Contents

07 — Stripe Billing & SaaS Infrastructure: A Thinking Guide

> **Flagline** — Feature Flag SaaS Stack: React, Next.js 14+ (App Router), Node.js, TypeScript, PostgreSQL (Prisma), Redis Plans: Free ($0), Starter ($15/mo), Pro ($29/mo) Audience: Backend developer with 5+ years PHP/Laravel experience, transitioning to this stack

---

This document is not a code reference. It is a reasoning guide. The goal is to give you the mental models, decision frameworks, and architectural intuitions that a senior engineer uses when building Stripe billing into a SaaS product. If you understand the *why* behind each decision, writing the code becomes the straightforward part.

Think of this as the conversation you would have over coffee with someone who has built billing systems before. They would not hand you a codebase. They would explain how to think about it so you can build it yourself.

Table of Contents

1. Stripe's Mental Model

The Entity Hierarchy That Makes Everything Click

Before you write a single line of Stripe code, you need to internalize how Stripe thinks about billing. Stripe is not just a payment processor – it is a complete billing data model. Once you understand the hierarchy, every API call becomes obvious.

The hierarchy is: **Customer -> Subscription -> Price -> Product**.

Read that bottom-up to understand what each entity *is*:

- A **Product** is an abstract thing you sell. For Flagline, you have two products: "Flagline Starter" and "Flagline Pro." You do not create a Product for the Free tier – Free is the *absence* of a paid subscription. This is an important conceptual point. Free is not a thing the customer buys. Free is the default state.
- A **Price** is a specific way to pay for a Product. Each product typically has at least two prices: monthly and annual. So "Flagline Starter" has a $15/month Price and a $150/year Price. Prices are immutable once created – if you want to change the price, you create a new Price object and migrate existing subscribers. This immutability is intentional: it provides an auditable pricing history.
- A **Subscription** is the ongoing relationship between a customer and a price. It tracks billing cycle dates, payment status, trial periods, and cancellation state. A subscription is the most complex entity in Stripe because it has a rich state machine (more on that in Section 4).
- A **Customer** is a person or organization that pays you. In Flagline, this maps one-to-one with a Tenant (organization). One Customer, one Subscription, one Price at a time. This is the simple case – some SaaS products have multiple subscriptions per customer or metered billing, but Flagline does not need that complexity.

Read the hierarchy top-down to understand the *relationships*: a Customer has zero or one active Subscription. That Subscription references a Price. That Price belongs to a Product. When you want to know "what plan is this tenant on?", you walk this chain: Customer -> active Subscription -> Price -> map Price ID to plan name.

How This Maps to Flagline

Stripe Entity	Flagline Entity	Cardinality
Customer	Tenant (organization)	1:1
Subscription	Tenant's current plan	0..1 active per Tenant
Price	Plan + billing interval	4 total (Starter monthly, Starter annual, Pro monthly, Pro annual)
Product	Plan tier	2 (Starter, Pro)

The mapping is simple because Flagline has a simple billing model. You are not doing per-seat pricing, metered billing, or usage-based billing (yet). Each tenant is on exactly one plan, billed at one interval. This simplicity is a feature – resist the urge to over-engineer billing for scenarios you do not have.

The Key Insight: Stripe Is the Source of Truth

This is the single most important concept in Stripe integration, and the one most developers get wrong:

Stripe is the source of truth for billing state. Your database is a local cache of that truth.

What does this mean in practice? It means that when you store a `currentPlan` field on your Tenant model, or a `status` field on your Subscription model, those are *cached copies* of what Stripe knows. If your database says a tenant is on the Pro plan but Stripe says their subscription is canceled, Stripe is right and your database is wrong.

This has three important implications:

1. **You never update billing state based on user actions alone.** When a user clicks "Upgrade," you do not update your database directly. You send the user to Stripe Checkout. Then Stripe tells *you* (via webhook) that the subscription was created, and *then* you update your database. The user action initiates a flow; Stripe confirms the outcome.
2. **Webhooks are your primary data sync mechanism.** Stripe pushes state changes to you. You do not poll Stripe asking "has anything changed?" This is event-driven architecture, and it is the correct pattern for billing.
3. **You need a reconciliation safety net.** Webhooks are reliable, but not perfectly reliable. A missed webhook means your local cache drifts from truth. A periodic job that fetches subscription state from Stripe and corrects your database catches this drift. Think of it like a database replica that occasionally re-syncs from the primary.

Coming from Laravel Cashier: The Abstraction Gap

If you have used Laravel Cashier, you have worked with Stripe indirectly. Cashier wraps Stripe's API behind Eloquent-friendly methods: `$user->newSubscription('default', 'price_xxx')->create()` becomes a multi-step Stripe API interaction that Cashier handles for you. Cashier also provides its own webhook controller that automatically updates your local `subscriptions` table.

In the Next.js/Node.js world, there is no Cashier equivalent. You interact with the Stripe Node.js SDK directly. This feels like more work at first, but it is actually an advantage for three reasons:

First, you understand exactly what happens at each step. There is no magic method that does five things behind the scenes. When something goes wrong (and something will go wrong – this is billing), you know exactly where to look.

Second, you have full control over the flow. Cashier makes opinionated decisions about things like when to create customers, how to handle webhooks, and what statuses mean. Some of those opinions may not match your product's needs. With the raw SDK, you make those decisions yourself.

Third, the Stripe SDK is remarkably well-designed. The method names mirror the REST API paths. If you can read the Stripe docs, you can write the SDK calls. `stripe.customers.create()` calls `POST /v1/customers`. `stripe.subscriptions.update(id, data)` calls `POST /v1/subscriptions/:id`. There is almost no translation layer.

The mental model bridge: every Cashier method maps to one or two Stripe SDK calls. When you see Cashier code and wonder "what is this really doing?", the answer is always a straightforward Stripe API call. `$user->subscribed()` is a database query against the local subscriptions table. `$user->subscription()->swap('price_xxx')` is `stripe.subscriptions.update()` with a new price. `$user->billingPortalUrl()` is `stripe.billingPortal.sessions.create()`.

The Stripe Client: One Decision

You need a Stripe client instance. The only decision worth discussing is: make it a singleton. Create one instance of `new Stripe(secretKey, { apiVersion: '...' })` in a shared module and import it everywhere. Two things matter here: pin the API version (Stripe releases new versions frequently and you do not want breaking changes surprising you in production), and never instantiate multiple clients (each one creates its own connection pool).

The secret key goes in an environment variable (`STRIPE_SECRET_KEY`). The publishable key (`STRIPE_PUBLISHABLE_KEY`) can be exposed to the browser via `NEXT_PUBLIC_` prefix. The secret key must never be exposed. This is the same pattern as Laravel's `.env` – server-only secrets stay server-only.

---

2. The Checkout Flow Thinking

The Fundamental Choice: Stripe Checkout vs. Custom Payment Form

When a Flagline user decides to upgrade from Free to Starter or Pro, they need to enter payment details. You have two paths:

Path A: Stripe Checkout (hosted). You create a "Checkout Session" via the API, and Stripe gives you a URL. You redirect the user to that URL. Stripe shows a fully-built payment page with card input, validation, 3D Secure handling, localization, and error messages. When the user pays, Stripe redirects them back to your app.

