Engineer

Feature flag

a switch turning code on or off without rebuilding the app

What it is

A feature flag is a decision point in the code that reads a setting the team can change without building the application again. Behind the flag sits a new code path that is already in production but stays dormant until the setting turns it on, for everyone or for a chosen group of users. Pete Hodgson described the technique at length on Martin Fowler's site, in instalments from January 2016, with a revision in October 2017. He calls them feature toggles and lists feature flags, feature bits and feature flippers as other names for the same thing.

Hodgson splits a flag into parts. The toggle point is the place in the code that asks whether the new path is on. The toggle router answers that question from the toggle configuration and, when the answer depends on who is asking, from a toggle context: usually the current user, or a special cookie or HTTP header that stands in for one. He warns against coupling the two. The code at the toggle point should ask a narrow question, such as whether this order confirmation includes the new cancellation link, while the rule that decides it lives in one place behind the router, so it can change without touching every call site.

Flags differ in how long they live and how often their answer changes, and Hodgson sorts them into four kinds. Release toggles let unfinished, untested code ship to production as latent code; he says they "should generally not stick around much longer than a week or two". Experiment toggles split users into groups for an A/B test, with each user placed in the same group every time. Ops toggles let operators change how the system behaves in production. Many of them are short-lived, but a few long-lived kill switches let operators scale back functionality the system can do without when load is unusually high, such as a recommendations panel that is expensive to generate; Hodgson likens these to a manually managed circuit breaker. Permissioning toggles decide which features a given user sees, such as premium features for paying customers or alpha features for the company's own staff, and they can stay for years.

A feature flag is not a canary release, though the two often work together. A canary runs two versions of a service side by side and routes a share of the traffic to the new one; a flag ships one version that contains both paths and picks between them on each request. A long-lived branch in version control also keeps new work apart, but the code then reaches production only when the branch is merged, all at once.

Use the concept in a decision

Put “What is a feature flag?” to work in a real decision. We can help relate the concept to the system you are building or buying. One team taking an idea from a sketch to something your customers can log into, instead of handing it between three suppliers.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Led a GIS map application to 10x revenue growth — 10x revenue growth, led from the map product

Led a GIS map application to 10x revenue growth Get in touch →

Why it matters

The main reason to use flags is to separate deployment from release. Hodgson describes release toggles as the most common way to put a Continuous Delivery principle into practice: the release of a feature is a separate step from the deployment of its code. Code can go to production every day from a single main branch while a month-long feature stays dark until it is ready, which is what makes trunk-based development workable.

Because a flag can give different answers to different users, a release can be widened in steps. Hodgson describes turning new features on first for internal users, which he calls a Champagne Brunch, since the company gets to drink its own champagne first. A canary cohort comes next: his example is a random 1 per cent of users, chosen with a modulo of the user ID so that the same people stay in the group, while the other 99 per cent keep the old behaviour. If that cohort reports problems, the flag goes off and nobody has to roll back a deployment.

All of this has a price. Every flag adds a branch to the code and a combination to test, and the configuration behind it becomes a part of production that can change without a build. Hodgson’s advice is to keep toggle configuration in source control and change it through re-deployments wherever the flag allows it, so that the configuration moves through the same pipeline as the code and changes to it are reviewed and recorded. Flags that must change within minutes, such as ops toggles, are where runtime configuration earns its keep.

What it looks like in practice

flagd is an open-source flag evaluation engine that follows the OpenFeature specification. It has no user interface and stores nothing of its own; it reads flag definitions from a JSON file or another source and answers requests, either as a sidecar, inside the application process or as a central service. A flag that sends about one user in twenty to a new checkout looks like this:

```
{
  "$schema": "https://flagd.dev/schema/v0/flags.json",
  "flags": {
    "new-checkout": {
      "state": "ENABLED",
      "variants": {
        "on": true,
        "off": false
      },
      "defaultVariant": "off",
      "targeting": {
        "fractional": [
          ["on", 5],
          ["off", 95]
        ]
      },
      "metadata": {
        "owner": "checkout",
        "expires": "2027-01-31"
      }
    }
  }
}
```

Each flag has a state and a set of variants that must all be of one type; the default variant is optional. The `fractional` rule splits users into buckets by relative weight, so 5 and 95 put about one user in twenty on `on`. By default the bucket is computed from the flag key and the `targetingKey` in the evaluation context, which means the same user always gets the same answer, and that renaming the flag reshuffles everyone. `metadata` holds whatever the team wants to attach to the flag, here an owner and the date by which it should be gone. Setting `state` to `DISABLED` makes flagd answer with no value at all, and the caller falls back to its own default.

