MEGA GUIDE

30 Feature Flagging Best Practices

Essential tips, tricks, definitions, use cases, and more to improve your feature management workflow

LaunchDarkly

Contents

Introduction

Feature flags, sometimes called feature toggles, are standard practice for many software development teams. As the top feature management platform, we're often asked for advice on feature flag best practices.

And what are those best practices? Well, it depends on

- Whether the flag is a short-term or permanent flag.
- The purpose of the flag.
- Your specific business needs. What works for one company may not work for others.

In this ebook, you'll read about the nuances of feature flagging best practices, so you can avoid technical debt and other common troubles and use feature flags to your benefit. Below, you'll learn about best practices for:

1. Short-term and permanent flags
2. Release management
3. Operational feature flags
4. Experimentation

Whether you're just getting started or are a seasoned veteran, we'll demonstrate how your teams can get the most out of feature flags.

7 best practices for short-term and permanent flags

First, you need to determine whether a flag will be a short-term or permanent flag, as that will influence future decisions and best practices.

Short-term flags

A short-term flag has a limited lifespan, and you generally remove the flag once it has fulfilled its business purpose. Thinking of feature flags, most people think of short-term flags. Types of short-term flags include:

Release

Slowly exposing a feature to new users—moving from internal to beta and/or canary users until 100% of users receive the feature. When you reach 100%, remove the flag (unless it's needed as a circuit-breaker as described below).

Segment your population to determine a preference for one option over another. Once testing ends, remove the flag, and 100% of users should receive the preferred variation.

Operational interaction testing

When rolling out a new microservice, infrastructure component, or third-party tag, a flag can be used to determine the impact on systems. If the CPU spikes or there is a memory leak, or unexpected errors occur, disable the new element while further troubleshooting takes place.

Maybe you're thinking that a release flag sounds a lot like an operational interaction testing flag. They are similar, but the primary difference is the controller. An operational interaction flag is controlled by the ops team to protect the systems, whereas the release flag is controlled by the product or business owner and controls how user adoption progresses.

Permanent flags

A permanent flag is designed to provide control for an extended time after the release of a given feature. In some cases, the flag will be in existence for the life of a feature.

Circuit breakers/Load shedding

Having the ability to quickly turn a feature off or terminate a connection when problems arise prevents problems from impacting all users. These flags are often activated based on an event. For example, a monitoring tool generates an alert when orders fail to complete. When the alert is triggered, a flag is toggled, setting the site to "read-only."

White labeling

Configuring the look and feel of an application for each client for a white-labeled solution.

Best practices for all feature flags

Whether you have a permanent or short-term flag, consider these best practices. Please note: These are recommendations that we follow and have collected from other customers. Recommendations may change over time. As such, feel free to modify and alter based on your specific needs. What works well for one company or team may not work well for another team.

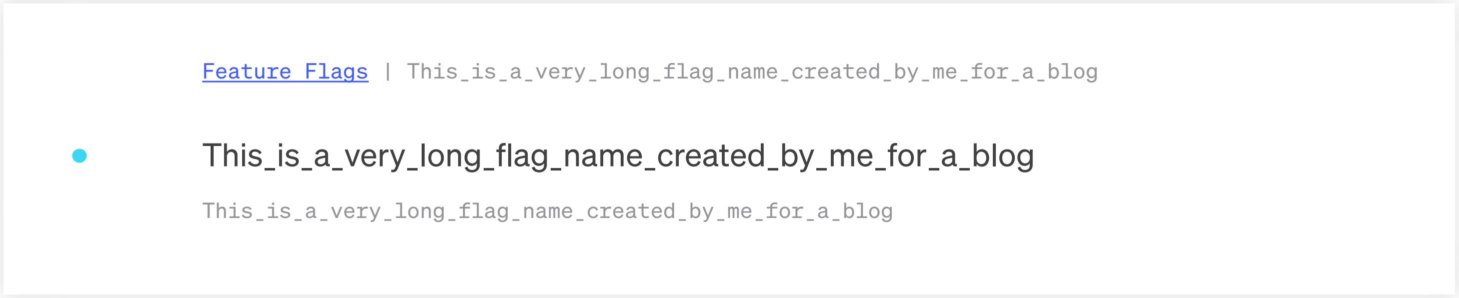
1 Make flag planning a part of feature design

Feature flags shouldn't be an afterthought. If you think about feature flags during the design process, you will be setting yourself up for success. Once you decide whether the flag will be a short-term or permanent flag, this decision will then impact other areas such as a naming convention, configuration settings, review and removal processes, and access control and safety checks. We suggest proper planning upfront for all flags.