Path B: Custom payment form (Stripe Elements). You embed Stripe's JavaScript components in your own page. You control the layout and design. The card data goes directly to Stripe (never touching your server), but you handle the form submission, error display, and loading states yourself.

Use Path A. Use Stripe Checkout. Here is why:

PCI compliance: Stripe Checkout puts you in SAQ A, the simplest PCI compliance level. You self- certify that card data never touches your servers. With a custom form, you are still SAQ A (because Stripe Elements also keeps card data off your server), but you have more surface area for mistakes.

3D Secure: European cards increasingly require 3D Secure (the popup that asks for a bank verification code). Stripe Checkout handles this automatically. With a custom form, you have to build the 3D Secure redirect/popup flow yourself. It is fiddly, and getting it wrong means European customers cannot pay you.

Localization: Stripe Checkout automatically translates to the user's browser language and displays local payment methods. You get Apple Pay, Google Pay, and local methods (iDEAL in Netherlands, Bancontact in Belgium) without any work.

Conversion optimization: Stripe A/B tests their checkout page across millions of transactions. Their form has better conversion than anything you will build yourself. This is not false modesty – they have more data and more incentive to optimize checkout UX than you do.

The only reason to use a custom form is if you need an embedded checkout experience (the user never leaves your page) and you are willing to invest significant engineering effort in it. For a feature flag SaaS, this is not worth it. Ship with Checkout, focus on your actual product.

Coming from Laravel: Cashier's `$user->checkout('price_xxx')` method creates a Checkout Session behind the scenes. You were already using Stripe Checkout in Laravel without realizing it. The `checkout()` method returns a redirect response to the Stripe-hosted page. Exact same concept here.

The Redirect Flow

The checkout flow has a specific sequence, and understanding it prevents a common mistake:

1. User clicks "Upgrade to Starter" in your Flagline dashboard.
2. Your frontend sends a POST to your backend (`/api/billing/checkout`).
3. Your backend creates a Stripe Checkout Session, specifying the price, customer, success URL, and cancel URL.
4. Stripe returns a URL. Your backend sends it to the frontend.
5. Your frontend redirects the browser to that Stripe URL (`window.location.href = url`).
6. User enters payment info on Stripe's page and pays.
7. Stripe redirects the browser to your success URL.
8. **Meanwhile, independently**: Stripe fires a `checkout.session.completed` webhook to your server.

The common mistake is in step 7: **do not provision the subscription when the user hits the success URL.** The success URL redirect is a convenience for the user's browser – it tells them "great, it worked!" But the user might close the tab before the redirect completes. They might have a slow connection. Their browser might crash. You cannot rely on the success URL being visited.

Instead, provision the subscription in the webhook handler (step 8). The webhook is server-to-server and is retried by Stripe if your server does not acknowledge it. It is the reliable signal.

What do you show on the success page? A simple "Thank you, your subscription is being activated" message. If the webhook has already fired by the time the user lands on the page (likely, since webhooks are fast), you can show their new plan. If it has not fired yet, show a brief loading state. The key point: the success page *reads* state, it never *writes* state.

The cancel URL is simpler: the user clicked "back" or closed the Stripe Checkout page without paying. Redirect them to your pricing page. No state change needed.

When to Create a Stripe Customer

You need a Stripe Customer ID before you can create a Checkout Session (the `customer` parameter). The question is: when do you create this Customer object in Stripe?

Option A: Eager creation (on signup). The moment a user creates a Flagline organization, you call `stripe.customers.create()` and store the `cus_xxx` ID on your Tenant model. Pro: you always have a Customer ID when you need it. Con: you create Customer objects for users who may never pay. If 90% of your signups stay on Free, that is 90% unnecessary Stripe API calls and orphan Customer objects in your Stripe dashboard.

Option B: Lazy creation (on first checkout). You create the Customer only when the user first tries to upgrade. You check: does this Tenant already have a `stripeCustomerId`? If yes, use it. If no, create one now, store it, then proceed with checkout. Pro: you only create customers for users who show intent to pay. Con: slightly more complex checkout code (the "get or create" pattern).

Flagline uses lazy creation. The reasoning: for a freemium SaaS with a generous free tier, most users will never pay. Creating Stripe customers eagerly clutters your Stripe dashboard and makes analytics harder (you see thousands of customers with $0 lifetime value). Lazy creation keeps your Stripe data clean and meaningful.

The "get or create" pattern is simple: check `tenant.stripeCustomerId`, if null then create and persist, return the ID. This function gets called at the start of your checkout and portal endpoints.

One crucial detail: when creating the Customer, set `metadata` with your `tenantId`. This is how you link Stripe data back to your data. Every Stripe entity that supports metadata should carry your internal IDs. You will thank yourself when debugging webhook events.

Coming from Laravel: Cashier uses the eager approach – `$user->createAsStripeCustomer()` is typically called during registration, or Cashier creates one lazily on the first `newSubscription()` call. Same trade-off, same decision to make.

Preventing Double Subscriptions

Before creating a Checkout Session, check if the tenant already has an active subscription. If they do, do not let them go through checkout again – redirect them to the Customer Portal instead (where they can change plans). This prevents the scenario where a user accidentally subscribes twice and gets double-billed.

A simple database query – "does this tenant have a subscription with status `active` or `trialing`?" – is sufficient. Return an error like "Already subscribed, use billing portal to change plans."

---

3. Webhook Thinking – The Hardest Part

Why Webhooks Are the Backbone

Webhooks are where most developers struggle with Stripe, and it is not because the code is hard. It is because the *mental model* is unfamiliar. Most web development is request/response: the

user does something, your server handles it, you return a result. Webhooks flip this: Stripe does something to your server, and your server reacts.

The mental model is: **Stripe sends you events about things that happened. You do not ask Stripe "did the payment succeed?" – Stripe TELLS you.**

When a payment goes through, Stripe fires `invoice.paid` to your webhook endpoint. When a subscription is canceled, Stripe fires `customer.subscription.deleted`. When a trial is about to end, Stripe fires `customer.subscription.trial_will_end`. Your webhook handler receives these events and updates your local database accordingly.

This is event-driven architecture. If you have used Laravel events and listeners, the concept is the same. The difference is that the event publisher is an external service (Stripe), not your own application. And the delivery mechanism is HTTP, not an in-memory event bus.

The Webhook Endpoint

Your webhook endpoint is a single Next.js Route Handler at something like `/api/webhooks/stripe`. It receives POST requests from Stripe. Each request contains a JSON event with a `type` field (like `checkout.session.completed`) and a `data.object` field (the relevant Stripe object).

Your handler has three responsibilities in order:

1. **Verify the signature.** Stripe signs every webhook with a shared secret. You must verify this signature before processing the event. Without verification, anyone could POST fake events to your endpoint and give themselves a free Pro plan. The SDK provides `stripe.webhooks.constructEvent(rawBody, signature, webhookSecret)` – if it throws, the event is not from Stripe. Reject it with a 400.
2. **Route the event to the appropriate handler.** A switch/case on `event.type`. You handle the events you care about and ignore the rest. Stripe sends dozens of event types; you probably care about six or seven.
3. **Return a response quickly.** Stripe expects a 2xx response within 20 seconds. If you do not respond in time, Stripe treats it as a failure and retries. Return 200 as soon as you have processed (or queued) the event. If your processing is heavy, consider acknowledging the webhook immediately and processing the event asynchronously via a job queue.