Running flagd against the file and asking it for an answer over HTTP:

```
flagd start --uri file:./flags.json
curl -X POST "localhost:8013/flagd.evaluation.v1.Service/ResolveBoolean" \
  -H "Content-Type: application/json" \
  -d '{"flagKey":"new-checkout","context":{"targetingKey":"user-4711"}}'
```

The reply carries the value, the variant and the reason `TARGETING_MATCH`. In the application, the call goes through an OpenFeature client with the flag key, a default value of `false` and the evaluation context, and the toggle point sees nothing more than true or false. Raising the share from 5 to 20 per cent is an edit to the flag file, not to the application build.

The hiring and seafarer career workspaces, from 2025, show a release flag in the recorded implementation snapshot: it describes the hiring side as live and the career workspace as feature-flagged and incompletely released. That is a historical snapshot, not a declaration of the present deployment state. The operator back-office from the same year is where flags like that get changed. Operators flip flags and turn a feature on for only some users from a screen, instead of an engineer editing the production database by hand, and each change is authorized and recorded like any other action there.

Where it goes wrong

Flags that never leave. A release flag that has done its job is pure cost: a branch nobody needs, a path nobody tests and a setting someone may flip by mistake. Hodgson suggests treating flags as inventory that carries a cost, and keeping that inventory small. His remedies include a removal task on the backlog the moment a release flag is introduced, an expiry date, a time bomb that fails a test or stops the application from starting once a flag outlives its date, and a fixed limit on the number of flags, so that adding one means removing another. Removing a flag means deleting the losing code path as well as the check.

An old flag given a new meaning. On 1 August 2012, Knight Capital’s automated order router, SMARS, sent millions of orders into the market over 45 minutes, and the firm lost more than $460 million. The SEC’s order sets out how. New code for the New York Stock Exchange’s Retail Liquidity Program reused a flag, carried on incoming orders, that had once switched on Power Peg, a function Knight had stopped using in 2003 but never removed. The deployment missed one server out of eight, so on that server the reused flag woke the old function. When staff then removed the new code from the other seven, the flag

activated Power Peg there too. Never give a retired flag a new meaning, delete dead code together with the flag that guarded it, and check that every server runs the same build before a flag goes on.

Combinations nobody tested. Two flags give four combinations and ten give over a thousand; Hodgson calls it a combinatoric explosion. Testing all of them is rarely possible, so test the ones that can reach production: the current production configuration with the flags about to be released turned on, the fallback with those flags off, and, as many teams also do, everything on. A fixed convention helps: off always means the existing behaviour and on means the new one.

A default nobody chose. The OpenFeature specification says flag evaluation must not throw. When something goes wrong, the call returns the default the caller passed, the detailed form of the result carries an error code, and the client should not write to the log either, because an application deployed before its flag is defined would otherwise flood the log with `FLAG_NOT_FOUND`. That keeps the application running, but it also means a typo in a flag key silently becomes the default for every user. Pass the old, safe behaviour as the default, and attach a hook, which is where the specification puts logging, to count evaluations that end in an error and raise an alert when they appear.

Overrides anyone can set. It is convenient to let testers force a flag on with a special cookie, a parameter in the URL or an HTTP header. Hodgson points out that curious or malicious users can set the same override and change the state of a flag for themselves. Honour overrides only on internal traffic, or sign them cryptographically, which eases the concern but, as Hodgson notes, adds complexity and attack surface of its own.

Further reading

- Feature Toggles (aka Feature Flags) — Pete Hodgson's article on Martin Fowler's site: the parts of a toggle, the four categories, testing and keeping the inventory small.
- Flag Evaluation API — the OpenFeature specification for evaluating a flag, including defaults, error codes and why evaluation never throws.
- flagd: Flag Definitions — the JSON format flagd reads: state, variants, default variant, targeting and metadata.
- flagd: Fractional Operation — deterministic percentage splits and what decides which bucket a user lands in.

- In the Matter of Knight Capital Americas LLC — the SEC's 2013 order on the August 2012 trading loss, including the reused flag.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how "What is a feature flag?" applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/feature-flag/