2 Standardize naming

You may have a style guide outlining conventions on how to write code for your application; this could include things like when and where to use CamelCase or the proper use of indentation. These style conventions make it easier to read and understand the code.

Before creating your first flag, come up with a naming convention to be used. Our first recommendation is for verbose flag names. Don't try to be brief. Verbose flag names can help others understand what the flag does.



Feature Flags | This_is_a_very_long_flag_name_created_by_me_for_a_blog

This_is_a_very_long_flag_name_created_by_me_for_a_blog

This_is_a_very_long_flag_name_created_by_me_for_a_blog

Things to consider when writing the style guide or naming convention:

- Be descriptive about the flag's behavior.
- Include a prefix with a project name or team name.
- Indicate whether the flag is temporary or permanent.
- Include a creation date for the flag. (This will be helpful when cleaning up flags, more on this below).

Whether or not to use flag in the name: When using a service like LaunchDarkly, using flag in the name is redundant. If you're using a home-grown solution, using flag in the name may help clarify the purpose of the code.

For example, suppose you are creating a flag to progressively roll out and test a new chatbox widget of your UI. This will be a short-term flag. Without a naming convention in place, you may end up with a flag called “brand-new-flag” or “new-UI-widget.”

These names don’t tell us a whole lot about the flag. But with a standard naming convention in place that addresses all of the above, you can create a more descriptive flag name like “aTeam-chatbox-widget-temp-030619.” We know from the name that this is a temporary flag for a chatbox widget created by the “a team” on June 3—much better!

3 Minimize the reach of a flag

The focus of a flag should be small. Having a flag that controls more than one feature action at a time can be confusing and will make troubleshooting issues harder. Think about the smallest unit of logic needed for the most responsive flag. If there are multiple parts to a feature that have to work together, we suggest creating a master flag as a dependency.

For example, say you’re launching a new dashboard in your application. The dashboard has three widgets. You should create a total of four flags: one flag for each widget with a dependency on a fourth flag for the main dashboard. With this scenario, if one widget causes problems, the dashboard with two widgets can still be served.

4 Review use at regular intervals

Whether creating a short-term or permanent flag, you need to review flag use at regular intervals. The frequency at which you review the flags may vary based on business requirements and the type of flag. To avoid accrual of technical debt, check both permanent and short-term flags at a regular cadence.

For short-term flags, look to see if the flag has rolled out to 100% of users or if a flag is served to no users. For permanent flags, examine whether the flag is still needed (was a feature once part of a premium bundle but is now available for all users?) We will cover specific criteria related to removing short-term flags below.

5 Establish access control and safety checks

If you have regularly scheduled flag clean-up events, you may worry about the accidental removal of permanent flags. Minimize this risk by implementing access control and safety checks.

Within LaunchDarkly, a flag cannot be deleted without confirmation, but that is a partial solution. There are two additional ways to implement:

1. Use tags and custom roles to assign permissions to flags within LaunchDarkly quickly.
2. Set role-based access control (RBAC) to specify who can delete flags in a given environment.

Best practices for short-term feature flags

1 Create a process for removing flags before you create one

Coding a flag is a two-part process. The act of removing a flag should not be a separate process from the act of creating a flag. As mentioned above, you should plan for flags during the feature design process. This includes the removal of short-term flags. An easy way to handle the removal process is to write a pull request to remove the flag at the time you create it.

Schedule a GitHub reminder for after the feature is deployed to review and determine if the PR to remove the flag should be committed.

2 Conduct regular clean-up and review cycles

Avoiding and eliminating technical debt is necessary. Here are some ways to schedule flag clean-up.