Coming from Laravel: Cashier provides a `WebhookController` with methods like `handleCustomerSubscriptionUpdated()` that you override. Here, you write the routing logic yourself. The `stripe.webhooks.constructEvent()` call replaces Cashier's built-in signature verification middleware. Same concepts, one less abstraction layer.

The Critical Events and What They Mean

Not all webhook events matter equally. Here are the ones Flagline needs and the mental model for each:

`checkout.session.completed` – A customer just finished Stripe Checkout. This is where you create the subscription record in your database. The event gives you the Checkout Session object, which contains the `subscription` ID and your `metadata` (including `tenantId`). Retrieve the

full subscription from Stripe, extract the price ID, map it to a plan, and write the record. This is "provisioning" – the moment a free user becomes a paying customer.

`invoice.paid` – An invoice was successfully paid. This fires both on initial signup (the first invoice) and on every renewal. On renewals, use it to confirm the subscription is still active and update the billing period dates in your database. This is also a good place to reset any periodic usage counters if you track them per billing cycle.

`invoice.payment_failed` – A payment attempt failed. The card was declined, had insufficient funds, or expired. Stripe will retry based on your Smart Retry settings. Meanwhile, the subscription enters `past_due` status. This is where you update your local status to `past_due` and send the user a notification: "Your payment failed, please update your payment method." Do not lock them out yet – give them a grace period (more on this in Section 4).

`customer.subscription.updated` – The subscription changed in some way. This is the most versatile event. It fires when a plan swap happens (user upgraded or downgraded via the Customer Portal), when a cancellation is scheduled (`cancel_at_period_end` becomes true), when a trial ends, and various other changes. Your handler should read the *current state* of the subscription object (which is included in the event payload) and sync it to your database. Do not try to infer what changed from the event type – just take the current state and write it.

`customer.subscription.deleted` – The subscription is fully terminated. Either the user canceled and the cancellation period ended, or Stripe gave up on collecting payment after all retry attempts. Downgrade the tenant to the Free plan. Important: do not delete their data. Do not remove their flags or projects. Just restrict what they can *create*. This is both good UX (they can see what they will get back if they re-subscribe) and good business (it creates an incentive to come back).

`customer.subscription.trial_will_end` – Stripe sends this 3 days before a trial ends. If you are using Stripe-managed trials (credit card required), this is your cue to send a reminder email. For Flagline's no-credit-card trial, you manage trial timing yourself and this event is less relevant, but still good to handle.

Idempotency: The Non-Negotiable

Stripe may deliver the same event more than once. This is not a bug – it is a design decision. Network issues, retries, and race conditions can all cause duplicate delivery. Your webhook handlers **must be safe to run twice** with the same event.

How to think about idempotency:

The simplest approach is to use `upsert` instead of `create` when writing records. If the subscription record already exists (keyed by `stripeSubscriptionId`), update it. If it does not exist, create it. Running an upsert twice with the same data produces the same result. This is naturally idempotent.

For a belt-and-suspenders approach, track processed event IDs. Create a `ProcessedStripeEvent` table with a unique `stripeEventId` column. Before processing an event, check if the ID exists. If it does, skip it. If it does not, insert the ID and process the event. This catches duplicates even when the processing has side effects that are not naturally idempotent (like sending emails – you do not want to send the "payment failed" email twice for the same failure).

Be aware of a subtlety: the event ID check and the event processing should ideally be atomic.

If your handler crashes between recording the event ID and finishing processing, you will have recorded the event as processed but not actually processed it. For most SaaS applications, this edge case is acceptable – the reconciliation job (Section 7) will catch any drift. But if you need stronger guarantees, wrap both operations in a database transaction.

The Ordering Problem

Stripe does not guarantee event delivery order. This is the second-hardest thing about webhooks (after idempotency). You might receive `invoice.paid` before `checkout.session.completed`. You might receive `customer.subscription.updated` before the subscription record exists in your database.

How to think about this: design your handlers to be **tolerant of missing records.** If a handler tries to find a subscription record and it does not exist yet, do not crash. Log a warning and return 200. The "earlier" event will arrive eventually and create the record. If it does not, the reconciliation job (Section 7) will fix the state.

Another defensive pattern: do not derive state from the event type. Do not think "I received `invoice.paid`, so the subscription must be active." Instead, read the subscription object's actual `status` field from the event payload. The subscription object always reflects its current state, regardless of which event type delivered it.

Signature Verification: Non-Optional

You might be tempted to skip signature verification during development. Do not. Here is why:

Without signature verification, your webhook endpoint is an unauthenticated API that changes billing state. Anyone who discovers the URL can POST a crafted payload and modify subscriptions. This is not a theoretical risk – automated scanners look for common webhook paths like `/api/webhooks/stripe`.

Verification requires the raw request body (as a string or buffer, not parsed JSON) and the `stripe-signature` header. In Next.js App Router, read the body with `request.text()` (not `request.json()`). This is the most common Stripe webhook bug in Next.js: if you parse the body as JSON first, the raw bytes are consumed and signature verification fails.

During local development, use `stripe listen --forward-to localhost:3000/api/webhooks/stripe` to forward events from Stripe to your local server. The CLI generates its own webhook secret – use that as your `STRIPE_WEBHOOK_SECRET` in `.env.local`. In production, register the endpoint in the Stripe Dashboard and use the production webhook secret.

Return Values: What Stripe Does with Your Response

Your Response	What Stripe Does
200-299	Marks event as delivered. No retry.
400-499	Marks event as failed permanently. No retry.
500-599	Marks event as failed. Retries with exponential backoff for up to 3 days.
Timeout (>20 seconds)	Treats as 500. Retries.

The implication: return 200 for events you handled successfully or chose to ignore (an event type you do not care about). Return 500 only for *transient* failures where a retry would help (your database is temporarily down, an external service timed out). Never return 500 for permanent failures (invalid data, missing metadata) – retrying will not help, and Stripe will keep retrying for 3 days, polluting your error logs.

Coming from Laravel: This is one area where Cashier's abstraction actually hides important behavior. Cashier's webhook controller returns 200 for all recognized events and 404 for unrecognized ones. You rarely think about return codes. With the raw SDK, you need to be deliberate about what you return.

---

4. Plan-Based Feature Gating Thinking

The Relationship Between "What Stripe Says" and "What the User Can Do"

Feature gating is where billing state meets product behavior. The question your app constantly asks is: "Can this tenant do this thing?" The answer depends on what plan they are on, which depends on their Stripe subscription state.

The key design principle: **the plan limits configuration lives in your code, not in Stripe.** Stripe knows that a tenant pays $15/month. Your code knows that $15/month means they can have 25 flags, 5 projects, and 100K evaluations per day. This separation is intentional – Stripe handles money, you handle product logic.

