PuigFlows

Feature Flags en Producción: Despliegues sin Riesgo con OpenFeature y Rollouts Progresivos

Feature Flags in Production: Risk-Free Releases with OpenFeature and Progressive Rollouts

Guía · Dificultad intermedia · Recurso Premium · ~37 min de lectura por idioma
Edición ampliada — julio 2026 · Documento bilingüe: Español / English

Contenido: tipos de flag y ciclo de vida · OpenFeature y flagd · contexto de evaluación · rings, canarios y guardarraíles · experimentos A/B · frontend y móvil · testing · observabilidad con OpenTelemetry · resiliencia · gobernanza · caso práctico completo · checklist de producción y FAQ

Versión en Español

Por qué los feature flags cambian tu forma de desplegar

Desplegar código y lanzar una funcionalidad parecen lo mismo, pero no lo son. Cuando ambas cosas ocurren a la vez, cada despliegue es una apuesta: si la funcionalidad falla, la única salida es un rollback completo, con las prisas, el estrés y el riesgo que eso implica. Los feature flags rompen ese acoplamiento. El código llega a producción apagado, y tú decides cuándo, para quién y a qué ritmo se enciende.

Esta separación tiene consecuencias prácticas enormes: puedes integrar ramas pequeñas y frecuentes sin esperar a que la funcionalidad esté completa (trunk-based development), probar en producción con usuarios internos antes de abrir el grifo, y apagar una funcionalidad problemática en segundos sin tocar un solo pipeline. No es casualidad que el despliegue progresivo se haya convertido en la práctica estándar de los equipos que despliegan varias veces al día.

Los cuatro tipos de flag (y por qué conviene distinguirlos)

No todos los flags son iguales, y tratarlos igual es la vía rápida hacia el caos. La clasificación más útil distingue cuatro tipos según su propósito y su vida esperada:

- **Release flags**: ocultan funcionalidad incompleta o sin validar. Deben vivir días o semanas, nunca meses. Se eliminan cuando la funcionalidad llega al 100 % de los usuarios.
- **Experiment flags**: dividen el tráfico para un test A/B. Viven lo que dura el experimento; cuando hay resultados, se consolida la variante ganadora y el flag desaparece.
- **Ops flags**: interruptores operativos como kill switches o modos degradados. Son de los pocos que viven indefinidamente, y a cambio exigen tests periódicos de ambos caminos.
- **Permission flags**: controlan acceso por plan, rol o cliente (por ejemplo, funcionalidades premium). También son permanentes, pero en realidad son lógica de negocio y conviene plantearse si deberían migrar a tu sistema de permisos.

El tipo determina el ciclo de vida. Un release flag que sigue vivo seis meses después ya no es una herramienta: es deuda técnica con disfraz.

OpenFeature: un estándar para no casarte con un proveedor

Durante años, adoptar feature flags significaba acoplar tu código al SDK de un proveedor concreto. **OpenFeature**, un proyecto en incubación de la CNCF, resuelve exactamente eso: define una API unificada para evaluar flags, y cada proveedor (LaunchDarkly, Flagsmith, Unleash, GO Feature Flag, o tu propio backend) se conecta detrás como un *provider* intercambiable. Si mañana cambias de herramienta, tu código de aplicación no se entera.

Así se ve en Python:

```
# pip install openfeature-sdk
from openfeature import api
from openfeature.evaluation_context import EvaluationContext

# En producción, registra el provider de tu backend de flags
# (LaunchDarkly, Flagsmith, Unleash, GO Feature Flag...)
api.set_provider(MiProvider())
client = api.get_client("checkout-service")

def procesar_pago(user):
    ctx = EvaluationContext(
        targeting_key=user.id,
        attributes={"plan": user.plan, "pais": user.country},
    )
    if client.get_boolean_value("nuevo-flujo-pago", False, ctx):
        return nuevo_flujo_pago(user)
    return flujo_pago_actual(user)
```

Dos detalles importan aquí. El segundo argumento de `get_boolean_value` es el **valor por defecto**: lo que se usará si el backend de flags no responde. Elígelo siempre pensando en el fallo, no en la conveniencia — para funcionalidad nueva, el defecto seguro casi siempre es `False`. Y el `EvaluationContext` transporta los atributos del usuario que permiten segmentar: plan, país, versión de la app, lo que necesites.

El equivalente en TypeScript (lado servidor):

```
// npm install @openfeature/server-sdk
import { OpenFeature } from '@openfeature/server-sdk';

await OpenFeature.setProviderAndWait(new MyProvider());
const client = OpenFeature.getClient('checkout-service');

const enabled = await client.getBooleanValue('nuevo-flujo-pago', false, {
  targetingKey: user.id,
  plan: user.plan,
  country: user.country,
});

if (enabled) {
  await nuevoFlujoPago(user);
} else {
  await flujoPagoActual(user);
}
```

Una regla de seguridad que no admite excepciones: la lógica sensible se evalúa **en el servidor**. Un flag evaluado en el navegador puede inspeccionarse y manipularse; nunca debe ser lo único que protege una funcionalidad de pago o un dato restringido.

Rollouts progresivos: porcentajes con bucketing determinista

Encender una funcionalidad para el 5 % de los usuarios exige responder una pregunta engañosamente simple: ¿qué 5 %? Si lo decides con un número aleatorio en cada petición, el mismo usuario verá la funcionalidad aparecer y desaparecer entre página y página. La solución es el **bucketing determinista**: un hash estable del usuario y el flag que asigna a cada usuario un punto fijo entre 0 y 100.

```
import hashlib

def esta_en_rollout(flag: str, user_id: str, porcentaje: float) -> bool:
    """Asigna cada usuario a un bucket estable entre 0 y 100.

    El mismo usuario cae siempre en el mismo bucket para un flag dado,
    así que subir el porcentaje solo AÑADE usuarios, nunca cambia
    la experiencia de quienes ya estaban dentro.
    """
    digest = hashlib.sha256(f"{flag}:{user_id}".encode()).hexdigest()
    bucket = (int(digest[:8], 16) / 0xFFFFFFFF) * 100
    return bucket < porcentaje
```

Fíjate en que el hash incluye el nombre del flag. Sin eso, el "primer 5 %" sería siempre el mismo grupo de usuarios para todos tus flags, y ese grupo acumularía todo el riesgo de todas tus funcionalidades nuevas.

Con el bucketing resuelto, la secuencia clásica funciona bien: usuarios internos -> 1 % -> 5 % -> 25 % -> 50 % -> 100 %, avanzando solo cuando las métricas clave (errores, latencia, conversión) se mantienen estables en cada escalón. Define antes de empezar qué métricas vigilas y qué umbral te hace retroceder; decidirlo durante el incidente es decidirlo mal.

Kill switches: el botón de apagado que agradecerás a las 3 de la mañana

Un ops flag bien colocado convierte un incidente de madrugada en un clic. La idea: envolver dependencias no críticas en un flag que, al apagarse, activa una degradación controlada.

```
async function getRecommendations(userId: string) {
  const enabled = await client.getBooleanValue(
    'ops-recommendations-enabled',
    true, // por defecto encendido: es un kill switch, no un release flag
    { targetingKey: userId },
  );

  if (!enabled) return FALLBACK_POPULAR_ITEMS; // degradación controlada

  try {
    return await recommendationService.fetch(userId);
  } catch {
    return FALLBACK_POPULAR_ITEMS;
  }
}
```

Si el servicio de recomendaciones empieza a fallar o a degradar la latencia global, se apaga el flag y la página sigue funcionando con contenido popular. Nada de despliegues de emergencia. Este patrón combina de maravilla con circuit breakers: el breaker reacciona automáticamente, el kill switch te da control manual explícito.

Errores comunes que anulan los beneficios

Deuda de flags. El error número uno. Cada flag es un `if` que duplica los caminos de ejecución; diez flags activos en un mismo servicio son, en teoría, 1.024 combinaciones posibles. Los flags de release y experimento deben nacer con fecha de caducidad y dueño asignado.

Valores por defecto peligrosos. Si el backend de flags se cae y tus defaults encienden funcionalidad a medio hacer, tienes un incidente doble. El default es tu comportamiento en el peor momento: elígelo con esa mentalidad.

Flags anidados y dependientes. Cuando el flag B solo tiene sentido si A está encendido, has creado un grafo de dependencias invisible. Si no puedes evitarlo, documenta la relación y testea las combinaciones

válidas explícitamente.

Evaluar en el cliente lo que protege algo de valor. Ya mencionado, pero se repite tanto que merece su propio punto: el cliente puede mentir. Servidor, siempre, para lo sensible.

Usar flags como configuración. Un timeout o una URL de servicio no son flags: son configuración. Mezclarlos en el mismo sistema convierte tu panel de flags en un vertedero donde nadie distingue lo que se puede tocar de lo que no.

Buenas prácticas que marcan la diferencia

- **Nombres que se explican solos**: `checkout-nuevo-flujo-pago` dice más que `flag-23`. Incluye el área y, si tu convención lo permite, el tipo (`ops-`, `exp-`).
- **Cada flag tiene dueño y caducidad**: sin un responsable y una fecha objetivo de eliminación, ningún flag temporal se limpia jamás.
- **Ritual de limpieza**: reserva un hueco recurrente (por ejemplo, en cada sprint) para eliminar flags al 100 % o al 0 %. Algunos equipos generan tickets automáticos cuando un flag supera su fecha.
- **Testea ambos caminos**: mientras el flag exista, la CI debe ejercitar el código con el flag encendido y apagado, al menos en los flujos críticos.
- **Auditoría y control de acceso**: cambiar un flag en producción es un cambio en producción. Registro de quién cambió qué y cuándo, y permisos acordes.

Cómo funciona un sistema de flags por dentro

Antes de acumular más patrones, conviene entender la arquitectura que estás comprando. Todo sistema de flags serio separa dos planos. El **plano de control** es donde se definen y cambian los flags: un panel, una API o un repositorio Git. El **plano de datos** es donde se evalúan: dentro de tu proceso, en cada petición, miles de veces por segundo. La mayoría de los errores de latencia y resiliencia nacen de confundir los dos.

Existen dos modelos de evaluación, y la diferencia importa más que cualquier lista de funcionalidades de un proveedor. Con **evaluación in-process**, el SDK descarga el conjunto de reglas una vez, lo mantiene en memoria y evalúa localmente. Una evaluación es una búsqueda en un hash más unas comprobaciones de reglas: microsegundos, sin red. Las actualizaciones llegan en segundo plano por polling o streaming. Así funcionan la mayoría de los SDKs de servidor, y es lo que quieres en una ruta caliente.

Con **evaluación remota**, cada llamada pregunta la decisión a un servicio externo (un agente, un sidecar o la API del proveedor). Es más simple de razonar y mantiene las reglas fuera de tu proceso, pero ahora hay un salto de red dentro de tu ruta de petición, y necesitas caché, timeouts y un fallback. La evaluación remota está bien para funciones edge y clientes; en un backend con tráfico exige cuidado.

De aquí salen dos reglas directas. Primera: **nunca bloquees una petición esperando una llamada de red al backend de flags**. Evalúa desde memoria, o contra una caché local con un timeout estricto. Segunda: en el arranque, bloquea una sola vez mientras el SDK se inicializa — con un timeout de un segundo o dos — y si la inicialización falla, sirve los valores por defecto y sigue arrancando. Un servicio que no puede arrancar porque el proveedor de flags está caído tiene las dependencias invertidas.

Por último, trata la configuración de flags como lo que es: **datos que cambian el comportamiento de producción**. Un cambio de flag es un despliegue disfrazado. Merece lo mismo que un despliegue — traza de auditoría, revisión para cambios arriesgados, posibilidad de comparar y revertir — aunque viaje por un panel en vez de por un pipeline.

flagd en la práctica: un backend de flags open source con filosofía Unix

Si te gusta el modelo de OpenFeature pero no quieres un contrato SaaS para empezar, **flagd** es la respuesta de la CNCF: un demonio pequeño que lee definiciones de flags desde ficheros, endpoints HTTP, fuentes de sincronización gRPC o recursos personalizados de Kubernetes, evalúa reglas de targeting y sirve decisiones a los SDKs de OpenFeature. Sin base de datos, sin interfaz, sin cuentas. Los flags viven en un documento JSON versionado en Git, lo que significa que las pull requests, las revisiones y el historial salen gratis — flags como GitOps.