- Schedule time at the end of every sprint to review existing flags.
- Perform a clean-up/refactoring sprint at a regular cadence (quarterly, semi-annually, whatever works for your business) to pull out old flags & tags.
- Make it a competition. Hold a “Capture the Flag” day. The individual or team that removes the most flags wins.

Within LaunchDarkly, we make it easier to identify flags for removal.

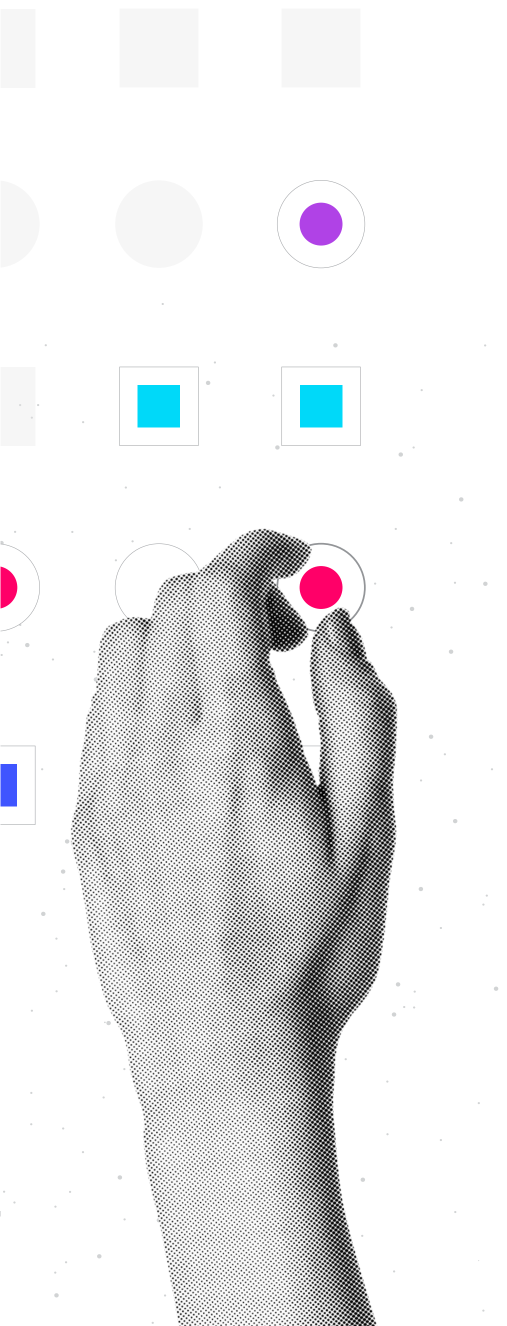
You can:

- Filter tags by creating a date to view the oldest flags
- View which variations of a flag were recently served. Are 100% of users receiving the same variation?
- Filter on the last evaluation date of a flag

Once you have identified a flag for removal, use code references to find all the instances of that flag in your codebase.

If you are not using LaunchDarkly, a consistent naming strategy can help you prep the code for instances of flags.

9 best practices for release management



Release management feature flags use cases

When most people hear “feature flag,” they probably think of a flag that controls who can see a feature and when. In fact though, release management is one of the most common use cases for feature flags.

Whether you call it early access, a canary release, or a beta, giving a select group of users access to features prior to release helps you fine-tune a feature before releasing it to everyone. Starting small and rolling out to larger groups over time helps you:

- Observe the behavior of the systems and services under increasing load
- Collect user feedback and make changes if you need to
- Limit the blast radius if something goes wrong

A release management flag is intentionally short-lived. Once 100% of users have access to the feature, remove the flag.

