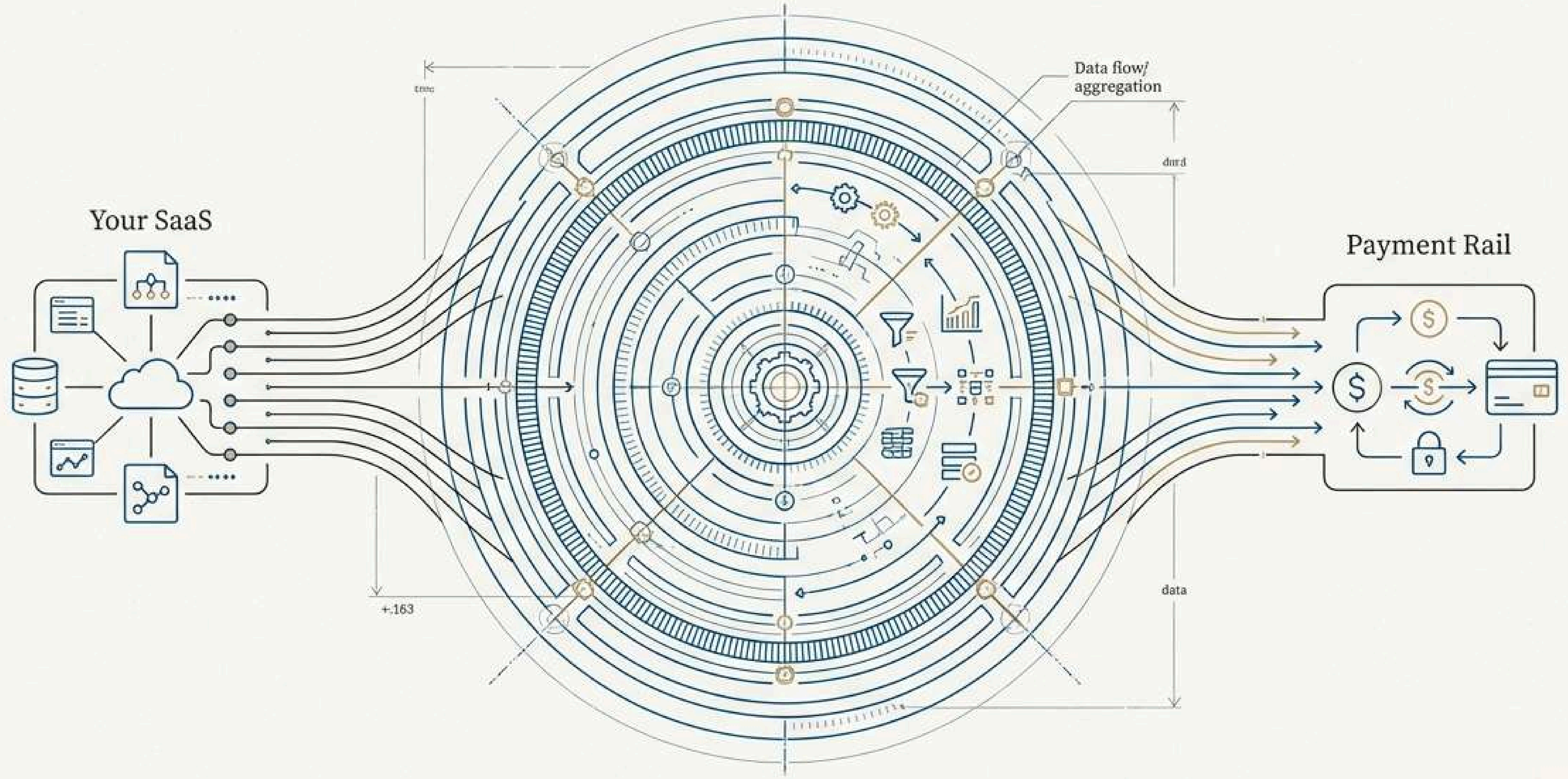
A Complete Backend for Modern Usage-Based SaaS

The Abstract Monetization Engine



Stripe is a Payment Rail, Not Your Monetization Logic

Most teams underestimate the engineering required to charge money correctly. Payment processing is only the first step.