Una definición completa con targeting y rollout progresivo tiene esta pinta:

```
{
  "$schema": "https://flagd.dev/schema/v0/flags.json",
  "flags": {
    "new-payment-flow": {
      "state": "ENABLED",
      "variants": { "on": true, "off": false },
      "defaultVariant": "off",
      "targeting": {
        "if": [
          { "in": [ { "var": "plan" }, ["internal", "beta"] ] },
          "on",
          {
            "fractional": [
              { "var": "targetingKey" },
              ["on", 25],
              ["off", 75]
            ]
          }
        ]
      }
    }
  }
}
```

Léelo de dentro hacia fuera. `variants` nombra los valores posibles y `defaultVariant` es lo que recibes si no hay coincidencia de targeting. El bloque `targeting` es JsonLogic: si el atributo `plan` del usuario es `internal` o `beta`, siempre recibe `on`. El resto cae en el operador `fractional`, que aplica murmurhash3 sobre la `targetingKey` y asigna el 25% de los usuarios a `on` y el 75% a `off`. El hash es determinista, así que el mismo usuario recibe la misma variante en cada petición y en cada instancia — exactamente la propiedad de bucketing que vimos antes, implementada por ti.

En Kubernetes el despliegue habitual es un sidecar, para que la evaluación se quede en localhost:

```
containers:
  - name: app
    image: my-service:1.42.0
  - name: flagd
    image: ghcr.io/open-feature/flagd:latest
    args:
      - start
      - --uri
      - file:/etc/flagd/flags.json
    volumeMounts:
      - name: flags
        mountPath: /etc/flagd
```

Y la aplicación se conecta mediante el provider de flagd — cambias una línea y todo lo demás de este artículo sigue funcionando:

```
# pip install openfeature-sdk openfeature-provider-flagd
from openfeature import api
from openfeature.contrib.provider.flagd import FlagdProvider

api.set_provider(FlagdProvider(host="localhost", port=8013))
```

flagd también ofrece un **provider in-process** para varios lenguajes: en lugar de hablar con un demonio, tu servicio consume la fuente de sincronización directamente y evalúa en memoria. Los mismos ficheros de flags, sin sidecar, evaluaciones en microsegundos. Para equipos que ya hacen GitOps, flagd más un ConfigMap sincronizado por Argo CD es una plataforma de flags sorprendentemente completa a precio cero.

Diseñar el contexto de evaluación (y mantener el PII fuera)

Toda decisión de targeting de este artículo cuelga del **contexto de evaluación**: la bolsa de atributos que pasas con cada evaluación. Merece más atención de diseño de la que suele recibir, porque un contexto descuidado produce bucketing inconsistente, segmentos rotos y problemas de privacidad — todos silenciosos.

El campo más importante es la `targetingKey`. Debe ser **estable para la unidad que quieres que sea consistente**. Si un usuario con sesión iniciada debe ver la misma variante en web y en móvil, la clave es el ID de usuario, no el de sesión ni el de dispositivo. Para tráfico anónimo, genera un ID anónimo duradero, guárdalo en una cookie de primera parte o en el almacenamiento local, y sigue usándolo tras el registro para los flags que se evaluaron con él en mitad de un experimento — si no, los usuarios cambian de variante en cuanto inician sesión y tus métricas se emborronan.

Más allá de la clave, mantén el contexto **pequeño y deliberado**. Plan, país, versión de la app, plataforma y un puñado de atributos de segmento cubren casi cualquier regla real de targeting. Resiste la tentación de volcar el objeto de usuario entero. Por dos razones. Primera, consistencia: si el servicio de checkout envía `plan: "pro"` y el de facturación envía `plan: "PRO"`, el mismo usuario coincide con reglas distintas en servicios distintos. Centraliza la construcción del contexto en una función pequeña por servicio, y acuerda los nombres de atributos entre equipos — trata el contexto como un pequeño esquema, versionado en tu documentación.

```
def build_flag_context(user, request) -> EvaluationContext:
    """Única fuente de verdad para los atributos de targeting."""
    return EvaluationContext(
        targeting_key=str(user.id),
        attributes={
            "plan": user.plan.lower(),
            "country": user.country_code, # ISO 3166-1 alfa-2
            "app_version": request.headers.get("x-app-version", "unknown"),
            "platform": request.headers.get("x-platform", "web"),
        },
    )
```

Segunda razón: **privacidad**. Los atributos del contexto viajan al backend de flags — un tercero, si usas SaaS. Los correos, nombres y cualquier dato sensible no pintan nada ahí; las reglas de targeting casi nunca los necesitan, y un ID numérico o hasheado hace el bucketing igual de bien. Si una regla parece necesitar PII en crudo ("usuarios cuyo correo termina en @granCliente.com"), modélala como un atributo de segmento calculado en tu lado (`is_gran_cliente: true`) en vez de enviar el correo. Tu responsable de protección de datos te lo agradecerá, y de paso las reglas quedan más rápidas y claras.

Rollouts avanzados: rings, canarios y métricas guardarraíl

Un deslizador de porcentaje es el principio, no el final, de la entrega progresiva. Los equipos maduros estructuran los rollouts en **rings**: grupos concéntricos ordenados por radio de impacto. Una secuencia típica: primero empleados, luego clientes beta que se apuntaron, luego el 1% del tráfico general, luego 10%, 50% y 100%. Los primeros rings no son muestras estadísticas — son pruebas de humo con humanos reales que abrirán un bug en vez de marcharse. La fase de porcentajes es donde vigilas métricas.

Mantén nítida la diferencia con un **despliegue canario**. Un canario desplaza tráfico entre dos versiones del binario completo en la capa de infraestructura (decide un balanceador o una service mesh). Un rollout con flag desplaza tráfico entre dos rutas de código dentro de un mismo binario (decide tu código, por usuario). Se complementan de maravilla: el canario caza crashes, fugas de memoria y regresiones de latencia del build nuevo; el rollout con flag aísla el comportamiento de una funcionalidad concreta. Lo que no debes hacer es usar uno para sustituir al otro — un canario a nivel de mesh no puede dar a un usuario concreto una experiencia consistente entre peticiones, y un flag no te salva de un build que hace segfault.

Cada paso de porcentaje necesita dos cosas más: un **tiempo de horneado** (bake time) y **métricas guardarraíl**. El bake time es el período mínimo que un paso debe mantenerse antes de avanzar — suficiente para cubrir tus ciclos de tráfico; un paso que solo vio tráfico de madrugada demuestra poco. Los guardarraíles son el puñado de métricas que definen "algo va mal": tasa de errores, latencia p99 y una métrica de negocio pegada a la funcionalidad (conversión de checkout para un flujo de pago). Define los umbrales *antes* de empezar, divide cada dashboard por variante del flag, y deja la regla por escrito: si un guardarraíl se rompe, el flag va a 0% primero y las preguntas vienen después.

Los equipos que hacen esto cada semana acaban automatizándolo. La lógica no es sofisticada — es un bucle:

```
# Esbozo de una puerta de rollout automatizada (cron o paso de pipeline)
STEPS = [1, 5, 10, 25, 50, 100] # porcentaje de tráfico
BAKE = timedelta(hours=6) # tiempo mínimo por paso

def advance_rollout(flag, metrics, now):
    step = flag.current_step
    if now - flag.step_started_at < BAKE:
        return "esperando"
    if metrics.error_rate("on") > metrics.error_rate("off") * 1.2:
        set_percentage(flag.key, 0) # primero apagar, luego investigar
        page_oncall(flag.key, "guardarrail roto: tasa de errores")
        return "revertido"
    if metrics.p99_latency("on") > metrics.p99_latency("off") * 1.3:
        set_percentage(flag.key, 0)
        page_oncall(flag.key, "guardarrail roto: latencia")
        return "revertido"
    next_pct = STEPS[STEPS.index(step) + 1]
    set_percentage(flag.key, next_pct)
    return "avanzado a " + str(next_pct) + "%"
```

Uses la automatización de tu proveedor o veinte líneas propias, la idea es la misma: **rollouts gobernados por umbrales y tiempos explícitos**, no por quien se acordó de mirar el dashboard después de comer.

Experimentos A/B sobre flags

Los flags de experimento parecen flags de release con dos variantes, pero la disciplina que los rodea es distinta, e ignorar esa diferencia es la manera en que los equipos acaban "haciendo A/B testing" hasta llegar a conclusiones que son puro ruido.

La primera distinción es **asignación frente a exposición**. La asignación ocurre cuando el hash de bucketing coloca al usuario en una variante. La exposición ocurre cuando el usuario experimenta de verdad la diferencia — llegó a la pantalla donde se renderiza el nuevo checkout. Si analizas por asignación, cada usuario que nunca tocó la funcionalidad diluye tus métricas hacia el efecto cero. Registra un **evento de exposición** en el momento en que se ejecuta el código dependiente de la variante, con la clave del flag, la variante y la targeting key, y analiza sobre usuarios expuestos. Los hooks de OpenFeature (siguiente sección) son el sitio limpio para emitir estos eventos sin ensuciar el código de negocio.

Segundo: comprueba el **sample ratio mismatch (SRM)** antes de creerte ningún resultado. Si tu división está configurada 50/50 pero observas 52/48 con una muestra grande, algo en la tubería está sesgado —

bots golpeando una variante, un redirect que pierde exposiciones, una caché sirviendo una sola ruta — y el experimento es inválido diga lo que diga la métrica. Un test chi-cuadrado contra la proporción esperada es una línea con scipy y debería ser automático:

```
from scipy.stats import chisquare

exposed = {"on": 50_412, "off": 48_103}
total = sum(exposed.values())
stat, p = chisquare(list(exposed.values()), [total/2, total/2])
if p < 0.001:
    raise RuntimeError("SRM detectado: no te fíes de este experimento")
```

Tercero, higiene estadística básica: fija **una métrica primaria** y el tamaño mínimo de muestra antes de empezar (cualquier calculadora online sirve), no pares en cuanto el p-valor baje de 0,05 — mirar de reojo (peeking) infla los falsos positivos de forma dramática — y trata las métricas secundarias como generadoras de hipótesis, no como conclusiones. Si vas a experimentar con regularidad, adopta una capa especializada como **GrowthBook** o el módulo de experimentación de tu plataforma de analítica sobre tus flags: se encargan del registro de exposiciones, los chequeos de SRM y la estadística secuencial para que cada equipo no los reinvente mal.

Y recuerda la regla del ciclo de vida con fuerza extra: un flag de experimento cuyo experimento terminó es deuda pura. Consolida la variante ganadora, borra el flag, archiva el análisis.

Flags en el frontend: React, parpadeo y bootstrapping

Todo lo anterior corría en el servidor, donde el SDK mantiene un conjunto de reglas y evalúa por petición. Los navegadores y las apps móviles invierten el modelo: el SDK de cliente evalúa (o descarga) los flags **para un usuario conocido**, cachea los resultados y reevalúa cuando el contexto cambia — login, mejora de plan, cambio de idioma. OpenFeature llama a esto el paradigma de contexto estático o de cliente.

Con el SDK de React el cableado es pequeño:

```
// npm install @openfeature/react-sdk
import { OpenFeature, OpenFeatureProvider, useFlag } from '@openfeature/react-sdk';

await OpenFeature.setContext({ targetingKey: user.id, plan: user.plan });
await OpenFeature.setProviderAndWait(new MyWebProvider());

function App() {
  return (
    <OpenFeatureProvider>
      <Checkout />
    </OpenFeatureProvider>
  );
}

function Checkout() {
  const { value: newFlow } = useFlag('new-payment-flow', false);
  return newFlow ? <NewCheckout /> : <ClassicCheckout />;
}
```

El fallo clásico del frontend es el **parpadeo** (flicker): la página se renderiza con la variante por defecto, la respuesta del flag llega 300 ms después y la interfaz salta a la otra variante delante del usuario. Tres arreglos, por orden de preferencia. Haz bootstrap de los flags en el servidor: evalúa durante el renderizado en servidor e incrusta los valores en la página, de modo que el SDK de cliente arranque hidratado y el primer pintado ya sea correcto. Si no puedes hacer SSR, condiciona el renderizado a que el provider esté listo (el SDK de React expone hooks compatibles con suspense) y muestra tu estado de carga normal durante esos milisegundos. Como último recurso, acepta el valor por defecto el tiempo justo para evitar el salto — solo para flags cosméticos de bajo riesgo.