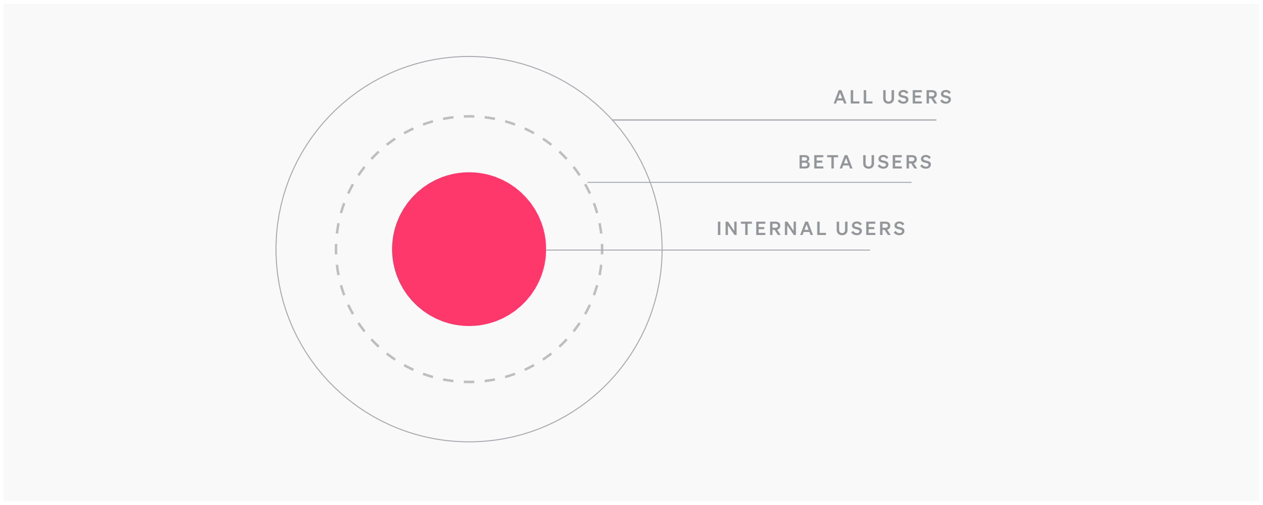
Deploying release management feature flags

The two most common deployment models to use are ring deployments and percentage-based deployments.

1 Ring deployment

Jez Humble introduced the concept of a ring deployment in his book “Continuous Delivery.” In this scenario, different groups gradually receive the feature to manage the risk of deployment. Users for each group are selected based on a set of similar attributes or an opt-in process. Then make features available to the selected groups.

For example, release to internal users first, then beta users, and then to all users:



2 Percentage deployment

The selection of who receives a new feature occurs randomly in a percentage-based deployment. The new feature rolls out to 10% of users, then 25%, then 50%, until all users receive the new feature. These deployments are useful when you cannot run a beta program or have little variation in your targeted user base.

Suppose you run a small, B2B application with the same 1,000 monthly active users. In that case, a percentage-based release is likely to give you an accurate estimate of whether the full rollout will have a positive or negative impact on your user base.

Some companies automate percentage-based rollouts, gradually increasing the number of users receiving the new feature. The rollout is terminated if there are too many errors or if a response time exceeding a set threshold occurs.

You can combine ring and percentage-based deployments. You first would deploy with selected groups and then use a percentage-based deployment to roll out to the remainder of your user base.

Release management best practices

Creating release management flags is a three-part process: planning, implementation, and deployment. Consider the following best practices for each phase in the process.

1 Planning

Make feature flags a priority from the start

Again, the right time to think about whether a feature needs a flag is during the design process. The design of a feature may vary based on whether or not it is behind a flag. When feature flags are an afterthought, you end up with flags that don't work as expected, don't target the right users, or have unexpected behaviors.

Thinking about a feature flag during the design phase requires you to consider who will use the feature. If you determine the feature is for larger enterprises during the design phase, you know that targeting customers at small or mid-sized businesses would not be appropriate.

Understand the purpose of a feature flag

A flag can serve many purposes. Release flags can control how a feature rolls out to users as well as serve as a kill switch if something goes wrong. If a flag will serve multiple purposes, notify all parties and make sure everyone has access to the flag.

Establish naming conventions

As mentioned earlier, a consistent naming strategy ensures everyone understands the purpose of the flag and ultimately helps avoid
technical debt.

2 Implementation

You've planned the feature; now you're ready to implement the flag. Questions to ask to determine the deployment path:

- Do features get deployed to staging, pre-production, or test environments before rolling out to production?
- How will you roll out the feature to users?
- Is it first used by developers, then internal staff, then external users?
- Will there be a beta or a canary launch?
- Will release occur through a controlled rollout or to all users at once?

There is not a right or wrong answer when it comes to how you deploy, but to create an appropriate flag, you need to have a deployment plan in place.