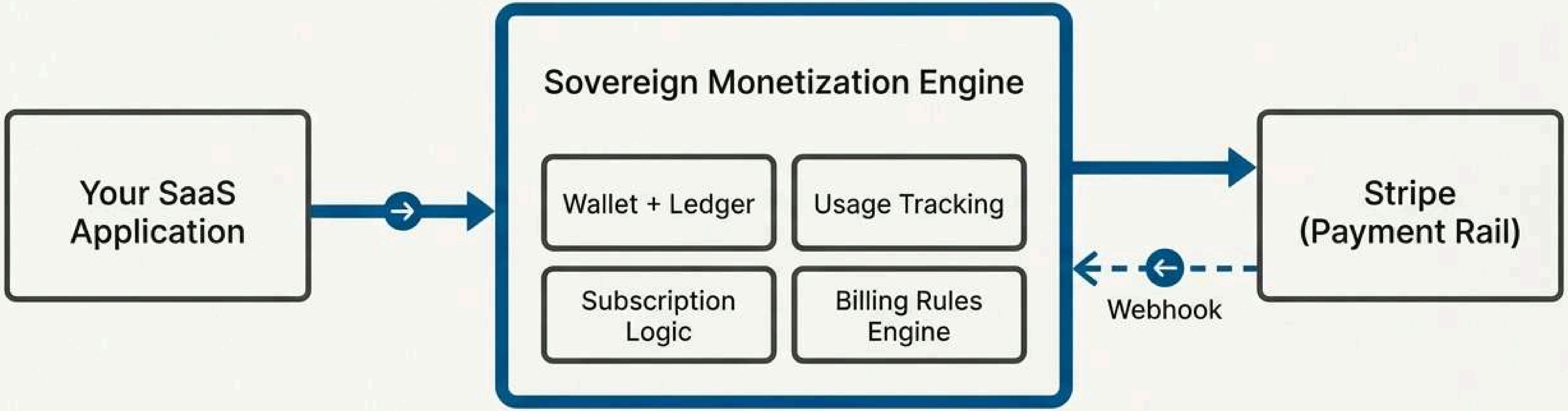
What Stripe Provides (Payment Primitives)

- Payment Processing
- Invoicing Primitives
- Basic Subscription APIs
- Raw Event Webhooks

What Modern SaaS Needs (Monetization Logic)

- Multi-tenancy & Team Roles
- A True Financial Ledger (Credits System)
- Granular Usage Metering & Charging
- Flexible, Configurable Pricing Rules
- Hardened, Idempotent Processing
- Comprehensive Audit Trails

The Solution: An API-First Monetization Engine



This is the missing layer that turns payment primitives into a complete, productized billing platform.

The Core Concepts: A Shared Language for Monetization

Projects
The tenant boundary. A container for a customer's workspace, users, and resources. Supports mapping to an `externalKey`.

Roles & Members
Governs access within a Project. Defines Owners, Admins, Members, and Viewers.

Wallets
The store of value. A per-user or per-project balance of credits.

Transactions
The immutable financial record. Every balance change is a ledger entry with `balanceBefore`, `balanceAfter`, and an `idempotencyKey`.

Usage Metrics
The record of consumption. Events representing billable actions like `api_calls` or `tokens_processed`.