Dos notas de seguridad que ahorran dolores reales. Las claves de SDK de cliente son **públicas** — viajan en el bundle — así que solo deben desbloquear evaluación, nunca administración de flags; y cualquier flag que el cliente evalúe es visible y falsificable por el cliente. La regla de la sección de errores se aplica sin excepciones: las decisiones cosméticas y de UX pueden vivir en el cliente, pero todo lo que toque dinero, cuotas o permisos se decide en el servidor, y el frontend se limita a renderizar el resultado. Cuidado también con las fugas del payload: los nombres y reglas de flags sin lanzar pueden leerse en las respuestas de red, que es como más de un lanzamiento ha sido destripado por usuarios curiosos con las DevTools abiertas.

Testing con feature flags

Los flags multiplican las rutas de ejecución, y tu suite de tests tiene que seguir el ritmo sin multiplicarse hasta volverse inútil. La base es el **provider en memoria** que los SDKs de OpenFeature incluyen exactamente para esto: los tests fijan flags a valores conocidos sin red, sin mocks de clientes HTTP, sin dobles del SDK del proveedor.

```
import pytest
from openfeature import api
from openfeature.provider.in_memory_provider import InMemoryFlag, InMemoryProvider

def set_flags(**flags):
    memory_flags = {
        key: InMemoryFlag(
            "on" if enabled else "off",
            {"on": True, "off": False},
        )
        for key, enabled in flags.items()
    }
    api.set_provider(InMemoryProvider(memory_flags))

@pytest.mark.parametrize("new_flow", [True, False])
def test_checkout_funciona_en_ambas_rutas(new_flow):
    set_flags(**{"new-payment-flow": new_flow})
    result = checkout(user=make_user(), cart=make_cart())
    assert result.status == "confirmed"
```

El test parametrizado de arriba codifica la regla de oro: **mientras un flag exista, ambas rutas son código de producción**, y ambas se testean. En tests unitarios, parametriza sobre los flags que el código bajo prueba lee de verdad — no sobre todos los del sistema. La explosión combinatoria (diez flags, 1.024 combinaciones) solo es un riesgo real si dejas que flags sin relación se filtren en un mismo módulo; con flags bien acotados, el conjunto relevante por test se queda en uno o dos.

Para tests de integración y end-to-end, dos prácticas mantienen las suites deterministas. Primera: **fija todos los flags** en el entorno de test — una suite que hereda lo que diga el panel de staging ese día empezará a fallar aleatoriamente el día que alguien toque algo. Un provider controlado por variables de entorno o un fichero de flags versionado junto a los tests funciona bien. Segunda: añade un smoke test por *rollout activo* que recorra el flujo completo con el flag encendido, en staging, contra dependencias reales: la cobertura unitaria de ambas ramas no caza el bug de integración que solo dispara la ruta nueva.

Y cierra el círculo con el ciclo de vida: cuando un flag se elimina, sus parametrizaciones y los tests de la rama perdedora se borran en la misma pull request. Los tests de código que ya no existe son el origen de las suites "desactivadas temporalmente".

Observabilidad: saber qué hizo un flag en producción

En cuanto los flags controlan comportamiento, "¿qué versión tiene el usuario?" deja de tener una única respuesta por despliegue — tiene una respuesta **por petición**. Tu telemetría tiene que transportar esa respuesta, o te pasarás los incidentes adivinando qué variante produjo los errores que llenan tu pantalla.

OpenTelemetry define convenciones semánticas exactamente para esto: una evaluación de flag se registra como un evento llamado `feature_flag` con atributos como `feature_flag.key`, `feature_flag.provider.name`, `feature_flag.result.value` y `feature_flag.result.reason`. Los **hooks** de OpenFeature son el lugar natural para emitirlo — lo escribes una vez, y todas las evaluaciones del servicio quedan anotadas sin tocar el código de negocio:

```
from openfeature import api
from openfeature.hook import Hook
from opentelemetry import trace

class OtelFlagHook(Hook):
    def after(self, hook_context, details, hints):
        span = trace.get_current_span()
        if span and span.is_recording():
            span.add_event("feature_flag", {
                "feature_flag.key": details.flag_key,
                "feature_flag.provider.name": hook_context.provider_metadata.name,
                "feature_flag.result.value": str(details.value),
                "feature_flag.result.reason": str(details.reason or ""),
            })

api.add_hooks([OtelFlagHook()])
```

Con eso en marcha, tres vistas se vuelven posibles y deberías montar las tres. **Trazas**: cualquier traza lenta o fallida muestra qué variantes vio la petición — el misterio de "solo falla a algunos usuarios" suele disolverse en el acto. **Métricas divididas por variante**: tasa de errores e histogramas de latencia con una etiqueta de variante para los flags en rollout activo (ojo a la cardinalidad — etiqueta con la variante del flag o los dos flags que están en rollout, no con todos). **Anotaciones de cambios**: cada cambio de flag aterriza como una marca estilo despliegue en tus dashboards, porque "¿qué cambió a las 14:32?" es la primera pregunta de todo incidente, y un flip de flag es la respuesta más veces de las que los equipos esperan.

Registra también la razón de la evaluación. Los valores responden *qué* recibió el usuario; la razón (`TARGETING_MATCH`, `SPLIT`, `DEFAULT`, `ERROR`) responde *por qué* — y un pico súbito de razones `DEFAULT` o `ERROR` es tu señal más temprana de que el backend de flags está degradado y todo el mundo está recibiendo en silencio el comportamiento de fallback. Alerta sobre ese ratio; es el chequeo de salud del propio sistema de flags.

Resiliencia: cuando el backend de flags se cae

Tu backend de flags tendrá un mal día. Cuando llegue, la diferencia entre un no-evento y un incidente global se decidió semanas antes, así que decide con intención.

Los valores por defecto son tu verdadero plan de desastre. Cada evaluación lleva un valor por defecto, y ese valor es el comportamiento completo de tu sistema cuando nada más funciona. Audítalos: el default seguro de un flag de release es la ruta antigua; el de un kill switch, lo que mantenga vivo el flujo principal; el de un flag de permisos, denegar. Un grep rápido buscando evaluaciones con default `True` sobre funcionalidad sin lanzar es un ejercicio de quince minutos que ha evitado incidentes reales.

Cachea con agresividad, caduca con reticencia. Los SDKs in-process ya mantienen el conjunto de reglas en memoria; la pregunta es qué pasa cuando se queda obsoleto porque las actualizaciones no pueden llegar. La respuesta debería ser "seguir sirviendo el conjunto obsoleto" — unos flags rancios casi siempre son mejores que unos flags por defecto, porque lo rancio preserva el estado de rollout que los usuarios ya veían. En el cliente, persiste los últimos valores conocidos para que un usuario que reabre la app sin conexión tenga continuidad en vez de defaults. Para evaluación remota, pon delante una caché de lectura con TTL estricto y política stale-while-revalidate, y trata el servicio de flags como cualquier otra dependencia: timeouts de decenas de milisegundos y un circuit breaker para que un backend de flags lento no se convierta en tu problema de latencia.

Bootstrap desde fichero. El patrón más robusto para servicios críticos: incluye una instantánea de los valores de flags en el artefacto de despliegue (o en un ConfigMap) como fallback de inicialización. El orden de arranque queda: intenta el backend vivo con timeout corto; si falla, carga la instantánea; converge a los valores vivos cuando vuelva la conectividad. Tu servicio ahora arranca correctamente aunque el proveedor de flags esté completamente caído durante tu rollback de las 3 de la mañana.

```
import json, logging

def init_flags(timeout_seconds: float = 2.0) -> None:
    try:
        provider = FlagdProvider(host="localhost", port=8013,
                                 deadline_ms=int(timeout_seconds * 1000))
        api.set_provider(provider) # bloquea hasta estar listo o agotar plazo
    except Exception:
        logging.exception("backend de flags no disponible, usando instantánea")
        with open("/etc/flags/snapshot.json") as f:
            api.set_provider(InMemoryProvider(parse_snapshot(json.load(f))))
```

Y después, **ensáyalo**. Un ejercicio de caos trimestral — bloquea el endpoint del backend de flags en staging y observa qué hace la flota — verifica la cadena completa: defaults sensatos, cachés sirviendo datos rancios, ruta de arranque tirando de instantánea, y la alerta de `reason=ERROR` de la sección de observabilidad disparándose de verdad. Los equipos se llevan una sorpresa fiable la primera vez que lo ejecutan.

Gobernanza: nombres, propiedad y matar la deuda de flags

La deuda de flags era el error número uno, así que merece un sistema, no un propósito de año nuevo. Los equipos que se mantienen limpios tratan los flags como tratan las dependencias: inventariados, con dueño y con caducidad.

Nombra para el grep. Una convención como `equipo.propósito-nombre-corto` (`checkout.release-new-payment-flow`, `platform.ops-disable-recommendations`) hace visible la propiedad en cada llamada de evaluación y convierte la búsqueda de limpieza en algo trivial. Impón kebab-case y prohíbe los renombrados — un flag renombrado es un flag nuevo con el historial huérfano.

Todo flag nace con metadatos: un dueño (un equipo, no una persona que cambiará de trabajo), un tipo (release, experimento, ops, permiso) y — para los de release y experimento — una fecha de caducidad. La mayoría de plataformas soportan etiquetas o campos personalizados; si la tuya no, un registro `flags.yaml` en el repositorio funciona perfectamente. Lo que importa es que los datos existan donde un script pueda leerlos, porque la única imposición que funciona es la automatizada:

```
# Job de CI: falla el build cuando un flag de release supera su caducidad
import sys, yaml, datetime

registry = yaml.safe_load(open("flags.yaml"))
today = datetime.date.today()
expired = [
    f["key"] for f in registry["flags"]
    if f["type"] in ("release", "experiment")
    and datetime.date.fromisoformat(f["expires"]) < today
]
if expired:
    print("Flags caducados: elimínalos o renuévalos con un motivo:")
    for key in expired:
        print(" -", key)
    sys.exit(1)
```

Un build que falla con "elimina este flag" convierte la deuda en una tarea visible y asignable. Acompáñalo de una **revisión de flags** mensual de quince minutos por equipo — se recorre la lista y para cada flag se

pregunta "¿sigue haciendo falta?" — y de herramientas de eliminación donde la base de código lo justifique: las herramientas del linaje de Piranha (Uber) y los frameworks de reescritura estructurada como OpenRewrite pueden extirpar automáticamente un flag y su rama muerta en una base de código grande una vez decidido el ganador.

Vigila un número: **flags activos por servicio**, con su tendencia en un dashboard que alguien mire de verdad. Es una métrica de salud perfectamente honesta — plana o descendente es un equipo con control; monotónicamente creciente son 1.024 rutas de ejecución esperando la combinación que nadie probó.

Caso práctico: lanzar un nuevo flujo de pago de principio a fin

Ejecutemos el manual completo una vez, sobre el ejemplo que este artículo lleva usando. El equipo es dueño del checkout; el proyecto sustituye la orquestación del flujo de pago por una integración nueva con el proveedor. Dinero real, así que paranoia máxima.

Semana 0 — antes de mergear código. Se crea el flag `checkout.release-new-payment-flow` con dueño `team-checkout`, tipo `release`, caducidad a 90 días, default `off` en todas partes. Los guardarraíles quedan escritos en el documento de rollout: tasa de errores de pago, latencia p99 del checkout y tasa de autorización del proveedor de pagos, cada una con umbral numérico y la regla permanente — si se rompe, primero 0% y después la discusión. Los dashboards reciben la división por variante usando el hook de OpenTelemetry de antes.