Test the behavior of the feature flag

We are big proponents of testing in production. However, this doesn't mean you only test in production before you deploy a flag—you should test the behavior of the application with the flag toggled on and off. This is especially important if the feature is related to a schema or database change. What happens when a feature is on and then toggled off? Is data lost or corrupted?

Another aspect of testing is dogfooding—using your own product or service internally. This lets you test out changes on internal employees first. Put yourself in your customer's shoes. With dogfood environments, you can see firsthand the pains that users might experience with a feature. Using the application as a user would use it is a step towards empathy-driven development.

You can test all you want before deployment, but until customers use the feature in production, you won't know if it is working and what needs to be changed. This is why we regularly endorse testing in production.

Configure a fallback value

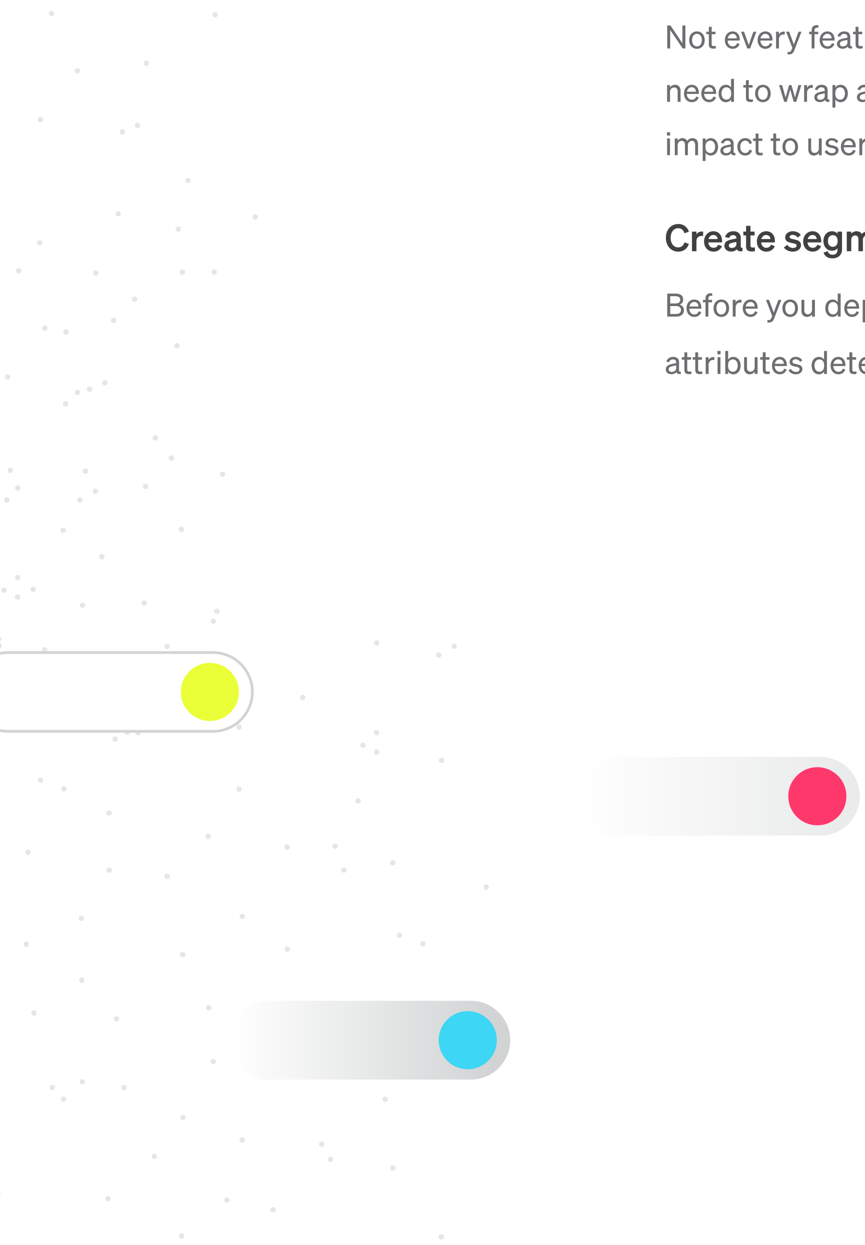
What happens with a flag if a user is not able to access the most recent settings? What happens if a user accesses your application in offline mode? Configuring a fallback value can eliminate unexpected experiences or errors with interrupted connections. Every flag should have a fallback value describing the expected behavior of the feature in case failures occur during evaluation. Should the flag be on or off by default?