Your plan limits live in a simple configuration object:

```
PLAN_LIMITS = { FREE: { maxFlags: 3, ... }, STARTER: { maxFlags: 25, ... }, PRO: { maxFlags:
1, ... } }
```

And a mapping from Stripe Price IDs to plan names:

```
PRICE_TO_PLAN = { "price_starter_monthly": "STARTER", "price_pro_annual": "PRO", ... }
```

These two data structures are the bridge between Stripe's billing world and your product's feature world. Everything else follows from them.

Where to Enforce Limits

Always on the server. Never trust the client.

This is not a Stripe-specific principle, but it is worth emphasizing because billing enforcement is a common place to get lazy. You might think: "I will disable the 'Create Flag' button in the UI when the user is at their limit." That is good UX, but it is not enforcement. Anyone with browser dev tools or an API client can bypass your UI. The real enforcement happens in your API route handlers: before creating a flag, query the count and compare it to the plan limit. If over the limit, return a 403 with a clear message.

The pattern for every gated action:

1. Determine the tenant's effective plan.
2. Look up the plan's limits.

3. Check current usage against the limit.
4. If at or over the limit, return an error with a clear message and an upgrade nudge.
5. If under the limit, proceed with the action.

This check should be a reusable function: `canCreateFlag(tenantId)`, `canCreateProject(tenantId)`, `canAddTeamMember(tenantId)`. Call these at the top of the relevant API handlers, before doing any work. The pattern is identical to Laravel's Gates and Policies – a centralized authorization check that returns allow/deny.

The Subscription Status State Machine

A Stripe subscription has a status field that moves through a state machine. Understanding this state machine is essential for getting feature gating right:

`active` – The subscription is paid and current. Full access to plan features. This is the happy path.

`trialing` – The subscription is in a trial period. Full access to plan features. The user has not been charged yet but will be when the trial ends (assuming credit-card-required trial). If you are using a no-credit-card trial like Flagline, you manage the trial state yourself and this Stripe status is less relevant.

`past_due` – A payment failed and Stripe is retrying. This is the critical state for your grace period logic. The subscription is technically still alive – Stripe has not given up yet. The question is: do you still give the user access?

`canceled` – The subscription is terminated. Either the user chose to cancel and the period ended, or all payment retry attempts failed. Downgrade to Free.

`incomplete` – The initial payment for a new subscription failed. The user started checkout but the payment did not go through (e.g., 3D Secure was abandoned). The subscription was never fully activated.

`incomplete_expired` – Same as above, but the incomplete subscription timed out (after about 23 hours).

`unpaid` – All retry attempts exhausted for a `past_due` subscription. This is the terminal failure state. Downgrade to Free.

The decision you need to make: which statuses grant access to paid features?

A reasonable default: `active` and `trialing` get full access. `past_due` gets full access during a grace period (7 days from period end is common). Everything else gets Free tier.

Your "effective plan" function encodes this logic. It looks up the tenant's subscription, checks the status, applies grace period rules, and returns the plan slug. Every feature gating check goes through this function.

Grace Periods: The UX Thinking

When a payment fails, your instinct might be to immediately restrict the user to the Free tier. Do not do this. Here is why:

The user probably does not know their payment failed. Their card expired, or their bank flagged the charge, or they hit a temporary card limit. If you lock them out instantly, their experience is: "I was using Flagline normally, and suddenly my flags stopped working in production." That is terrifying for a developer relying on your service.

Instead, give them time to fix it. A 7-day grace period is industry standard. During those 7 days:

- Show a prominent banner: "Your payment failed. Please update your payment method."
- Send an email with a direct link to the billing portal (where they can update their card).
- Continue providing full access to their plan features.

On day 3, send a reminder email. On day 7, if the payment is still failing, downgrade to Free tier limits. Even then, do not delete their data – just restrict new creation. They can still see their flags, they just cannot create new ones beyond the Free limit.

This approach reduces involuntary churn (customers who leave because of payment issues, not because they chose to leave). Involuntary churn is the most frustrating kind of churn because the customer *wanted* to keep paying.

Coming from Laravel: Cashier provides `$user->subscription()->onGracePeriod()` which checks if the subscription is canceled but the billing period has not ended yet. The concept is similar, but Flagline's grace period is specifically about *payment failure* grace, not cancellation grace. You build this logic yourself: check if status is `past_due` and the current date is within N days of `currentPeriodEnd`.

Upgrade and Downgrade: Proration Thinking

When a user changes plans mid-billing-cycle, someone is owed money. Stripe handles this through proration, and the behavior differs by direction:

Upgrade (Starter to Pro): The user gets Pro features immediately. They are charged the prorated difference for the remainder of the current billing cycle. So if they upgrade on day 15 of a 30-day cycle, they pay roughly half the price difference. Stripe calculates the exact amount. The key product decision: upgrades take effect *immediately*. The user should not have to wait until the next billing cycle to get the features they just paid for.

Downgrade (Pro to Starter): This is trickier. You have two options:

1. **Downgrade immediately, credit the difference.** The user loses Pro features right away and receives a prorated credit applied to future invoices. This is the simpler implementation but a harsher user experience.
2. **Downgrade at period end.** The user keeps Pro features until the end of the current billing cycle, then switches to Starter on the next renewal. This is the standard SaaS pattern and what the Stripe Customer Portal implements by default. The user already paid for the full cycle at the Pro rate – let them use what they paid for.

Flagline should use option 2 for downgrades. The subscription's `cancel_at_period_end` behavior (or more precisely, the schedule behavior) handles this in Stripe. The Customer Portal implements this out of the box if you configure it to allow plan changes.

The practical implication for your code: when you see a `customer.subscription.updated` event with a new price, check whether the change is immediate or scheduled. If scheduled, you

might show the user “You are on Pro until [date], then switching to Starter” in the billing dashboard.

---

5. Usage Tracking Thinking

Two Types of Limits, Two Enforcement Strategies

Flagline has two fundamentally different types of usage limits, and they require different tracking strategies:

Count limits are things like “number of flags” and “number of team members.” These are *stock* quantities – they go up when you create something and down when you delete something. Enforcement happens at creation time: before creating a flag, count the existing flags and compare to the limit. PostgreSQL is the right data store for these because the data is already there (your `flags` table and `team_members` table). A simple `COUNT(*)` query at creation time is sufficient.

Rate limits are things like “evaluations per day.” These are *flow* quantities – they accumulate over a time window and reset periodically. Enforcement happens on every request. A counter incremented on every API call, with automatic reset at the end of each UTC day. Redis is the right data store for these because the access pattern is “increment a counter on every request” – PostgreSQL would collapse under this write pressure at scale.

Understanding this distinction prevents a common mistake: trying to use the same mechanism for both. Do not try to track evaluation counts in PostgreSQL (too slow for per-request writes). Do not try to track flag counts in Redis (the data is already in Postgres, and you would have to keep Redis in sync with every flag creation and deletion).

Why Redis for Rate-Based Tracking

The evaluation API is the highest-volume endpoint in a feature flag service. Every feature flag check in every SDK in every client’s application calls this endpoint. If a tenant has 100K evaluations per day, that is roughly 1.2 requests per second sustained, with bursts potentially much higher.