Semanas 1–3 — merge en oscuro, tests de ambas rutas. El flujo nuevo se mergea a main detrás del flag en pull requests pequeñas, desplegado continuamente, ejecutado por nadie. La suite parametrizada corre ambas rutas en cada commit; un smoke test de staging ejecuta el flujo nuevo contra el sandbox del proveedor cada noche. Dos bugs de integración mueren aquí en silencio, delante de cero clientes.

Semana 4 — ring interno. Regla de targeting: `plan in ["internal"]` -> `on`. Los empleados ejercitan el flujo con tarjetas reales sobre una cuenta de comercio de prueba. Al segundo día aflora un bug de redondeo de divisas — de los que los tests unitarios con importes bonitos no cazan jamás. Se arregla, se despliega, el flag ni se toca.

Semana 5 — ring beta y 1%. Entran los clientes beta apuntados. Tras tres días tranquilos, `fractional` envía el 1% del tráfico general a `on`. Bake time: mínimo 24 horas, cubriendo un ciclo diario completo incluida la punta de la tarde.

Semana 6 — el guardarraíl se gana el sueldo. Al 25%, la tasa de autorización de la variante `on` cae un 0,8% por debajo de `off` — dentro del margen de la alerta pero visible en el dashboard dividido. El ingeniero de guardia pone el flag a 0%. Tiempo desde la sospecha hasta la mitigación completa: unos cuarenta segundos, sin despliegue, sin rollback, sin llamada de incidente. Causa raíz: la integración nueva formateaba distinto un campo opcional y un banco regional lo rechazaba. El arreglo de tres líneas sale a la mañana siguiente; el rollout se reanuda al 10% ese mismo día.

Semana 7 — 50% y 100%. Las métricas aguantan. El rollout se completa; el flujo antiguo sigue existiendo detrás del flag como salida de emergencia instantánea mientras crece la confianza.

Semanas 8–9 — la parte que todo el mundo se salta. Dos semanas al 100% con métricas limpias. Después, la PR de limpieza: se borran la evaluación del flag, el flujo antiguo, sus tests y su división en dashboards; se cierra la entrada del registro; se programa la revocación de las credenciales antiguas del proveedor. El flag vivió 65 días — dentro de su caducidad de 90 — y el chequeo de CI nunca tuvo que dispararse.

Nada en esa historia es heroico. Esa es la gracia: los feature flags convierten lo que habría sido un evento de lanzamiento de alto riesgo en nueve semanas de pasos aburridos y reversibles.

Elegir proveedor en 2026

Como todo lo anterior está escrito contra OpenFeature, el proveedor es un detalle intercambiable — que es justo lo que permite elegirlo con calma. Las preguntas que de verdad importan:

Modelo de evaluación. ¿El SDK de servidor evalúa in-process (ruta rápida, funciona durante caídas del proveedor) o llama por red en cada evaluación? Para tu tráfico y presupuesto de latencia, este es el primer filtro. **Alojamiento.** Comodidad SaaS frente a autoalojamiento por residencia de datos y control de costes. **Experimentación.** Si necesitas análisis A/B de verdad, ¿la plataforma registra exposiciones, chequea SRM y hace estadística secuencial, o solo divide porcentajes? **Gobernanza.** Log de auditoría, RBAC, aprobaciones para cambios en producción, metadatos de ciclo de vida — la lista enterprise, que importa con cincuenta ingenieros aunque no importara con cinco. **Eje de precios.** Por asiento, por MAU o por evaluación; cada uno castiga un patrón de crecimiento distinto, así que modela el tuyo antes de firmar.

La guía de campo abreviada: **LaunchDarkly** es la referencia gestionada y madura — actualizaciones por streaming, gobernanza fuerte, precio acorde. **Unleash** y **Flagsmith** son las plataformas open source consolidadas que puedes autoalojar, con planes gestionados para cuando te canses de operarlas. **GrowthBook** lidera con la experimentación sobre flags. **PostHog** empaqueta flags con analítica de producto, atractivo si quieres una sola herramienta para ambas cosas. Y **flagd**, como vimos arriba, es la opción GitOps-nativa sin panel y sin factura por asiento. Una progresión sensata para muchos equipos: empezar con flagd o Unleash mientras las necesidades son simples, y dejar que OpenFeature mantenga barata la salida si los superas — la migración es un cambio de provider y de configuración, no una reescritura de cada punto de llamada.

Flags y trunk-based development: mergear trabajo sin terminar

Los feature flags y el trunk-based development son dos mitades de un mismo flujo de trabajo, y merece la pena hacer explícita la conexión porque cambia cómo escribes código, no solo cómo lanzas. Trunk-based development significa que todo el mundo mergea a main en incrementos pequeños, a diario o mejor; las ramas de funcionalidad de larga vida — y sus conflictos de merge de una semana — desaparecen. La objeción obvia es "pero mi funcionalidad lleva seis semanas". Los flags son la respuesta: el trabajo incompleto se mergea continuamente, protegido por un flag que lo mantiene apagado en producción.

La técnica que hace mergeable por rebanadas un cambio grande es **branch by abstraction**. En vez de un `if` en cincuenta puntos de llamada, introduce una costura — una interfaz, una factoría, una estrategia — y pon la decisión del flag en exactamente un sitio:

```
class PaymentFlow(Protocol):
    def execute(self, order: Order) -> PaymentResult: ...

def get_payment_flow(client, ctx) -> PaymentFlow:
    """El único punto de la base de código que lee este flag."""
    if client.get_boolean_value("checkout.release-new-payment-flow", False, ctx):
        return NewPaymentFlow()
    return ClassicPaymentFlow()
```

Ahora la implementación nueva crece detrás de la interfaz pull request a pull request, los revisores ven diffs pequeños, y la limpieza final borra una rama de una factoría en vez de cazar condicionales por toda la base de código. Como regla general, **un flag debería leerse en un solo sitio**; cuando encuentras la misma clave evaluada en cinco módulos, es el código diciéndote que quiere una abstracción.

Dos hábitos completan el flujo. Mantén la *comprobación del flag en la capa más alta razonable* — decide una vez por petición en el punto de entrada y pasa la decisión (o la implementación elegida) hacia abajo, en vez de reevaluar en cada helper, lo que invita a la inconsistencia de un flag cambiando a mitad de petición. Y resiste la deriva hacia la *compilación* condicional: los flags controlan comportamiento en tiempo de ejecución precisamente para que un único artefacto pueda hacer ambas cosas; en el momento en que

bifurcas builds por variante has reinventado las ramas de release con más pasos.

Móvil y edge: flags donde desplegar es lento o queda lejos

El valor de un flag es proporcional a lo doloroso que sea el rollback alternativo — lo que convierte al móvil en el caso más fuerte de todos. Un mal despliegue web se revierte en minutos; una mala release de app son días de revisión de la tienda más semanas de usuarios que nunca actualizan. Los equipos que publican apps lo tratan como ley: **todo cambio arriesgado en una release móvil sale apagado**, detrás de un flag que puede activarse desde el servidor sin un binario nuevo. La app evalúa flags con el paradigma de cliente — descarga para este usuario, cachea localmente, persiste entre arranques — y esa cachépersistida hace doble servicio como comportamiento offline: un usuario en modo avión recibe las últimas variantes conocidas, no los defaults.

El móvil añade dos restricciones que conviene diseñar. Primera: **el desfase de versiones es permanente**: hay usuarios ejecutando cada versión de la app de los últimos dos años, así que el targeting de un flag suele necesitar una condición de `app_version`, y un flag no puede eliminarse hasta que el último binario que lo lee se haya extinguido en la práctica. Presupuesta la vida de los flags móviles en trimestres, no en semanas. Segunda: sorpresas en la revisión: las políticas de las tiendas miran mal las funcionalidades que cambian materialmente la app después de la revisión, así que los flags ahí deben modular y proteger funcionalidad, no colar un producto distinto.

En el otro extremo, la **evaluación en el edge** mueve la decisión al worker del CDN antes de que la petición toque tu origen: enrutado A/B, desplazamiento gradual de tráfico entre versiones del origen, kill switches instantáneos para rutas enteras. Aplican las mismas reglas con presupuestos más estrictos — el worker no puede permitirse una llamada por petición a una API de flags, así que los flags llegan como instantánea empujada (muchos proveedores se integran con los almacenes clave-valor del edge exactamente para esto), la evaluación es una búsqueda local y el contexto de targeting se limita a lo que la propia petición lleva: cookies, cabeceras, geografía. Mantén los flags de edge pocos y gruesos; la personalización fina sigue perteneciendo detrás del origen, donde vive tu contexto real.

Migrar una base de código existente a OpenFeature

La mayoría de los equipos no parten de cero — llegan con un SDK de proveedor llamado directamente desde doscientos sitios, o con un diccionario casero `settings.FEATURES` al que le salieron dientes. La migración a OpenFeature es piadosamente mecánica, porque el estándar se diseñó para envolver lo que ya tienes.

Paso uno: **envuelve, no reescribas**. Instala el provider de OpenFeature para tu proveedor actual (existen providers oficiales para las plataformas principales) o escribe uno fino sobre tu almacén casero — un provider es una clase con un puñado de métodos resolve:

```
from openfeature.provider import AbstractProvider
from openfeature.flag_evaluation import FlagResolutionDetails, Reason

class LegacySettingsProvider(AbstractProvider):
    """Sirve flags desde el almacén casero durante la migración."""
    def resolve_boolean_details(self, flag_key, default_value, context=None):
        if flag_key in legacy_store:
            return FlagResolutionDetails(
                value=legacy_store.get(flag_key, context),
                reason=Reason.TARGETING_MATCH,
            )
        return FlagResolutionDetails(value=default_value, reason=Reason.DEFAULT)
```

Paso dos: migra los puntos de llamada **de forma oportunista** — cada fichero tocado cambia su llamada directa al SDK por el cliente de OpenFeature, el código nuevo solo usa OpenFeature, y una regla de lint

sencilla (prohibir importar el SDK del proveedor fuera del módulo del provider) evita regresiones. No hay fin de semana de big bang; los dos estilos conviven sin hacerse daño durante meses.

Paso tres, cuando los puntos de llamada están limpios: **cambia el provider** en una línea del arranque, ejecuta brevemente ambas pilas en modo sombra si lo justifica el riesgo (evalúa contra las dos, registra las discrepancias, sirve la respuesta antigua) y desmantela. Los equipos que lo han hecho describen el anticlímax como el objetivo mismo — el día que cambias de proveedor es el día en que nadie se entera.

Notas de seguridad que merecen sección propia

Los flags cambian el comportamiento de producción sin despliegue, lo que convierte el panel de flags en un **plano de control privilegiado** — trátalo como tal. Los cambios de flags de producción merecen el mismo rigor que el acceso a producción: SSO con MFA, permisos por rol (lectores, editores por proyecto, aprobadores para producción) y flujos de aprobación para los flags marcados como críticos. El log de auditoría no es teatro de cumplimiento: es la respuesta a "¿qué cambió a las 14:32?" en tu próximo incidente, y debería fluir al mismo sitio que tus eventos de despliegue.

Piensa en el **radio de impacto de un flag comprometido**. Un atacante con acceso al panel puede apagar funcionalidades de seguridad, exponer funcionalidad sin lanzar o redirigir porcentajes de tráfico — en silencio, sin pipeline que dé la alarma. Las mitigaciones son directas: roles de mínimo privilegio, alertas sobre cambios en flags etiquetados como relevantes para seguridad, y doble aprobación para los kill switches de funciones protectoras (un flag que bypasea el WAF, por ejemplo).

Las claves de servidor, las de cliente y las credenciales de relay son **secretos de grados distintos**: una clave de SDK de servidor puede leer tu conjunto completo de reglas, incluidos los nombres de flags sin lanzar y sus reglas de targeting (que pueden filtrar roadmap o nombres de clientes), así que vive en tu gestor de secretos con rotación, nunca en un repositorio ni en un bundle de cliente. Y en productos B2B multi-tenant, recuerda que las reglas que nombran clientes concretos (`org_id in [...]`) son visibles para cualquiera con acceso de lectura al panel y, si se evalúan en el cliente, potencialmente para los propios clientes — un nombre de segmento como `tier-enterprise-pilot` filtra menos que la lista de quién está dentro.