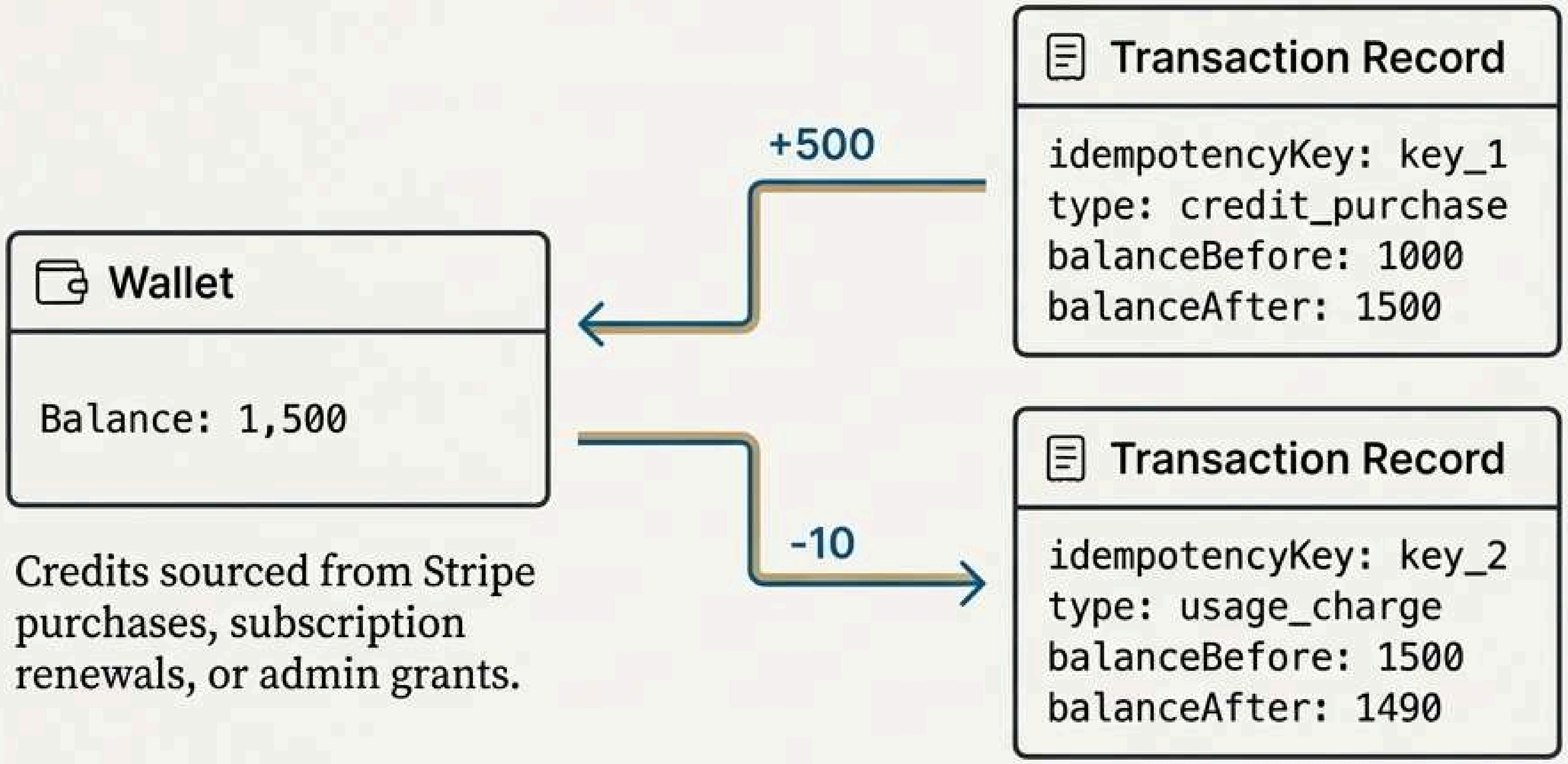
Built for Business: Multi-Tenancy, Roles, and Invites

The foundation for selling to teams and enterprises.

- ✓ **Tenant Isolation:** Create and manage distinct `Projects` for each customer.
- ✓ **Team Management:** Full member lifecycle (add, update role, remove) with four distinct roles: Owner, Admin, Member, Viewer.
- ✓ **Ownership Control:** Secure ownership transfer process.
- ✓ **Invitation Flow:** Seamlessly invite new users into projects via email, with full invite lifecycle management (create, resend, accept, peek).
- ✓ **Granular Access:** Route protection via guards for project members, admins, and 'billable members' (non-viewers).

The Financial Core: A True Ledger, Not Just a Balance Column

Correctly handling credits requires an immutable, idempotent transaction log.



Credits sourced from Stripe purchases, subscription renewals, or admin grants.

First-Class Records: Every balance change is a permanent `Transaction`.

Full Traceability: Records `balanceBefore` and `balanceAfter` for perfect auditability.

Built-in Safety: An `idempotencyKey` prevents double-charging and ensures safe retries.

Many credit systems fail because they don't implement a proper ledger. This one does.

The Pricing Brain: The Configurable Billing Rules Engine

Implement sophisticated pricing models without changing your application code. Billing rules are configured per-wallet and run automatically.

Supported Billing Modes

Daily Flat: A fixed daily charge.

Selected Days: Charge on specific days of the week/month.

Monthly Flat: A fixed monthly charge.

Usage-Based: Charge based on metered consumption.

Hybrid: A fixed base fee plus usage-based charges.

Cron Expression: Ultimate flexibility for custom recurring schedules.

Operational Excellence

Automated Scheduler: Runs periodically to execute due billing rules.

Distributed Lock: A DB-backed lease prevents double-runs in a multi-instance deployment.

Auditable Runs: `BillingRunLogs` provide a complete history of every execution, success or failure.

The Monetization Catalog: Credit Packs & Subscriptions

A complete system for managing both one-off purchases and recurring entitlements, fully integrated with Stripe.

One-Off Purchases (Credit Packs)

Define Products: Create, update, and manage credit packs (e.g., ‘1,000 credits for $10’).

Stripe Checkout: Generates Stripe Checkout Sessions for a seamless purchase experience.

Reconciliation: Credits are applied to the wallet upon successful payment via webhook, with a durable `Payment` record created.

Use Case: The classic ‘self-serve top-up’ model.

Recurring Entitlements (Subscriptions)

Define Plans: Create recurring subscription plans (e.g., ‘$50/month for 5,000 credits’).