Redis handles this trivially because of three properties:

Atomic increment. `INCR key` atomically increments a counter and returns the new value. No race conditions, no read-modify-write cycles, no locks. Two concurrent requests incrementing the same key will both get correct results.

Automatic expiry. `EXPIRE key seconds` sets a time-to-live. When you create a daily counter key like `usage:evals:{tenantId}:2026-02-06`, you set it to expire at the end of the UTC day (plus a small buffer for clock skew). The counter auto-deletes when the day is over. No cleanup jobs needed.

Sub-millisecond latency. A Redis INCR takes less than 1ms. Adding this to every evaluation request adds negligible overhead. Compare this to a PostgreSQL `UPDATE ... SET count = count + 1`, which involves disk I/O, WAL writes, and transaction overhead.

The key naming pattern is straightforward: `usage:evals:{tenantId}:{YYYY-MM-DD}`. Date in the key means each day gets a fresh counter. The TTL ensures cleanup.

The Increment-Then-Check Pattern

A subtle but important design decision: do you check the limit *before* incrementing, or increment *first* and then check?

Increment first. Here is why: if you check first ("is the count below the limit?") and then increment, there is a time-of-check-to-time-of-use (TOCTOU) race condition. Between your check and your increment, another request might have pushed the count over the limit. You cannot use a Redis transaction for this because the check and increment are conceptually separate operations.

By incrementing first and then comparing the result to the limit, you guarantee correctness. Yes, this means the counter might briefly exceed the limit (by the number of concurrent requests that squeezed in between the increment and the rejection). This slight over-counting is acceptable – the important thing is that requests *beyond* the limit are rejected.

In code, the pattern is: `newCount = redis.incr(key)` then `if (newCount > limit) return 429`. Single atomic operation followed by a comparison. Clean and race-free.

Persisting to PostgreSQL for Analytics

Redis data is ephemeral. It is fast but not designed for historical storage. For billing dashboards, usage analytics, and trend reporting, you need usage data in PostgreSQL.

The pattern: run a nightly cron job that reads each tenant's daily counter from Redis and writes it to a `DailyUsage` table in PostgreSQL. Use an `upsert` keyed on `(tenantId, date)` so the job is idempotent – running it twice for the same day just overwrites with the latest count.

This separation of concerns is important: Redis handles the real-time, per-request counting. PostgreSQL handles the historical, analytical storage. Each data store does what it is good at.

Hard Limits vs. Soft Limits: The UX Decision

When a tenant hits their evaluation limit, you have a choice:

Hard limit: Return a 429 error. The SDK evaluation call fails. The feature flag defaults to its fallback value (typically `false`). This is what Flagline does – and it is the right choice for evaluation limits because of how SDKs work. A well-built SDK handles 429s gracefully by returning the default value, so the end user's application does not crash. But the flag evaluations stop reflecting the configured state.

Soft limit: Let the evaluation through but log that the tenant is over their limit. Show a warning in the dashboard. This is a less aggressive approach and might be appropriate for count limits (flags, team members) where you want to nudge rather than block.

Flagline's strategy: hard limit on evaluations (429 on the API), soft limit on counts at the 80% threshold (show a warning), hard limit on counts at 100% (prevent creation). This balances enforcement with good UX.

One important detail: **never rate-limit the dashboard.** Even if a tenant's evaluation API is returning 429s, they must always be able to log in, see their flags, manage their settings, and upgrade their plan. Only the high-volume SDK evaluation endpoint is rate-limited.

Rate Limit Headers

When you return a 429, include standard rate limit headers: `X-RateLimit-Limit`, `X-RateLimit-Remaining`, `X-RateLimit-Reset`, and `Retry-After`. These help SDK developers understand what happened and when they can retry. Even on successful responses, include the limit and remaining headers so SDKs can proactively monitor usage.

---

6. Free Trial Thinking

The Strategic Decision: Credit Card Required vs. Not Required

This is not a technical decision – it is a business decision with technical implications. The trade-off is well-studied:

No credit card required: More people start the trial (lower friction). Fewer convert to paid (no payment method on file, so conversion requires a separate purchase action). You get more top-of-funnel volume but a lower conversion rate. Good for products where the value becomes obvious with usage – if people use it, they will want to pay.

Credit card required: Fewer people start the trial (payment form is a barrier). More convert to paid (automatic charge when trial ends, inertia works in your favor). You get less trial volume but a higher conversion rate. Good for products where you are confident that a trial user will find value.

Flagline uses no-credit-card trials. The reasoning: feature flags are a "use it and see" kind of product. A developer needs to integrate the SDK, create a flag, and see it work in their app before they understand the value. Asking for a credit card before they have done any of that kills activation. Get them using the product first; ask for money later.

Implementation Thinking: Stripe vs. Self-Managed Trials

If you choose credit-card-required trials, Stripe manages the trial for you. You pass `trial_period_days: 14` when creating the Checkout Session, and Stripe handles everything: the user is not charged for 14 days, the subscription status is `trialing`, and Stripe fires the `customer.subscription.trial_will_end` event 3 days before it ends. Clean and simple.

If you choose no-credit-card trials (like Flagline), Stripe is not involved in the trial at all. There is no Stripe Subscription yet – the user has not entered payment info. You manage the trial entirely in your own database: a `trialEndsAt` timestamp on the Tenant model, a boolean `isTrialing` flag, and a cron job that expires trials when the timestamp passes.

The self-managed approach requires more code but gives you more control. You decide exactly when the trial starts, what plan the trial simulates (Flagline trials Pro), how to handle expiration (soft downgrade vs. hard lockout), and how to handle re-activation (can an expired trial user get another trial?).

The Trial Lifecycle

The flow for Flagline's no-credit-card trial:

1. **Start:** User creates an organization. Your signup flow calls a `startTrial` function that sets `isTrialing = true`, `trialEndsAt = now + 14 days`, and `currentPlan = PRO` on the Tenant.
2. **Active trial:** The user has full Pro features. Show a banner in the dashboard: "You are on a free Pro trial (X days remaining)." Include a link to the pricing page. The banner should be informative but not aggressive – they are still evaluating.
3. **Approaching end (days 11-13):** The banner becomes more urgent: "Your trial ends in 3 days. Upgrade now to keep your features." Send an email reminder. The CTA is to go to the pricing page and click upgrade, which takes them through Stripe Checkout.
4. **Expiration:** A cron job (runs hourly is fine) queries for tenants where `isTrialing = true` and `trialEndsAt <= now()` and there is no active paid subscription. For each, set `isTrialing = false` and `currentPlan = FREE`. The user's existing flags and projects are preserved but they cannot create new ones beyond Free limits.
5. **Conversion:** At any point during or after the trial, the user can upgrade. If during the trial, they go through Stripe Checkout and get a real subscription. The trial banner disappears, replaced by their subscription status. If after the trial expires, they go through the same checkout flow – they just experience the Free limitations until they pay.

Preventing Trial Abuse

Users will try to get unlimited free trials. The most common vector: creating a new account with a different email address. Defenses include:

- Track whether a user (by user ID) has ever had a trial across any organization they own. If yes, no new trial.
- For business email domains, track how many trials have been started from that domain. After a threshold (5 is reasonable), flag it.
- For consumer email domains (gmail, etc.), rely on the user-level check.