Checklist de producción

Antes de que tu próxima funcionalidad salga detrás de un flag, recorre esta lista. Comprime todo lo anterior en las preguntas que, de un modo u otro, se acabarán haciendo — ahora, o en la revisión del incidente.

- **Tipo y ciclo de vida declarados:** el flag tiene tipo (release, experimento, ops, permiso), equipo dueño y — salvo que sea permanente por diseño — fecha de caducidad registrada donde un script pueda leerla.
- **El nombre sigue la convención** (`equipo.propósito-nombre`), es buscable con grep y no se renombrará jamás.
- **Default elegido para el peor momento:** si el backend de flags no responde, el valor por defecto es el comportamiento con el que puedes vivir. Para funcionalidad nueva, casi siempre apagado.
- **Evaluación en servidor para todo lo valioso:** las decisiones de dinero, cuotas, permisos y precios se toman en el backend; el cliente solo las renderiza.
- **Un solo punto de lectura:** el flag se evalúa en un único sitio (factoría, estrategia, punto de entrada) y la decisión fluye hacia abajo — sin reevaluaciones dispersas.
- **Ambas rutas testeadas:** tests unitarios parametrizados cubren on y off; un smoke test de staging ejercita la ruta nueva de extremo a extremo; los entornos de test fijan los valores de los flags.
- **Bucketing determinista** sobre una targeting key estable, para que los usuarios conserven su variante entre peticiones, instancias y sesiones.

- **Plan de rollout escrito antes de empezar:** orden de rings, pasos de porcentaje, bake time por paso, y métricas guardarraíl con umbrales numéricos y regla de reversión pactada.
- **Dashboards divididos por variante** para los flags en rollout, evaluaciones anotadas en las trazas, y una alerta sobre la tasa de razones `DEFAULT`/`ERROR`.
- **Modo de fallo ensayado:** el servicio arranca y se comporta con sensatez con el backend de flags caído — bootstrap por instantánea, caché rancia, defaults sanos — y lo has visto hacerlo en staging con tus propios ojos.
- **Kill switch alcanzable a las 3 de la mañana:** la guardia sabe dónde está el botón de apagado, tiene permisos para pulsarlo y el log de auditoría registra quién lo pulsó.
- **Limpieza programada:** la tarea de eliminación existe en el backlog desde el día uno, y la CI falla cuando pasa la caducidad.

Preguntas frecuentes

¿En qué se diferencia un feature flag de la configuración normal?

Mecánicamente se solapan; la diferencia es intención y cadencia. La configuración describe cómo corre un despliegue (tamaños de pool, endpoints) y cambia poco, con los despliegues. Los flags describen qué comportamiento reciben los usuarios, cambian en tiempo de ejecución — a veces cada hora durante un rollout — y llevan targeting por usuario. La consecuencia operativa: los flags necesitan auditoría, contexto de targeting y propagación instantánea; la configuración necesita validación y fijado de versiones. Usar un fichero de configuración como sistema de flags pierde las mitades de runtime y targeting; usar flags como almacén de configuración entierra configuración real en un panel fuera de la revisión de código.

¿Cuántos flags son demasiados?

No hay número mágico; hay una forma. Docenas de flags activos en un servicio pueden ser sanos si la mayoría son de ops y permisos, con dueños y rutas ejercitadas. Diez flags de release de hace tres trimestres son insanos con cualquier total. Vigila la tendencia de flags activos por servicio y la distribución de edad de los de release y experimento — si el flag de release más viejo es más antiguo que tu becario más antiguo, la gobernanza ha fallado, diga lo que diga el total.

¿Los flags ralentizan mi aplicación?

La evaluación in-process es una búsqueda en memoria más unas comprobaciones de reglas — microsegundos, invisible al lado de una llamada a base de datos. Los costes que sí muerden son arquitectónicos: evaluación remota en una ruta caliente sin caché (salto de red por decisión), evaluar el mismo flag muchas veces por petición en vez de una, y cardinalidad de métricas sin límite por etiquetar todo con todos los flags. Los tres son evitables con los patrones de esta guía.

¿Deberían los flags decidir entitlements — lo que pueden usar los clientes de pago?

Los flags de permiso están bien como *mecanismo de entrega* del acceso por plan, y media industria los usa así. La trampa es dejar que la plataforma de flags se convierta en la *fuente de verdad* del estado de facturación. Los entitlements pertenecen a tu modelo de facturación y permisos, testeado y consistente; un flag puede cachear o reflejar ese estado para comprobaciones rápidas, pero cuando los dos discrepan, gana facturación. Si quitar tu proveedor de flags cambiaría quién tiene acceso a lo que pagó, la frontera está en el sitio equivocado.

¿Y los flags en jobs por lotes y workers en segundo plano?

El mismo SDK, dos ajustes. Evalúa **una vez por job** (o por lote), no una vez por elemento — un millón de evaluaciones del mismo flag a mitad de job crean ruido y, peor, la posibilidad de que el comportamiento cambie a mitad de lote. Y elige la targeting key con intención: para jobs por cliente usa el ID del cliente, de

modo que el comportamiento del lote coincida con lo que ese cliente ve online; para jobs de infraestructura usa el nombre del job, que te da canarying determinista del propio job.

¿Puedo llevar toda mi respuesta a incidentes con flags de ops?

Los modos degradados detrás de flags de ops — desactivar recomendaciones, servir búsqueda cacheada, soltar carga no crítica — están entre los usos de mayor valor de toda la técnica. Dos condiciones los mantienen fiables: cada ruta degradada se ejercita con calendario (un game day mensual activando cada flag de ops en staging, como mínimo), porque un fallback sin probar es un rumor; y cada uno tiene una línea de runbook que dice cuándo activarlo y qué aspecto tiene "recuperado". Un flag de ops que nadie ha activado en un año debe asumirse roto hasta que se demuestre lo contrario.

Conclusión

Los feature flags son de esas herramientas cuyo coste de adopción es pequeño y cuyo efecto compuesto es enorme: despliegues más frecuentes y aburridos (en el buen sentido), incidentes que se resuelven con un interruptor y experimentos que no requieren infraestructura propia. La clave está en tratarlos con disciplina: tipos claros, defaults seguros, evaluación en servidor para lo sensible, y una rutina de limpieza que impida que la solución de hoy sea la deuda de mañana. Con OpenFeature como capa de abstracción, además, todo esto lo construyes una sola vez, con la libertad de cambiar de proveedor cuando lo necesites.

English Version

Why feature flags change the way you deploy

Deploying code and releasing a feature look like the same thing, but they are not. When both happen at once, every deploy is a gamble: if the feature misbehaves, your only way out is a full rollback, with all the rush, stress and risk that entails. Feature flags break that coupling. Code ships to production turned off, and you decide when, for whom and how fast it turns on.

That separation has enormous practical consequences: you can merge small, frequent branches without waiting for the feature to be complete (trunk-based development), test in production with internal users before opening the tap, and switch off a misbehaving feature in seconds without touching a single pipeline. It is no accident that progressive delivery has become standard practice for teams that deploy several times a day.

The four flag types (and why the distinction matters)

Not all flags are equal, and treating them equally is the fast lane to chaos. The most useful classification distinguishes four types by purpose and expected lifespan:

- **Release flags**: hide incomplete or unvalidated functionality. They should live for days or weeks, never months. They get deleted once the feature reaches 100% of users.
- **Experiment flags**: split traffic for an A/B test. They live as long as the experiment does; once you have results, the winning variant is consolidated and the flag disappears.
- **Ops flags**: operational switches such as kill switches or degraded modes. They are among the few that live indefinitely — in exchange, both code paths must be exercised regularly.
- **Permission flags**: control access by plan, role or customer (premium features, for instance). Also permanent, but they are really business logic, and it is worth asking whether they belong in your permissions system instead.

The type determines the lifecycle. A release flag still alive six months later is no longer a tool: it is technical debt in disguise.

OpenFeature: a standard so you don't marry a vendor

For years, adopting feature flags meant coupling your code to one vendor's SDK. **OpenFeature**, a CNCF incubating project, solves exactly that: it defines a unified API for evaluating flags, and each vendor (LaunchDarkly, Flagsmith, Unleash, GO Feature Flag, or your own backend) plugs in behind it as an interchangeable *provider*. If you switch tools tomorrow, your application code never notices.

Here is what it looks like in Python:

```
# pip install openfeature-sdk
from openfeature import api
from openfeature.evaluation_context import EvaluationContext

# In production, register the provider for your flag backend
# (LaunchDarkly, Flagsmith, Unleash, GO Feature Flag...)
api.set_provider(MyProvider())
client = api.get_client("checkout-service")

def process_payment(user):
    ctx = EvaluationContext(
        targeting_key=user.id,
        attributes={"plan": user.plan, "country": user.country},
    )
    if client.get_boolean_value("new-payment-flow", False, ctx):
        return new_payment_flow(user)
    return current_payment_flow(user)
```

Two details matter here. The second argument of `get_boolean_value` is the **default value**: what gets used if the flag backend does not respond. Always choose it with failure in mind, not convenience — for new functionality, the safe default is almost always `False`. And the `EvaluationContext` carries the user attributes you segment on: plan, country, app version, whatever you need.

The TypeScript equivalent (server side):

```
// npm install @openfeature/server-sdk
import { OpenFeature } from '@openfeature/server-sdk';

await OpenFeature.setProviderAndWait(new MyProvider());
const client = OpenFeature.getClient('checkout-service');

const enabled = await client.getBooleanValue('new-payment-flow', false, {
  targetingKey: user.id,
  plan: user.plan,
  country: user.country,
});

if (enabled) {
  await newPaymentFlow(user);
} else {
  await currentPaymentFlow(user);
}
```

One security rule with no exceptions: sensitive logic is evaluated **on the server**. A flag evaluated in the browser can be inspected and tampered with; it must never be the only thing protecting a paid feature or restricted data.

Progressive rollouts: percentages with deterministic bucketing

Turning a feature on for 5% of users forces you to answer a deceptively simple question: which 5%? If you decide with a random number on every request, the same user will watch the feature flicker on and off between page loads. The solution is **deterministic bucketing**: a stable hash of user and flag that pins every user to a fixed point between 0 and 100.

```
import hashlib

def in_rollout(flag: str, user_id: str, percentage: float) -> bool:
    """Assign each user to a stable bucket between 0 and 100.

    The same user always lands in the same bucket for a given flag,
    so raising the percentage only ADDS users — it never changes
    the experience of those already inside.
    """
    digest = hashlib.sha256(f"{flag}:{user_id}".encode()).hexdigest()
    bucket = (int(digest[:8], 16) / 0xFFFFFFFF) * 100
    return bucket < percentage
```

Notice that the hash includes the flag name. Without it, the "first 5%" would be the same group of users for every one of your flags, and that group would absorb all the risk of all your new features.

With bucketing solved, the classic sequence works well: internal users -> 1% -> 5% -> 25% -> 50% -> 100%, advancing only while your key metrics (errors, latency, conversion) hold steady at each step. Decide before you start which metrics you watch and which threshold makes you roll back; deciding during the incident means deciding badly.

Kill switches: the off button you'll be grateful for at 3 a.m.

A well-placed ops flag turns a middle-of-the-night incident into a single click. The idea: wrap non-critical dependencies in a flag that, when switched off, activates controlled degradation.

```
async function getRecommendations(userId: string) {
  const enabled = await client.getBooleanValue(
    'ops-recommendations-enabled',
    true, // default on: this is a kill switch, not a release flag
    { targetingKey: userId },
  );

  if (!enabled) return FALLBACK_POPULAR_ITEMS; // controlled degradation

  try {
    return await recommendationService.fetch(userId);
  } catch {
    return FALLBACK_POPULAR_ITEMS;
  }
}
```

If the recommendation service starts failing or dragging down overall latency, you flip the flag off and the page keeps working with popular content. No emergency deploys. This pattern pairs beautifully with circuit breakers: the breaker reacts automatically, the kill switch gives you explicit manual control.