Keep features small, but not too small

Not every feature needs a flag, and sometimes a feature needs multiple flags. You don't need to wrap a tiny change in a feature flag unless you're very concerned about its impact to users. Large features such as a new dashboard might require multiple flags.

Create segments or cohorts

Before you deploy, you need to determine how to segment your users. Identify which attributes determine who sees a feature when.



3 Deploying

You've planned and implemented the flag; all that's left is to cross your fingers and deploy to production. Wait, not so fast! As you deploy, consider the following:

Track metrics

What metrics do you need to track regarding features? Availability, reliability, and response time of an application should not change with the deployment of a new feature. Set up monitoring to track performance before, during, and after you deploy code to ensure performance does not get worse. Tracking metrics such as page load time, errors, uptime, conversions, or user registrations can tell you whether a feature is performing well or causing problems. The metrics you track will vary based on your business and objectives.

Avoid technical debt

Release management flags should be short-term. This means you should remove a flag after it has rolled out to 100% of users, as mentioned above. Technical debt accumulates when you forget to remove a flag from the code. Put processes and reminders in place to remove the flag.

4 best practices for operational feature flags

Using feature flags to maintain high system stability, migrate your infrastructure, recover from incidents quickly, and more.

Operational feature flags use cases

Operational flags can be either short-term or long-term flags. Here are some common use cases.

Operational interaction testing

You can't always predict how a new microservice or third-party tag will perform. These need to be tested in production to fully see how everything interacts. If the migration does not go as planned or unexpected interactions occur, you can use a kill switch to turn the service off quickly. Flags used for operational interaction testing are short-term flags. Once the service is entirely in production, remove the flag.

API rate limiting

A flood of API requests can make a highly-reliable API less reliable. Whether the API requests are coming from a misconfigured endpoint, a misbehaving script, or a DDoS attack, operational flags can help you limit the requests on the fly. You can change the number or types of requests allowed with a long-term operational flag.

Modify logging level

Verbose logs are great for debugging and troubleshooting but always running an application in debug mode is not viable. The amount of log data generated would be overwhelming. Changing logging levels on the fly typically requires altering a configuration file and restarting the application. A multivariate operational flag enables you to change the logging level from WARNING to DEBUG in real-time.

Release management

Releases don’t always go smoothly. Some companies use blue-green deployments to maximize a good user experience during a release. With a blue-green deployment, there are two nearly identical production environments. One environment is active while the other is idle. A new release is rolled out to the idle environment, and the traffic is cut over to this environment. If a problem occurs, traffic can quickly be re-routed to the idle environment with the previous version of code.

Blue-green deployments minimize risk with deploys by always having a last-known working configuration available to roll back to if problems occur. That said, one of the challenges of blue-green deployments is that all new features and functionality have to be rolled back—not just the feature(s) with issues.

Using an operational feature flag as a kill switch allows you to turn off only the problematic feature while all the other new features remain in production.

The majority of operational flags used for release management should be short-term flags.

Load shedding

Load shedding refers to reducing the load on systems. Non-critical requests are dropped or disabled to make sure that critical tasks complete. When your site is experiencing a traffic spike, turning off some features can help ensure essential tasks are complete. If your application is under heavy load, you may need to temporarily disable features that increase latency and stress on the database.

Turning off resource-intensive features that won't perform optimally under load can circumvent potential problems and a social media backlash. Adding long-term operational flags to turn off non-essential features during times of increased load helps reduce stress on the systems.

Operational feature flags best practices

1 Document the feature flag

Good documentation is important for permanent operational flags. Accidentally deleting a permanent flag would be a bad thing. Include information on when and why to use the flags in operational and service runbooks. Explain in detail what behavior may change and the impact it may have on your users. Provide information on how to communicate to end-users when toggling features off to eliminate inquiries and complaints that features are missing.