Do not over-invest in abuse prevention at launch. The cost of a few extra free trials is negligible compared to the cost of building a fraud detection system. Add more defenses as you see abuse patterns.

The Email Sequence

Trials convert better with a well-timed email sequence:

Day	Email	Purpose
0	Welcome	Onboarding – help them create their first flag
3	Tips	Show them Pro-only features they might not have discovered
10	Reminder	"4 days left – here is what you have built so far"

Day	Email	Purpose
13	Last chance	"Your trial ends tomorrow. Upgrade to keep everything."

Track which emails you have sent (a simple `SentEmail` table) so you never send the same one twice. Run the email sequence check as a daily cron job.

Coming from Laravel: Laravel has Cashier trial support built in (`$user->onTrial()`, `$user->onGenericTrial()`). Cashier distinguishes between "subscription trials" (Stripe-managed) and "generic trials" (self-managed, similar to Flagline's approach). The concepts map directly.

---

7. Keeping State in Sync

The Hardest Conceptual Challenge

You have two representations of billing state: Stripe (the source of truth) and your PostgreSQL database (the local cache). Keeping them in sync is an ongoing challenge, not a one-time setup.

The sync mechanism has three layers, each catching failures in the previous:

Layer 1: Webhooks (real-time). Stripe fires events as things happen. Your webhook handler updates the database. This handles 99%+ of state changes within seconds. It is your primary sync mechanism.

Layer 2: Reconciliation job (periodic). A cron job (daily is sufficient) that iterates over tenants with Stripe customers, fetches their subscription state from the Stripe API, and corrects any discrepancies in your database. This catches missed webhooks, which can happen due to network issues, endpoint downtime, or bugs in your handler.

Layer 3: On-demand verification (rare). For sensitive operations (like changing the plan or accessing a Pro-only feature for the first time in a session), you can fetch the subscription state from Stripe in real-time and compare it to your cached state. This is the most expensive layer (an API call per check) and should be used sparingly, but it provides the strongest guarantee.

In practice, Layers 1 and 2 are sufficient. Layer 3 is a paranoia option for high-stakes operations.

What to Store Locally

Your Subscription model should store:

- `stripeSubscriptionId` – the unique identifier in Stripe. This is your join key.
- `stripeCustomerId` – for quick lookups when you need to call Stripe.
- `stripePriceId` – which price they are paying. Mapped to plan slug via your configuration.
- `plan` – the denormalized plan slug (FREE, STARTER, PRO). Redundant with `stripePriceId` but makes queries much simpler. You do not want to join through a price-to-plan mapping on every feature gate check.
- `status` – the Stripe subscription status. Critical for feature gating.
- `currentPeriodStart` and `currentPeriodEnd` – the billing cycle dates. Used for grace period calculations and usage tracking.

- `cancelAtPeriodEnd` – whether the user has scheduled a cancellation.
- `trialStart` and `trialEnd` – if using Stripe-managed trials.

Your Tenant model should store:

- `stripeCustomerId` – the link to Stripe.
- `currentPlan` – a denormalized copy of the active subscription's plan, or FREE if no active subscription. This is your hot-path lookup – most feature gate checks only need this field.

The principle: if you need to know the plan on every request, do not make a Stripe API call or even a Subscription table join on every request. Cache the plan on the Tenant. Update the cache via webhooks.

The Reconciliation Job

The reconciliation job is simple in concept: for each tenant with a Stripe customer, fetch their subscriptions from Stripe and compare to your local state. If they differ, update local state to match Stripe.

Implementation considerations:

- **Rate limiting.** Stripe's API has rate limits (25 requests/second in test mode, higher in live). If you have thousands of tenants, batch your reconciliation. Process 50 tenants at a time with a small delay between batches.
- **Logging.** Log every correction. This data is gold for debugging. "Reconciliation: tenant X plan corrected STARTER -> PRO" tells you that a webhook was missed or mishandled.
- **Idempotency.** The reconciliation job should be safe to run at any time, multiple times. It reads from Stripe and writes to your database. Running it twice produces the same state.
- **Error tolerance.** If fetching one tenant's data from Stripe fails, log the error and continue with the next tenant. Do not let one failure stop the entire job.

The Golden Rule

When Stripe and your database disagree, **Stripe wins**. Always. Your database is a cache. Stripe is the source of truth. If your reconciliation job finds a discrepancy, it does not ask which is "right." It assumes Stripe is right and updates the database.

The only exception: data that exists only in your database and not in Stripe. Your trial state (for no-credit-card trials), your usage counters, your feature gate checks – these are your domain, not Stripe's. For billing state (plan, status, period, cancellation), Stripe is authoritative.

---

8. Billing Edge Cases That Will Bite You

These are the scenarios most developers do not think about until they happen in production. Thinking through them now saves you from scrambling later.

Failed Payment Retry Cycle

When a payment fails, Stripe does not immediately give up. It enters a retry cycle governed by your "Smart Retry" settings (configurable in Dashboard > Settings > Billing > Automatic collection). The default is roughly: retry after 1 day, again after 3 days, again after 5 days, then mark the subscription as canceled.

During this retry cycle, the subscription status is `past_due`. Each retry generates a new `invoice.payment_failed` event if it fails and an `invoice.paid` event if it succeeds.

What to show the user: a persistent banner in the dashboard saying "Your payment failed. Please update your payment method." with a link to the Customer Portal. Do not show the specific failure reason (those can contain sensitive info), just a clear call to action. Stripe will also send the user automated emails about the failure (if you have enabled receipt emails in the dashboard).

The key insight: most `past_due` subscriptions resolve themselves. The user's bank approves the retry, or the user updates their card. Aggressive lockouts during the retry period cause unnecessary churn.

Customer Disputes / Chargebacks

A chargeback happens when a customer tells their bank to reverse a charge. This is rare for SaaS (more common in e-commerce) but it does happen. Stripe sends a `charge.dispute.created` event.

You do not need to handle this in code for launch. Chargebacks are handled in the Stripe Dashboard where you can submit evidence. But be aware that a successful chargeback means you lose the money *and* Stripe charges a $15 dispute fee.

Prevention is better: use clear billing descriptors (the text that appears on credit card statements – configure this in Stripe Dashboard), send receipts, and have a clear cancellation policy. If your billing descriptor says "FLAGLINE.DEV" instead of "STRIPE* UNKNOWN", users are less likely to dispute a charge they do not recognize.

Subscription Changes: Immediate vs. Period End

When a user changes their subscription through the Customer Portal, the change can be either immediate or scheduled for the period end. The behavior depends on your Customer Portal configuration and whether the change is an upgrade or downgrade.

The standard configuration: upgrades are immediate (the user wants the new features now), downgrades take effect at the end of the current billing period (the user already paid for the current period).

Your code needs to handle both. The `customer.subscription.updated` event fires in both cases. For immediate changes, the subscription's items reflect the new price immediately. For scheduled changes, the subscription's `schedule` object describes what will change and when. You can detect this by checking if the event's subscription object shows a `pending_update` or if `cancel_at_period_end` is set.