Common mistakes that cancel out the benefits

Flag debt. Mistake number one. Every flag is an `if` that doubles your execution paths; ten active flags in one service are, in theory, 1,024 possible combinations. Release and experiment flags must be born with an expiry date and an assigned owner.

Dangerous defaults. If the flag backend goes down and your defaults turn on half-finished functionality, you now have two incidents. The default is your behavior at the worst possible moment: choose it with that mindset.

Nested and dependent flags. When flag B only makes sense if A is on, you have created an invisible dependency graph. If you cannot avoid it, document the relationship and explicitly test the valid combinations.

Evaluating on the client what protects something valuable. Mentioned already, but repeated so often it deserves its own bullet: the client can lie. Server side, always, for anything sensitive.

Using flags as configuration. A timeout or a service URL is not a flag: it is configuration. Mixing them in the same system turns your flag dashboard into a junk drawer where nobody can tell what is safe to touch.

Best practices that make the difference

- **Self-explanatory names**: `checkout-new-payment-flow` says more than `flag-23`. Include the area and, if your convention allows, the type (`ops-`, `exp-`).
- **Every flag has an owner and an expiry**: without a responsible person and a target removal date, no temporary flag ever gets cleaned up.
- **A cleanup ritual**: reserve a recurring slot (each sprint, for example) to delete flags sitting at 100% or 0%. Some teams auto-generate tickets when a flag passes its expiry date.
- **Test both paths**: as long as the flag exists, CI must exercise the code with the flag on and off, at least for critical flows.
- **Audit trail and access control**: changing a flag in production is a production change. Log who changed what and when, and set permissions accordingly.

How a flag system works under the hood

Before piling on more patterns, it pays to understand the architecture you are buying into. Every serious flag system separates two planes. The **control plane** is where flags are defined and changed: a dashboard, an API, or a Git repository. The **data plane** is where flags are evaluated: inside your process, on every request, thousands of times per second. Most latency and resilience mistakes come from confusing the two.

There are two evaluation models, and the difference matters more than any vendor feature list. With **in-process evaluation**, the SDK downloads the full ruleset once, keeps it in memory, and evaluates locally. An evaluation is a hash lookup plus a few rule checks: microseconds, no network. Updates arrive in the background via polling or streaming. This is how most server SDKs work, and it is what you want on a hot path.

With **remote evaluation**, every call asks an external service (an agent, a sidecar, or the vendor API) for the decision. It is simpler to reason about and keeps rules out of your process, but now a network hop sits inside your request path, and you must add caching, timeouts and a fallback. Remote evaluation is fine for edge functions and clients; on a busy backend it needs care.

Two rules follow directly from this. First, **never block a request on a network call to the flag backend**. Evaluate from memory, or against a local cache with a hard timeout. Second, at startup, block once while the SDK initializes — with a timeout of a second or two — and if initialization fails, serve defaults and keep booting. A service that cannot start because the flag vendor is down has inverted its dependencies.

Finally, treat flag configuration as what it is: **data that changes production behavior**. A flag change is a deploy in disguise. It deserves the same things a deploy gets — audit trail, review for risky changes, the ability to diff and roll back — even if it travels through a dashboard instead of a pipeline.

flagd in practice: an open-source flag backend with a Unix philosophy

If you like the OpenFeature model but do not want a SaaS contract to get started, **flagd** is the CNCF answer: a small daemon that reads flag definitions from files, HTTP endpoints, gRPC sync sources or Kubernetes custom resources, evaluates targeting rules, and serves decisions to OpenFeature SDKs. No database, no UI, no accounts. Flags live in a JSON document you version in Git, which means pull requests, reviews and history come for free — flags as GitOps.

A complete flag definition with targeting and a progressive rollout looks like this:

```
{
  "$schema": "https://flagd.dev/schema/v0/flags.json",
  "flags": {
    "new-payment-flow": {
      "state": "ENABLED",
      "variants": { "on": true, "off": false },
      "defaultVariant": "off",
      "targeting": {
        "if": [
          { "in": [ { "var": "plan" }, ["internal", "beta"] ] },
          "on",
          {
            "fractional": [
              { "var": "targetingKey" },
              ["on", 25],
              ["off", 75]
            ]
          }
        ]
      }
    }
  }
}
```

Read it from the inside out. `variants` names the possible values and `defaultVariant` is what you get with no targeting match. The `targeting` block is JsonLogic: if the user's `plan` attribute is `internal` or `beta`, they always get `on`. Everyone else falls into the `fractional` operator, which hashes the `targetingKey` with murmurhash3 and assigns 25% of users to `on` and 75% to `off`. The hash is deterministic, so the same user gets the same variant on every request, on every instance — exactly the bucketing property we discussed earlier, implemented for you.

In Kubernetes the usual deployment is a sidecar, so evaluation stays on localhost:

```
containers:
  - name: app
    image: my-service:1.42.0
  - name: flagd
    image: ghcr.io/open-feature/flagd:latest
    args:
      - start
      - --uri
      - file:/etc/flagd/flags.json
    volumeMounts:
      - name: flags
        mountPath: /etc/flagd
```

And the application connects through the flagd provider — swap one line and everything else in this article keeps working:

```
# pip install openfeature-sdk openfeature-provider-flagd
from openfeature import api
from openfeature.contrib.provider.flagd import FlagdProvider

api.set_provider(FlagdProvider(host="localhost", port=8013))
```

flagd also ships an **in-process provider** for several languages: instead of talking to a daemon, your service consumes the sync source directly and evaluates in memory. Same flag files, no sidecar, microsecond evaluations. For teams already doing GitOps, flagd plus a ConfigMap synced by Argo CD is a remarkably complete flag platform for the price of zero.

Designing the evaluation context (and keeping PII out of it)

Every targeting decision in this article hangs off the **evaluation context**: the bag of attributes you pass with each evaluation. It deserves more design attention than it usually gets, because a sloppy context produces inconsistent bucketing, broken segments and privacy problems — all silent.

The most important field is the `targetingKey`. It must be **stable for the unit you want to be consistent**. If a logged-in user should see the same variant on web and mobile, the key is the user ID, not the session or device ID. For anonymous traffic, generate a durable anonymous ID, store it in a first-party cookie or local storage, and keep using it after signup for the flags that were evaluated with it mid-experiment — otherwise users flip variants the moment they log in and your metrics smear.

Beyond the key, keep the context **small and deliberate**. Plan, country, app version, platform and a handful of segment attributes cover almost every real targeting rule. Resist dumping the whole user object into it. Two reasons. First, consistency: if the checkout service sends `plan: "pro"` and the billing service sends `plan: "PRO"`, the same user matches different rules in different services. Centralize context construction in one small function per service, and agree on attribute names across teams — treat the context as a tiny schema, versioned in your docs.

```
def build_flag_context(user, request) -> EvaluationContext:
    """Single source of truth for flag targeting attributes."""
    return EvaluationContext(
        targeting_key=str(user.id),
        attributes={
            "plan": user.plan.lower(),
            "country": user.country_code, # ISO 3166-1 alpha-2
            "app_version": request.headers.get("x-app-version", "unknown"),
            "platform": request.headers.get("x-platform", "web"),
        },
    )
```

Second, **privacy**. Context attributes travel to the flag backend — a third party, if you use SaaS. Email addresses, names and anything sensitive should not be in there; targeting rules almost never need them, and a numeric or hashed ID buckets just as well. If a rule seems to need raw PII ("users whose email ends in @bigcustomer.com"), model it as a segment attribute computed on your side (`is_bigcustomer: true`) instead of shipping the email. Your DPO will thank you, and your rules get faster and clearer at the same time.

Advanced rollouts: rings, canaries and guardrail metrics

A percentage slider is the beginning, not the end, of progressive delivery. Mature teams structure rollouts in **rings**: concentric groups ordered by blast radius. A typical sequence is employees first, then opted-in beta customers, then 1% of general traffic, then 10%, 50% and 100%. The early rings are not statistical samples — they are smoke tests with real humans who will file a bug instead of churning. The percentage phase is where you watch metrics.

Keep the distinction between this and a **canary deploy** sharp. A canary shifts traffic between two versions of the whole binary at the infrastructure layer (a load balancer or service mesh decides). A flag rollout shifts traffic between two code paths inside one binary (your code decides, per user). They compose beautifully: the canary catches crashes, memory leaks and latency regressions of the new build; the flag rollout isolates the behavior of one feature. What you should not do is use one to replace the other — a mesh-level canary cannot give a specific user a consistent experience across requests, and a flag cannot save you from a build that segfaults.

Every percentage step needs two more things: a **bake time** and **guardrail metrics**. Bake time is the minimum period a step must hold before you advance — long enough to cover your traffic cycles; a step that

only saw off-peak traffic proves little. Guardrails are the handful of metrics that define "something is wrong": error rate, p99 latency, and one business metric close to the feature (checkout conversion for a payment flow). Define thresholds *before* the rollout starts, split every dashboard by flag variant, and write down the rule: if a guardrail breaches, the flag goes to 0% first and questions come second.

Teams that do this weekly end up automating it. The logic is not sophisticated — it is a loop:

```
# Sketch of an automated rollout gate (run as a cron or pipeline step)
STEPS = [1, 5, 10, 25, 50, 100] # percent of traffic
BAKE = timedelta(hours=6) # minimum time per step

def advance_rollout(flag, metrics, now):
    step = flag.current_step
    if now - flag.step_started_at < BAKE:
        return "waiting"
    if metrics.error_rate("on") > metrics.error_rate("off") * 1.2:
        set_percentage(flag.key, 0) # kill first, investigate later
        page_oncall(flag.key, "guardrail breach: error rate")
        return "rolled back"
    if metrics.p99_latency("on") > metrics.p99_latency("off") * 1.3:
        set_percentage(flag.key, 0)
        page_oncall(flag.key, "guardrail breach: latency")
        return "rolled back"
    next_pct = STEPS[STEPS.index(step) + 1]
    set_percentage(flag.key, next_pct)
    return "advanced to " + str(next_pct) + "%"
```

Whether you use a vendor's automation or twenty lines of your own, the point is the same: **rollouts governed by explicit thresholds and time**, not by whoever remembered to check the dashboard after lunch.

A/B experiments on top of flags

Experiment flags look like release flags with two variants, but the discipline around them is different, and ignoring that difference is how teams end up "A/B testing" their way to conclusions that are pure noise.

The first distinction is **assignment versus exposure**. Assignment happens when the bucketing hash puts a user in a variant. Exposure happens when the user actually experiences the difference — they reached the screen where the new checkout renders. If you analyze by assignment, every user who never touched the feature dilutes your metrics toward zero effect. Log an **exposure event** at the moment the variant-dependent code runs, with the flag key, the variant and the targeting key, and run your analysis on exposed users. OpenFeature hooks (next section) give you a clean place to emit these events without littering business code.

Second, check for **sample ratio mismatch (SRM)** before believing any result. If your split is configured 50/50 but you observe 52/48 with a large sample, something in the pipeline is biased — bots hitting one variant, a redirect dropping exposures, caching serving one path — and the experiment is invalid regardless of what the metric says. A chi-squared test against the expected ratio takes one line with scipy and should be automatic:

```
from scipy.stats import chisquare

exposed = {"on": 50_412, "off": 48_103}
total = sum(exposed.values())
stat, p = chisquare(list(exposed.values()), [total/2, total/2])
if p < 0.001:
    raise RuntimeError("SRM detected - do not trust this experiment")
```

Third, basic statistical hygiene: fix **one primary metric** and the minimum sample size before starting (any online calculator works), do not stop the moment the p-value dips under 0.05 — peeking inflates false

positives dramatically — and treat secondary metrics as hypothesis generators, not conclusions. If you plan to run experiments regularly, adopt a purpose-built layer such as **GrowthBook** or your analytics platform's experimentation module on top of your flags: they handle exposure logging, SRM checks and sequential statistics so each team does not reinvent them badly.

Finally, remember the lifecycle rule from earlier with extra force: an experiment flag whose experiment has ended is pure debt. Ship the winner, delete the flag, archive the analysis.