2 Indicate which feature flags are permanent

A well-defined and well-documented naming strategy is useful for informing everyone which flags are permanent and which are short-term. We recommend including the following information in a flag name:

- A prefix with a project name or team name.
- Whether the flag is temporary or permanent.
- The creation date of the flag.

Using the above rules, if you wanted to create a new, permanent operational flag for the comments section, the name might look like this:

```
opsTeam-comments-widget-perm
```

If you are creating a temporary flag as a temporary kill switch during a release, the flag name might look like this:

```
springRelease-comments-widget-temp-June2019
```

In addition to the naming convention, you can also label a flag as permanent from the LaunchDarkly interface.

3 Control who can change a feature flag

Put processes and controls in place to prevent operational flags from accidentally being flipped or deleted. Just like with permanent feature flags, we recommend assigning permissions to flags within LaunchDarkly using tags and custom roles and setting role-based access control (RBAC) to specify who can delete flags in a given environment.

4 Use a relay proxy

Many operational flags reduce stress on your infrastructure. While not strictly related to operational flags, a relay proxy serves the same function as operational flags.

The LaunchDarkly relay proxy lets several servers connect to a local stream instead of making a large number of outbound connections. If you are concerned about the number of outbound connections from your infrastructure to LaunchDarkly, the relay proxy will drastically reduce the number of concurrent connections.

9 best practices for experimentation

Running experiments helps you bridge gaps between technical and business teams as you learn about user engagement patterns and application behavior.

What makes a good experiment?

Experiments can validate that your development teams' efforts align with your business objectives. You can configure an experiment from any feature flag (front-end, back-end, or operational), but what makes a good experiment?

Experiments use metrics to validate or disprove gut feelings about whether a new feature is meeting customers' expectations. With an experiment, you can create and test a hypothesis regarding any aspect of your feature development process, such as:

- Will adding white space to the page result in people spending more time on the site?
- Do alternate images lead to increased sales?
- Will adding sorting to the page significantly increase load times?

Experiments provide concrete reporting and measurements to ensure that you are launching the best version of a feature that positively impacts company metrics.

General experimentation best practices

Let’s first talk about best practices around creating an experiment. Even with a solid foundation of feature flagging in your organization, missteps in these areas can yield flawed results. Consider these best practices for experimentation.

1 Create a culture of experimentation

Experiments can help you prove or disprove a hypothesis, but only if you are willing to trust the outcome and not try to game the experiment. Creating a culture of experimentation means:

- People feel safe asking questions as well as questioning the answer. It’s ok to question the results and explore anomalies.
- Ideas are solicited from all team members—business stakeholders, data analysts, product managers, and developers.
- A data-driven approach is used to put metrics.

Part of creating a culture of experimentation is providing the tools and training to allow teams to test and validate their features.

Needed for experimentation:

- Tools to help you collect relevant metrics can include monitoring, observability, and business analytics tools
- Tools to help you segment users
- Tools to clean and analyze the results

2 Define what success looks like

Experiments help you determine when a feature is good enough to release. But how do you define what “good enough” is and who is involved in creating the definition? Experiments can involve a cross-functional team of people or only a handful, depending on the focus of the experiment. For example, if an experiment is focused on whether to add a “free trial” button to the home page, you may need to involve people from demand generation, design, and business development.

A good experiment requires well-defined and agreed-upon goals and metrics by all stakeholders. Ask yourself what success looks like. It should be defined as improving a specific metric by a specific amount. “Waiting to see what happens” is not a goal befitting a true experiment. Goals need to be concrete and avoid ambiguous words. And then, goals and metrics should be tied to business objectives like increasing revenue, time spent on a page, or growing the subscription base.

👎 Examples of poor goals include vague and ambiguous statements:

“Users will be happier with the home page”

“The response time of search results will improve”

👍 Examples of better goals typically include concrete statistics:

“Paid conversions from a trial will improve by 7%”

“The response time of search results will decrease by 450ms”

3 Statistical significance

Looking at a single metric is good, but viewing related metrics is even better. Identify complementary metrics and examine how they behave. If you get more new subscribers—that’s great, but you also want to look at the total number of subscribers. Let’s say you get additional subscribers, but it inadvertently causes existing subscribers to cancel. Your total number of subscribers may be down as a result, so does that make the experiment a success? Probably not.