In practice, for Flagline, let the Customer Portal handle the UX of this and just sync whatever state Stripe reports. Your `subscription.updated` handler writes the current state of the subscription

object to your database, regardless of whether the change was immediate or scheduled.

Tax Implications

If you sell to customers in the US (or EU, or increasingly anywhere), you may need to collect sales tax or VAT. This is a legal requirement, not a product feature. The good news: Stripe Tax handles this for you.

Enable Stripe Tax in the Dashboard, register your tax nexus (the jurisdictions where you have tax obligations), and pass `automatic_tax: { enabled: true }` in your Checkout Session. Stripe calculates the correct tax rate based on the customer's billing address, adds it to the invoice, and even handles the filing in supported jurisdictions.

Do not try to build tax calculation yourself. Tax law is more complex than billing code. Let Stripe handle it. The cost ($0.50 per transaction for Stripe Tax) is trivial compared to the engineering and legal cost of getting it wrong.

Coming from Laravel: Cashier v14+ supports Stripe Tax via `$user->taxRates()` and the `calculateTaxes()` method. Same concept: let Stripe do the math.

Refund Handling

Refunds are typically a manual operation: a customer contacts support, you decide to issue a refund, and you do it through the Stripe Dashboard. For an automated refund flow (e.g., a "30-day money-back guarantee" button), you would call `stripe.refunds.create()` with the payment intent ID.

The key thinking: refunds do not automatically change subscription status. If you refund a payment but do not cancel the subscription, the subscription remains active and the next invoice will charge the customer again. If you want to refund *and* cancel, do both: issue the refund and cancel the subscription.

Stripe fires a `charge.refunded` event when a refund is processed. You typically do not need to handle this in your webhook for Flagline – the refund is a financial operation, not a product state change. If you cancel the subscription as part of the refund, the `customer.subscription.deleted` event handles the state change.

Coupons and Promotions for Launch Marketing

For launch, you will probably want a promotional discount. Stripe's coupon/promotion code system handles this:

- A **Coupon** defines the discount: "50% off for 3 months" or "$10 off once."
- A **Promotion Code** is a user-facing code ("LAUNCH50") that maps to a coupon.

Create these in the Stripe Dashboard or via the API. In your Checkout Session, set `allow_promotion_codes: true` and Stripe shows a "Have a promo code?" input on the checkout page. That is all you need on the code side.

The thinking: promotion codes are a Stripe feature, not a Flagline feature. You do not need to validate codes yourself, track redemption counts yourself, or calculate discounted amounts yourself. Stripe does all of it. Your only decision is the discount structure.

Annual Billing

Offering annual billing with a discount is standard SaaS practice. The typical discount is "2 months free" – so annual Starter is $150/year (instead of $15 x 12 = $180) and annual Pro is $290/year (instead of $29 x 12 = $348).

In Stripe, this is simply two Prices per Product: one monthly and one annual. The annual Price is not a "discount on the monthly price" – it is a separate Price with a different amount and `recurring.interval = 'year'`.

In your UI, show both options with a toggle and display the savings clearly: "$12.50/mo billed annually (save $30)." The monthly equivalent is calculated client-side for display purposes – Stripe does not care, it just sees a $150/year charge.

Annual subscriptions introduce a wrinkle for downgrades and cancellations: the user has prepaid for the full year. Refunding partial years is messy. The standard approach is to let the annual subscription run to completion and not offer mid-year downgrades. Or, offer a prorated refund for the remaining months. Decide your policy before launch and document it in your terms of service.

Multi-Currency

If you serve international users, you have a decision: single currency (USD) or multi-currency?

Start with USD only. It is simpler. International customers pay in USD, their bank handles the conversion, and you do not have to think about exchange rates, currency-specific pricing, or multi-currency accounting. Most early-stage SaaS products (and many large ones) are USD-only.

If and when you add multi-currency, Stripe supports creating multiple Prices per Product in different currencies. You detect the user's location (via IP or billing address) and show the appropriate Price. Stripe handles the rest.

The complexity with multi-currency is not technical – it is business. You need to decide on prices for each currency that account for purchasing power parity and exchange rate fluctuations. $29/month might be reasonable in the US but expensive in India. These are pricing strategy decisions, not engineering decisions. Defer them until you have international traction.

---

9. The "Don't Build Billing Yourself" Principle

Let Stripe Handle as Much as Possible

There is a strong temptation, especially for engineers, to build custom UI for billing management. Custom plan selection page, custom card update form, custom invoice history, custom cancellation flow. Resist this temptation. Every minute you spend building billing UI is a minute not spent on feature flags, which is your actual product.

Stripe provides three hosted experiences that cover almost everything:

Stripe Checkout for payment collection. Handles card input, validation, 3D Secure, localization, Apple Pay, Google Pay, and dozens of local payment methods. PCI compliant. Optimized for conversion across millions of transactions. You cannot beat it.

Stripe Customer Portal for subscription management. Users can change plans, update payment methods, view invoices, and cancel. Configurable in the Stripe Dashboard. Looks professional. You link to it with a single API call (`stripe.billingPortal.sessions.create()`). The user manages their billing on Stripe's hosted page and is redirected back to your app when done.

Stripe Hosted Invoices for invoice delivery and payment. Stripe generates invoices, sends them via email, and hosts a page where the customer can view and pay them. You do not need to build an invoice viewer.

The only billing UI you need to build in Flagline:

1. **The pricing/plan selection page.** This is a marketing page, not a billing page. It shows your plans, their features, and "Upgrade" buttons that initiate Stripe Checkout. You need this in your own design because it is part of your brand.
2. **A billing dashboard page.** Shows the current plan, usage summary, and links to the Customer Portal ("Manage Subscription" button) and Checkout (for free users who want to upgrade). This is a light page – mostly reading state from your database and linking to Stripe's hosted experiences.
3. **Usage warnings and trial banners.** Inline UI components that show "You are approaching your limit" or "Your trial ends in 3 days." These are product UX, not billing UI.

Everything else – card management, invoice history, plan switching, cancellation – is handled by the Customer Portal.

The Billing Portal Pattern

The Customer Portal interaction is a simple redirect flow, identical to Checkout:

1. User clicks "Manage Subscription" in your billing dashboard.
2. Your backend creates a Portal Session with `stripe.billingPortal.sessions.create()`, passing the customer ID and a return URL.
3. Stripe returns a URL. You redirect the user to it.
4. The user manages their billing on Stripe's page.
5. When done, Stripe redirects them back to your return URL.
6. Meanwhile, any changes the user made (plan swap, cancellation, card update) trigger webhooks that update your database.

Configure the Portal in the Stripe Dashboard: which plans can be switched between, whether cancellation is allowed, whether invoices are shown. This configuration is done once and applies to all Portal sessions.

Coming from Laravel: Cashier provides `$user->billingPortalUrl(route('dashboard'))`. That is exactly what `stripe.billingPortal.sessions.create()` does. Cashier also had its own Blade-based billing views, but Stripe's hosted Portal has largely replaced them. Even in Laravel-land, the recommendation is to use Stripe's hosted Portal.