Flags in the frontend: React, flicker and bootstrapping

Everything so far ran on the server, where the SDK holds a ruleset and evaluates per request. Browsers and mobile apps invert the model: the client SDK evaluates (or fetches) flags **for one known user**, caches the results, and re-evaluates when the context changes — login, plan upgrade, locale switch. OpenFeature calls this the static-context or client paradigm.

With the React SDK the wiring is small:

```
// npm install @openfeature/react-sdk
import { OpenFeature, OpenFeatureProvider, useFlag } from '@openfeature/react-sdk';

await OpenFeature.setContext({ targetingKey: user.id, plan: user.plan });
await OpenFeature.setProviderAndWait(new MyWebProvider());

function App() {
  return (
    <OpenFeatureProvider>
      <Checkout />
    </OpenFeatureProvider>
  );
}

function Checkout() {
  const { value: newFlow } = useFlag('new-payment-flow', false);
  return newFlow ? <NewCheckout /> : <ClassicCheckout />;
}
```

The classic frontend failure is **flicker**: the page renders with the default variant, the flag response arrives 300 ms later, and the UI snaps to the other variant in front of the user. Three fixes, in order of preference. Bootstrap flags on the server: evaluate during server-side rendering and inline the values into the page, so the client SDK starts hydrated and the first paint is already correct. If you cannot SSR, gate rendering on provider readiness (the React SDK exposes suspense-friendly hooks) and show your normal loading state for those milliseconds. As a last resort, accept the default long enough to avoid the snap — for low-stakes cosmetic flags only.

Two security notes that save real pain. Client SDK keys are **public** — they ship in the bundle — so they must only unlock evaluation, never flag administration; and any flag the client evaluates is visible and forgeable by the client. The rule from the mistakes section applies with no exceptions: cosmetic and UX decisions may live client-side, but anything touching money, quotas or permissions is decided on the server, and the frontend merely renders the outcome. Also mind payload leakage: the names and rules of unreleased flags can be read from network responses, which is how more than one product launch has been scooped by curious users with DevTools open.## Testing with feature flags

Flags multiply execution paths, and your test suite has to keep up without multiplying itself into uselessness. The foundation is the **in-memory provider** that OpenFeature SDKs ship for exactly this purpose: tests set flags to known values with no network, no mocks of HTTP clients, no test doubles of the vendor SDK.

```
import pytest
from openfeature import api
from openfeature.provider.in_memory_provider import InMemoryFlag, InMemoryProvider

def set_flags(**flags):
    memory_flags = {
        key: InMemoryFlag(
            "on" if enabled else "off",
            {"on": True, "off": False},
        )
        for key, enabled in flags.items()
    }
    api.set_provider(InMemoryProvider(memory_flags))

@pytest.mark.parametrize("new_flow", [True, False])
def test_checkout_succeeds_on_both_paths(new_flow):
    set_flags(**{"new-payment-flow": new_flow})
    result = checkout(user=make_user(), cart=make_cart())
    assert result.status == "confirmed"
```

The parametrized test above encodes the golden rule: **while a flag exists, both paths are production code**, and both get tested. For unit tests, parametrize over the flags the code under test actually reads — not every flag in the system. The combinatorial explosion (ten flags, 1,024 combinations) is a real risk only if you let unrelated flags leak into one module; well-scoped flags keep the relevant set per test at one or two.

For integration and end-to-end tests, two practices keep suites deterministic. First, **pin every flag** in the test environment — a suite that inherits whatever the staging dashboard happens to say will flake the day someone toggles something. An env-var-driven provider or a flags file checked in next to the tests works well. Second, add one smoke test per *active rollout* that runs the full flow with the flag on, in staging, against real dependencies: unit coverage of both branches does not catch the integration bug that only the new path triggers.

And close the loop with the lifecycle: when a flag is removed, its parametrizations and the losing branch's tests get deleted in the same pull request. Tests for code that no longer exists are how "temporarily disabled" suites are born.

Observability: knowing what a flag did to production

The moment flags control behavior, "what version is the user on?" stops having a single answer per deploy — it has an answer **per request**. Your telemetry has to carry that answer, or you will spend incidents guessing which variant produced the errors filling your screen.

OpenTelemetry defines semantic conventions for exactly this: a flag evaluation is recorded as an event named `feature_flag` with attributes like `feature_flag.key`, `feature_flag.provider.name`, `feature_flag.result.value` and `feature_flag.result.reason`. OpenFeature **hooks** are the natural place to emit it — write it once, and every evaluation in the service is annotated with zero changes to business code:

```
from openfeature import api
from openfeature.hook import Hook
from opentelemetry import trace

class OtelFlagHook(Hook):
    def after(self, hook_context, details, hints):
        span = trace.get_current_span()
        if span and span.is_recording():
            span.add_event("feature_flag", {
                "feature_flag.key": details.flag_key,
                "feature_flag.provider.name": hook_context.provider_metadata.name,
                "feature_flag.result.value": str(details.value),
                "feature_flag.result.reason": str(details.reason or ""),
            })

api.add_hooks([OtelFlagHook()])
```

With that in place, three views become possible and you should build all three. **Traces**: any slow or failed trace shows which variants the request saw — the "it only breaks for some users" mystery usually dissolves on the spot. **Metrics split by variant**: error rate and latency histograms with a variant label for flags under active rollout (mind cardinality — label with the variant of the one or two flags being rolled out, not all of them). **Change annotations**: every flag change lands as a deployment-style marker in your dashboards, because "what changed at 14:32?" is the first question of every incident, and a flag flip is the answer more often than teams expect.

Log the evaluation reason too. Values answer *what* the user got; the reason (`TARGETING_MATCH`, `SPLIT`, `DEFAULT`, `ERROR`) answers *why* — and a sudden spike in `DEFAULT` or `ERROR` reasons is your earliest signal that the flag backend is degraded and everyone is silently getting fallback behavior. Alert on that ratio; it is the flag system's own health check.

Resilience: when the flag backend goes down

Your flag backend will have a bad day. When it does, the difference between a non-event and a sitewide incident is decided by choices you made weeks earlier, so make them deliberately.

Defaults are your real disaster plan. Every evaluation call carries a default value, and that default is the entire behavior of your system when nothing else works. Audit them: the safe default for a release flag is the old path; for a kill switch, whatever keeps the core flow alive; for a permissions flag, deny. A quick grep for evaluations defaulting to `True` on unreleased features is a fifteen-minute exercise that has prevented real incidents.

Cache aggressively, expire reluctantly. In-process SDKs already hold the ruleset in memory; the question is what happens when it goes stale because updates cannot arrive. The answer should be "keep serving the stale ruleset" — stale flags are almost always better than default flags, because stale preserves whatever rollout state users were already seeing. Client-side, persist the last known values so a user who reopens the app offline gets continuity instead of defaults. For remote evaluation, put a read-through cache with a hard TTL and a stale-while-revalidate policy in front of the calls, and treat the flag service like any other dependency: timeouts in the tens of milliseconds and a circuit breaker so a slow flag backend cannot become your latency problem.

Bootstrap from a file. The strongest pattern for critical services: bake a snapshot of flag values into the deploy artifact (or a ConfigMap) as the initialization fallback. Boot order becomes: try the live backend with a short timeout; on failure, load the snapshot; converge to live values when connectivity returns. Your service now starts correctly even if the flag vendor is entirely down during your 3 a.m. rollback.

```
import json, logging

def init_flags(timeout_seconds: float = 2.0) -> None:
    try:
        provider = FlagdProvider(host="localhost", port=8013,
                                 deadline_ms=int(timeout_seconds * 1000))
        api.set_provider(provider) # blocks until ready or deadline
    except Exception:
        logging.exception("flag backend unavailable, using snapshot")
        with open("/etc/flags/snapshot.json") as f:
            api.set_provider(InMemoryProvider(parse_snapshot(json.load(f))))
```

And then **rehearse it**. A quarterly chaos exercise — block the flag backend's endpoint in staging, watch what the fleet does — verifies the whole chain: defaults sane, caches serving stale, boot path taking the snapshot, and the `reason=ERROR` alert from the observability section actually firing. Teams are reliably surprised the first time they run this.

Governance: naming, ownership and killing flag debt

Flag debt was mistake number one, so it deserves a system, not a resolution. The teams that stay clean treat flags like they treat dependencies: inventoried, owned, and aged out.

Name for grep. A convention like `team.purpose-short-name` (`checkout.release-new-payment-flow`, `platform.ops-disable-recommendations`) makes ownership visible in every evaluation call and makes the cleanup search trivial. Enforce kebab-case and forbid renames — a renamed flag is a new flag with orphaned history.

Every flag gets metadata at birth: an owner (a team, not a person who will change jobs), a type (release, experiment, ops, permission), and — for release and experiment flags — an expiry date. Most platforms support tags or custom fields; if yours does not, a `flags.yaml` registry in the repo works fine. What matters is that the data exists where a script can read it, because the enforcement that works is automated:

```
# CI job: fail the build when release flags outlive their expiry
import sys, yaml, datetime

registry = yaml.safe_load(open("flags.yaml"))
today = datetime.date.today()
expired = [
    f["key"] for f in registry["flags"]
    if f["type"] in ("release", "experiment")
    and datetime.date.fromisoformat(f["expires"]) < today
]
if expired:
    print("Expired flags, remove them or renew with a reason:")
    for key in expired:
        print(" -", key)
    sys.exit(1)
```

A build that fails with "remove this flag" converts debt into a visible, assignable task. Pair it with a monthly fifteen-minute **flag review** per team — walk the list, for each flag ask "still needed?" — and with removal tooling where the codebase justifies it: tools in the lineage of Uber's Piranha, and structured-rewrite frameworks like OpenRewrite, can automatically excise a flag and its dead branch across a large codebase once you decide the winner.

Track one number: **active flags per service**, trending over time on a dashboard someone actually looks at. It is a perfectly honest health metric — flat or falling is a team in control; monotonically rising is 1,024 execution paths waiting for a combination nobody tested.

Case study: rolling out a new payment flow end to end

Let's run the whole playbook once, on the example this article has been using. The team owns checkout; the project replaces the payment flow's orchestration with a new provider integration. Real money, so maximum paranoia.

Week 0 — before any code merges. The flag `checkout.release-new-payment-flow` is created with owner `team-checkout`, type `release`, expiry in 90 days, default `off` everywhere. Guardrails are written into the rollout doc: payment error rate, checkout p99 latency, and authorization success rate from the payment provider, each with a numeric threshold and the standing rule — breach means 0% first, discussion second. Dashboards get a variant split using the OpenTelemetry hook from earlier.

Weeks 1–3 — merge dark, test both paths. The new flow merges to main behind the flag in small pull requests, deployed continuously, executed by nobody. The parametrized suite runs both paths on every commit; a staging smoke test runs the new flow against the payment provider's sandbox nightly. Two integration bugs die quietly here, in front of zero customers.

Week 4 — internal ring. Targeting rule: `plan in ["internal"]` -> `on`. Employees exercise the flow with real cards on a test merchant account. A currency-rounding bug surfaces on day two — the kind that unit tests with tidy amounts never catch. Fixed, redeployed, flag untouched.

Week 5 — beta ring, then 1%. Opted-in beta customers join. After three quiet days, `fractional` sends 1% of general traffic to `on`. Bake time: 24 hours minimum, covering a full daily cycle including the evening peak.

Week 6 — the guardrail earns its keep. At 25%, the authorization success rate for the `on` variant drops 0.8% below `off` — under the alert threshold's grace but visible on the split dashboard. The on-call engineer sets the flag to 0%. Time from suspicion to full mitigation: about forty seconds, no deploy, no rollback, no incident call. Root cause: the new integration formatted one optional field differently, and one regional bank rejected it. A three-line fix ships the next morning; the rollout resumes at 10% the same day.

Week 7 — 50%, then 100%. Metrics hold. The rollout completes; the old flow still exists behind the flag as an instant escape hatch while confidence builds.

Weeks 8–9 — the part everyone skips. Two weeks at 100% with clean metrics. Then the cleanup PR: the flag evaluation, the old flow, its tests and its dashboard split all deleted; the registry entry closed; the payment provider's old credentials scheduled for revocation. The flag lived 65 days — inside its 90-day expiry — and the CI check never had to fire.