The number of samples will depend on your weekly or monthly active users and your application. No matter the size of the samples, it is important to maintain a relatively equal sample size. Significantly more samples in one cohort can skew the results.

4 Proper segmentation

Think about how and when users access your application when starting an experiment. If users primarily use your application on the weekends, experiments should include those days. If users visit a site multiple times over the course of a couple of weeks before converting and before the experiment starts, early results may be skewed, showing a positive effect when it was neutral or negative.

There are two aspects to segmentation. First, how will you segment your users, and second, how will you segment the data? A successful experiment needs two or more cohorts. How you segment your users will vary based on your business and users. Some ways to segment users:

- Logged in vs. anonymous
- By company
- By geography
- Randomly

No matter how you segment users, always make sure the sample sizes are balanced. Having skewed cohorts will result in skewed results.

When analyzing the data after an experiment, you may want to do additional segmentation to see if the results vary based on other parameters.

> ⚠ **Caution**
>
> Don't go digging to find data to support your hypothesis. The segmentation should be done to understand anomalies better—not to make a case to call the experiment a success.

5 Recognize your bias

Everyone is biased. This isn't necessarily a bad thing; it is simply a matter of how our brains work. Understanding your biases and those of others will help when running experiments. Our biases influence how we process information, make decisions, and form judgments.

Some biases that may creep in:

- Anchoring bias - our tendency to rely on the first piece of information we receive
- Confirmation bias - searching for information that confirms our beliefs, preconceptions, or hypotheses
- Default effect - the tendency to favor the default option when presented with choices
- The bias blind spot - our belief that we are less biased than others

It is common in an experiment to scrub the data to remove outliers. This is acceptable, but make sure you aren't eliminating outliers due to bias. Don't invalidate the results of your experiment by removing data points that don't fit the story.

6 Conduct a retrospective

Experiments are about learning, so hold a retrospective after each one. Questions to ask can include:

- Was the experiment a success? Did we get the results we expected? Why or why not?
- What questions were raised by the experiment?
- What questions were answered?
- What did we learn?

And most importantly, should we launch the feature? You can still decide to launch a feature if the results of an experiment were neutral.

Experimentation feature flags best practices

Now that we've covered the basics of experiment best practices, let's dive into best practices around using feature flags in your experiment.
Well-executed feature flagging is a foundation for a well-run experiment.

If you've embraced feature management as part of your teams' DNA, any group that wants to run an experiment can quickly do so without involving engineering.

Follow these best practices to get the most from your existing feature flags:

7 Consider experiments during the planning phase

The decision of whether to wrap a feature in a flag begins during the planning phase—which is also the right time to think about experiments. When creating a feature, evaluate the metrics that will indicate whether the feature is a success: clicks, page load time, registrations, sales, etc.

Talk with the various stakeholders to determine whether an experiment may be necessary and provide them with the essential information on the flag, such as the name of the flag, to configure an experiment.

8 Empower others

When giving others the ability to run experiments, make sure you have the proper controls in place to avoid flags accidentally being changed or deleted. Empower other teams to create targeting rules, define segments, and configure roll-out percentages while preventing them from toggling or deleting flags.

9 Avoid technical debt

Experimentation flags should be short-lived. This means removing a flag after an experiment completes and is either rolled out to 100% of users or not released. Put processes and reminders in place to remove the flag.

1 best practice for leveraging best practices

Lastly? It’s always a best practice to modify any advice for your team. Recommendations may change over time, and what works well for one company or team may not work well for another. That said, we hope you’ve found these tips helpful and if you’re interested in staying updated on the latest best practice, check out the LaunchDarkly blog.

If you’re interested in taking the plunge with LaunchDarkly, see why consulting firm Forrester said you’ll get a 245% ROI from our platform in their report, “The Total Economic Impact of LaunchDarkly,” or go here to view survey results and learn more about the ROI LaunchDarkly has provided our users.

Empowering all teams to deliver and control their software.

launchdarkly.com

sales@launchdarkly.com

LaunchDarkly