put it, while release toggles is a useful technique that is used by many teams, they should be used as a last option when putting new features into production. Instead, we should be adopting the practice of breaking features down more so that they can be safely introduced into the product. The benefit of doing so is synonymous with that of any strategy advocating small and frequent releases. Risks can be reduced and development teams can get early feedback that can be used to improve the feature.

Avoid Coupling Decision Point and Toggle Router

Decision point is defined as the location in the code that calls the toggle router, while the Toggle router contains the logic to check if a certain decision point is active by reading toggling configurations to detect features, or simply detect an Toggle Context.

JAVASCRIPT

```
1  if(User == "john@doe.com" && Weather == "Sunny"){
2      DoSomething();
3  }
```

JAVASCRIPT

```
1  if(User.Segment == "Beta"){
2      DoSomething();
3  }
```

The two examples above exemplifies patterns we want to avoid, as they tightly couple toggles decisions into the application logic.

JAVASCRIPT

```
1  if(FeatureSwitchSvc.IsPsychicEnabled){
2      DoSomething();
3  }
```

What we want instead, is to de-couple decision points from decision logic by introducing a decision object that implements that logic. In this way, the strategy-pattern can be applied, encapsulating routing conditions to the routing layer. Thus a change in feature grouping will not break the code.

Avoid Nesting Feature Toggles

If nesting if-statements increases the cyclomatic complexity; nesting Feature Toggles increases that by many folds.

Dashboard with Feature Toggles Longevity

Like branches, long-lived feature toggles should be avoided as it adds significant complexity to the application and causes the code-base to be brittle over time. While it is easy to visualize how many stale branches you have in Github, long-lived feature toggles are less obvious and runs the risks of snuggling in forever. Hence a physical or digital dashboard in the team area that keeps track of the longevity of each Feature Toggle is always a good practice.

Always Check for Security

One of the common misconceptions is that if you hide a feature behind a feature flag, it is not accessible and hence we do not have to worry about security. This is untrue and a dangerous mindset to have in software development; a phenomenal known as “the security sandwich”. Consider a feature toggle implemented on the front-end. What is stopping a malicious user from manually setting that flag to true in the console? Security as an afterthought is always recipe for trouble.

Automated Tests Should Exercise Both Toggle On and Off States

If TDD is practiced in the team, it is always a good habit to practice red-green-refactor. When implementing feature toggles, it is always important that automated tests exercise flows with the switch turned on, and off, to avoid regressions. While this is not a silver bullet, it helps radiate obvious regressions. The added complexity in testing introduced by feature toggles warrants reviewing existing test strategies; constant and effective communication between between members of the development team is also important.

This article was first published in the Jan 2018 issue of Pragmatic Publications Magazine.



— THE AUTHOR

Darren Sim

Darren is a senior technology and product leader based in Singapore. He writes about the decisions, systems, and people behind meaningful transformation across Asia-Pacific.

[More about Darren](#) ↗

[Connect on LinkedIn](#) ↗