Nothing in that story is heroic. That is the point: feature flags convert what would have been a high-stakes release event into nine weeks of boring, reversible steps.

Choosing a provider in 2026

Because everything above is written against OpenFeature, the provider is a swappable detail — which is exactly what makes it worth choosing calmly. The questions that actually matter:

Evaluation model. Does the server SDK evaluate in-process (fast path, works during vendor outages) or call out per evaluation? For your traffic and latency budget, this is the first filter. **Hosting.** SaaS convenience versus self-hosting for data residency and cost control. **Experimentation.** If you need real A/B analysis, does the platform do exposure logging, SRM checks and sequential stats, or just percentage splits? **Governance.** Audit log, RBAC, approvals for production changes, flag lifecycle metadata — the enterprise checklist, which matters at fifty engineers even if it did not at five. **Pricing axis.** Per-seat, per-MAU or per-evaluation; each one punishes a different growth pattern, so model yours before signing.

The short field guide: **LaunchDarkly** is the mature managed reference — streaming updates, strong governance, priced accordingly. **Unleash** and **Flagsmith** are the established open-source platforms you can self-host, with hosted tiers when you tire of operating them. **GrowthBook** leads with experimentation on top of flags. **PostHog** bundles flags with product analytics, attractive when you want one tool for both. And

flagd, as covered above, is the GitOps-native choice with no dashboard and no per-seat bill. A sane progression for many teams: start with flagd or Unleash while needs are simple, and let OpenFeature keep the exit cheap if you outgrow them — the migration is a provider swap and a config change, not a rewrite of every call site.

Flags and trunk-based development: merging work that isn't finished

Feature flags and trunk-based development are two halves of one workflow, and it is worth making the connection explicit because it changes how you write code, not just how you release it. Trunk-based development means everyone merges to main in small increments, daily or better; long-lived feature branches — and their week-long merge conflicts — disappear. The obvious objection is "but my feature takes six weeks". Flags are the answer: incomplete work merges continuously, guarded by a flag that keeps it dark in production.

The technique that makes large changes mergeable in slices is **branch by abstraction**. Instead of an `if` at fifty call sites, introduce one seam — an interface, a factory, a strategy — and put the flag decision in exactly one place:

```
class PaymentFlow(Protocol):
    def execute(self, order: Order) -> PaymentResult: ...

def get_payment_flow(client, ctx) -> PaymentFlow:
    """The only place in the codebase that reads this flag."""
    if client.get_boolean_value("checkout.release-new-payment-flow", False, ctx):
        return NewPaymentFlow()
    return ClassicPaymentFlow()
```

Now the new implementation grows behind the interface pull request by pull request, reviewers see small diffs, and the eventual cleanup deletes one branch of one factory instead of hunting conditionals across the codebase. As a rule of thumb, **one flag should be read in one place**; when you find the same flag key evaluated in five modules, that is the code telling you it wants an abstraction.

Two habits complete the workflow. Keep the *flag check at the highest sensible layer* — decide once per request at the entry point and pass the decision (or the chosen implementation) down, rather than re-evaluating in every helper, which invites the inconsistency of a flag changing mid-request. And resist conditional *compilation* creep: flags gate behavior at runtime precisely so that a single artifact can do both things; the moment you fork builds per variant you have reinvented release branches with more steps.

Mobile and edge: flags where deploys are slow or far away

The value of a flag is proportional to how painful the alternative rollback is — which makes mobile the strongest case of all. A bad web deploy rolls back in minutes; a bad app release means days of store review plus weeks of users who never update. Teams that ship apps treat this as law: **every risky change in a mobile release ships dark**, behind a flag that can be flipped server-side without a new binary. The app evaluates flags with the client paradigm — fetch for this user, cache locally, persist across launches — and the persisted cache doubles as offline behavior: a user in airplane mode gets the last known variants, not the defaults.

Mobile adds two constraints worth designing for. First, **version skew is permanent**: some users run every app version from the last two years, so a flag's targeting often needs an `app_version` condition, and a flag cannot be removed until the last binary that reads it has effectively died out. Budget mobile flag lifetimes in quarters, not weeks. Second, review-time surprises: app store policies frown on features that materially change the app after review, so flags there should modulate and protect functionality, not smuggle in a different product.

At the other extreme, **edge evaluation** moves the decision into the CDN worker before a request touches your origin: A/B routing, gradual traffic shifting between origin versions, instant kill switches for whole routes. The same rules apply with tighter budgets — the worker cannot afford a per-request call to a flag API, so flags arrive as a pushed snapshot (many providers integrate with edge key-value stores for exactly this), evaluation is a local lookup, and the targeting context is limited to what the request itself carries: cookies, headers, geography. Keep edge flags few and coarse; fine-grained personalization still belongs behind the origin where your real context lives.

Migrating an existing codebase to OpenFeature

Most teams do not start from zero — they arrive with a vendor SDK called directly from two hundred places, or a homegrown `settings.FEATURES` dict that grew teeth. The migration to OpenFeature is mercifully mechanical, because the standard was designed to wrap what you already have.

Step one: **wrap, don't rewrite**. Install the OpenFeature provider for your current vendor (official providers exist for the major platforms) or write a thin one over your homegrown store — a provider is one class with a handful of resolve methods:

```
from openfeature.provider import AbstractProvider
from openfeature.flag_evaluation import FlagResolutionDetails, Reason

class LegacySettingsProvider(AbstractProvider):
    """Serves flags from the old homegrown store during migration."""
    def resolve_boolean_details(self, flag_key, default_value, context=None):
        if flag_key in legacy_store:
            return FlagResolutionDetails(
                value=legacy_store.get(flag_key, context),
                reason=Reason.TARGETING_MATCH,
            )
        return FlagResolutionDetails(value=default_value, reason=Reason.DEFAULT)
```

Step two: migrate call sites **opportunistically** — every touched file swaps its direct SDK call for the OpenFeature client, new code only uses OpenFeature, and a simple lint rule (forbid importing the vendor SDK outside the provider module) stops regressions. There is no big-bang weekend; the two styles coexist harmlessly for months.

Step three, when call sites are clean: **swap the provider** at one line in bootstrap, run both stacks briefly in shadow mode if the stakes justify it (evaluate against both, log disagreements, serve the old answer), then decommission. Teams that have done this describe the anticlimax as the whole point — the day you change vendors is the day nobody notices.

Security notes worth their own section

Flags change behavior in production without a deploy, which makes the flag dashboard a **privileged control plane** — treat it like one. Production flag changes deserve the same rigor as production access: SSO with MFA, role-based permissions (viewers, editors per project, approvers for production), and approval workflows for flags marked critical. The audit log is not optional compliance theater; it is the answer to "what changed at 14:32?" during your next incident, and it should flow into the same place as your deploy events.

Think about the **blast radius of a compromised flag**. An attacker with dashboard access can turn off security features, expose unreleased functionality, or redirect traffic percentages — silently, with no pipeline to alarm. Mitigations are straightforward: least-privilege roles, alerts on changes to flags tagged as security-relevant, and kill switches for protective features (a WAF bypass flag, say) requiring two-person approval.

Server keys, client keys and relay credentials are **secrets of different grades**: server SDK keys can read your full ruleset including unreleased flag names and targeting rules (which may themselves leak roadmap or

customer names), so they belong in your secret manager with rotation, never in a repo or a client bundle. And in multi-tenant B2B products, remember that targeting rules naming specific customers (`org_id in [...]`) are visible to anyone with dashboard read access and, if evaluated client-side, potentially to the customers themselves — segment names like `tier-enterprise-pilot` leak less than lists of who is in them.

Production checklist

Before your next feature goes out behind a flag, walk this list. It compresses everything above into the questions that get asked, one way or another — either now, or during the incident review.

- **Type and lifecycle declared:** the flag has a type (release, experiment, ops, permission), an owning team, and — unless it is permanent by design — an expiry date recorded where a script can read it.
- **Name follows the convention** (`team.purpose-name`), greppable, never to be renamed.
- **Default chosen for the worst moment:** if the flag backend is unreachable, the default value is the behavior you can live with. For new functionality that is almost always off.
- **Server-side evaluation for anything valuable:** money, quotas, permissions and pricing decisions are made on the backend; the client only renders them.
- **One read point:** the flag is evaluated in a single place (factory, strategy, entry point) and the decision flows down — no scattered re-evaluations.
- **Both paths tested:** parametrized unit tests cover on and off; one staging smoke test exercises the new path end to end; test environments pin flag values.
- **Bucketing is deterministic** on a stable targeting key, so users keep their variant across requests, instances and sessions.
- **Rollout plan written before it starts:** ring order, percentage steps, bake time per step, and guardrail metrics with numeric thresholds and a pre-agreed rollback rule.
- **Dashboards split by variant** for the flags under rollout, evaluations annotated on traces, and an alert on the rate of `DEFAULT`/`ERROR` evaluation reasons.
- **Failure mode rehearsed:** the service boots and behaves sanely with the flag backend down — snapshot bootstrap, stale cache, sane defaults — and you have actually watched it do so in staging.
- **Kill switch reachable at 3 a.m.:** on-call knows where the off button is, has permissions to press it, and the audit log records who pressed it.
- **Cleanup scheduled:** the removal task exists in the backlog from day one, and CI fails when the expiry passes.

Frequently asked questions

How is a feature flag different from plain configuration?

Mechanically they overlap; the difference is intent and cadence. Configuration describes how a deployment runs (pool sizes, endpoints) and changes rarely, with deploys. Flags describe which behavior users get, change at runtime — sometimes hourly during a rollout — and carry per-user targeting. The operational consequence: flags need audit trails, targeting context and instant propagation; config needs validation and version pinning. Using a config file as a flag system loses the runtime and targeting halves; using flags as a config store buries real configuration in a dashboard outside code review.

How many flags is too many?

There is no magic number; there is a shape. Dozens of active flags in a service can be healthy if most are ops and permission flags with owners and exercised paths. Ten release flags from three quarters ago is unhealthy at any count. Watch the trend of active flags per service and the age distribution of

release/experiment flags — if the oldest release flag is older than your oldest intern, governance has failed, whatever the total.

Do flags slow my application down?

In-process evaluation is a memory lookup plus rule checks — microseconds, invisible next to a database call. The costs that do bite are architectural: remote evaluation on a hot path without a cache (network hop per decision), evaluating the same flag many times per request instead of once, and unbounded metric cardinality from labeling everything with every flag. All three are avoidable with the patterns in this guide.

Should flags decide entitlements — what paying customers can use?

Permission flags are fine as the *delivery mechanism* for plan-based access, and half the industry uses them that way. The trap is letting the flag platform become the *source of truth* for billing state. Entitlements belong in your billing/permissions model, tested and consistent; a flag may cache or mirror that state for fast checks, but when the two disagree, billing wins. If removing your flag vendor would change who has access to what they paid for, the boundary is in the wrong place.

What about flags in batch jobs and background workers?

Same SDK, two adjustments. Evaluate **once per job** (or per batch), not once per item — a million evaluations of the same flag mid-job create noise and, worse, the possibility of the behavior flipping halfway through a batch. And choose the targeting key deliberately: for per-customer jobs use the customer ID so batch behavior matches what that customer sees online; for infrastructure-wide jobs use the job name, which gives you deterministic canarying of the job itself.

Can I run my whole incident response through ops flags?

Degraded modes behind ops flags — disable recommendations, serve cached search, shed non-critical load — are among the highest-value uses of the whole technique. Two conditions keep them trustworthy: every degraded path gets exercised on a schedule (a monthly game day flipping each ops flag in staging, at minimum), because an untested fallback is a rumor; and each one has a runbook line saying when to flip it and what "recovered" looks like. An ops flag nobody has flipped in a year should be assumed broken until proven otherwise.

Conclusion

Feature flags are one of those tools with a small adoption cost and a huge compounding effect: more frequent and boring deploys (in the best sense), incidents resolved with a switch, and experiments that need no dedicated infrastructure. The key is discipline: clear types, safe defaults, server-side evaluation for anything sensitive, and a cleanup routine that keeps today's solution from becoming tomorrow's debt. With OpenFeature as your abstraction layer, you build all of this once — free to change vendors whenever you need to.