When You Might Need Custom Billing UI

There are a few scenarios where Stripe's hosted experiences are not enough:

- **Deeply integrated upgrade flows.** If you want the upgrade prompt to appear inline when a user hits a limit (not as a redirect to another page), you might embed Stripe Elements for a seamless experience. This is significantly more work and usually not worth it for launch.
- **Complex pricing models.** If you have per-seat pricing, usage-based billing, or multi-product subscriptions, the standard Checkout and Portal might not fully support your model. Flagline does not have these, so this is not a concern.
- **White-labeled billing.** If you are building Flagline for resale (white-label), the Stripe- branded checkout and portal might not work. But this is an unlikely scenario for a developer tool.

For Flagline at launch, Stripe's hosted experiences are more than sufficient. Build the minimum billing UI described above and spend your time on the SDK, the evaluation engine, and the dashboard features that make Flagline valuable.

Putting It All Together: The Decision Checklist

Before you start writing billing code, make these decisions explicitly. Each one affects your implementation:

1. **Checkout approach:** Stripe Checkout (hosted). Unless you have a strong reason for a custom form, default to Checkout.
2. **Customer creation timing:** Lazy (on first checkout). Clean for freemium products with many non-paying users.
3. **Trial type:** No credit card required. Managed in your database, not Stripe. 14-day Pro trial.
4. **Grace period for failed payments:** 7 days. During grace period, full access with a warning banner.
5. **Downgrade behavior:** At period end (not immediate). The user already paid for the current cycle.
6. **Usage enforcement on evaluations:** Hard limit (429 response). Dashboard always accessible.
7. **Usage enforcement on counts (flags, projects):** Hard limit on creation. Existing items preserved on downgrade.
8. **Subscription management UI:** Stripe Customer Portal. No custom plan-switching or card-management UI.
9. **Tax:** Stripe Tax from day one if selling to US customers. Defer multi-currency.
10. **Annual billing:** Yes, with 2-months-free discount. Separate Prices per Product.

With these decisions made, the implementation flows naturally. Create Products and Prices in Stripe. Build the checkout endpoint. Build the webhook handler. Build the feature gate service. Build the usage tracking. Wire it all together. The code is the straightforward part when the thinking is done.

Appendix A: The Critical File Map

When you implement billing in Flagline, you will end up with roughly this set of files:

```
src/lib/stripe.ts                    -- Stripe client singleton
src/lib/billing/plans.ts                -- Plan limits config + price-to-
plan mapping
src/lib/billing/feature-gate.ts      -- "Can this tenant do this?" service
src/lib/billing/usage-tracking.ts    -- Redis-based evaluation counter
src/lib/billing/stripe-customers.ts  -- Get-or-create Stripe customer
src/lib/billing/trial.ts          -- Trial lifecycle (start, check, expire)
src/lib/billing/grace-period.ts      -- Grace period calculations
src/lib/billing/reconciliation.ts    -- Stripe <-> DB sync job
src/lib/billing/webhook-handlers/    -- One file per event type
src/app/api/billing/checkout/        -- Checkout Session creation
src/app/api/billing/portal/          -- Portal Session creation
src/app/api/webhooks/stripe/         -- Webhook receiver
```

This is roughly 12-15 files. Each one has a single responsibility. The billing module is self- contained: the rest of your application imports from `@/lib/billing` and does not interact with Stripe directly.

Appendix B: Stripe CLI Quick Reference

For local development, these are the commands you will use daily:

```
stripe login                                                          --
authenticate with your Stripe account
stripe  listen  --forward-to  localhost:3000/api/webhooks/stripe     --
forward webhooks to local dev
stripe trigger checkout.session.completed                             --
simulate a completed checkout
stripe trigger invoice.payment_failed                                 --
simulate a failed payment
stripe trigger customer.subscription.deleted                          --
simulate a cancellation
stripe products list                                -- list your products
stripe prices list --product prod_xxx                                 --
list prices for a product
stripe events list --limit 10                            -- recent events
stripe logs tail                                      -- live API request log
```

Appendix C: Production Checklist

- ☐ Switch from test keys (`sk_test_`) to live keys (`sk_live_`) via environment variables
- ☐ Register production webhook endpoint in Stripe Dashboard (not the CLI)
- ☐ Use the production webhook secret (different from the CLI-generated test secret)
- ☐ Configure the Customer Portal in Stripe Dashboard (allowed plan changes, cancellation, etc.)
- ☐ Configure Smart Retry settings (Dashboard > Settings > Billing > Automatic collection)

- ☐ Enable Stripe Tax if selling to US customers
- ☐ Enable billing emails in Stripe (receipts, failed payment notifications, renewal reminders)
- ☐ Test the full checkout flow end-to-end with Stripe test card numbers
- ☐ Test webhook handling for all critical event types using Stripe CLI triggers
- ☐ Test proration: upgrade and downgrade mid-cycle
- ☐ Test trial expiration flow
- ☐ Test payment failure and grace period flow
- ☐ Deploy the reconciliation cron job
- ☐ Set up monitoring for webhook failures (Dashboard > Developers > Webhooks)
- ☐ Load-test the evaluation endpoint with usage tracking under concurrency
- ☐ Set a clear billing descriptor (Dashboard > Settings > Public details > Statement descriptor)

Appendix D: Key Differences from Laravel Cashier

Concern	Laravel Cashier	Flagline (Raw Stripe SDK)
Installation	`composer require laravel/cashier`	`npm install stripe`
Customer creation	`$user->createAsStripeCustomer()`	`stripe.customers.create({...})`
Checkout	`$user->checkout('price_xxx')`	`stripe.checkout.sessions.create({..`
Portal	`$user->billingPortalUrl()`	`stripe.billingPortal.sessions.creat`
Subscription check	`$user->subscribed()`	Query Subscription table: `status IN ('active','trialing')`
Plan check	`$user->subscribedToPrice('price_xxx')`	Compare `tenant.currentPlan` field
Swap plan	`$user->subscription()->swap()`	`stripe.subscriptions.update({...})`
Cancel	`$user->subscription()->cancel()`	Portal handles it, or `stripe.subscriptions.update({ cancel_at_period_end: true })`
Trial check	`$user->onTrial()`	`tenant.isTrialing && tenant.trialEndsAt > now`
Webhooks	Extends `CashierController`, override methods	Route handler with switch on `event.type`
Signature verification	Automatic via middleware	Manual `stripe.webhooks.constructEvent()`
Grace period	`$user->subscription()->onGracePeriod()`	Custom `isInGracePeriod()` function
Database sync	Cashier webhook controller	Your webhook handlers + reconciliation job

Concern	Laravel Cashier	Flagline (Raw Stripe SDK)
Feature gating	Laravel Gates/Policies	Custom `canCreateFlag()` etc.

The core takeaway: Cashier is a convenience wrapper. Every method maps 1:1 to a Stripe API call. By working with the SDK directly, you trade convenience for understanding and control. For a production SaaS where billing is a core concern, that trade is worth it. You will debug billing issues faster, handle edge cases more precisely, and never be blocked by an abstraction that does not support what you need.

Last updated: 2026-02-06