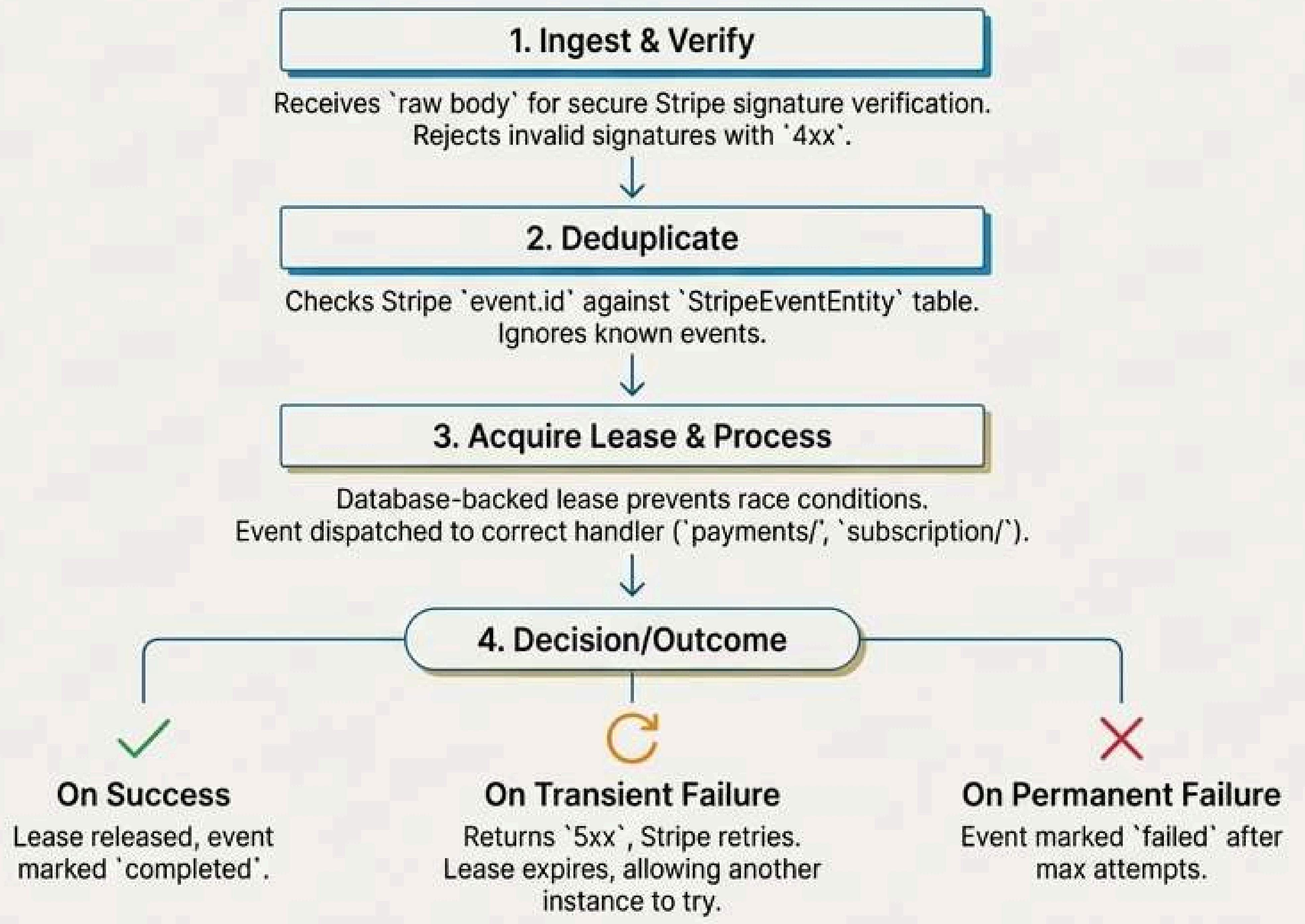
Stripe Sync: Manages Stripe Price and Plan objects.

Webhook-Driven Lifecycle: Handles `checkout.completed`, `invoice.paid`, and `subscription.deleted` events to automatically grant credits each interval.

Use Case: Entitlements-as-credits for API products.

Production-Grade Reliability: The Hardened Stripe Webhook Pipeline

Handling webhooks correctly is non-trivial. Our pipeline is built for resilience and correctness.



1. Ingest & Verify

Receives `raw body` for secure Stripe signature verification. Rejects invalid signatures with `4xx`.

2. Deduplicate

Checks Stripe `event.id` against `StripeEventEntity` table. Ignores known events.

3. Acquire Lease & Process

Database-backed lease prevents race conditions. Event dispatched to correct handler (`payments/`, `subscription/`).

4. Decision/Outcome

On Success

Lease released, event marked `completed`.

On Transient Failure

Returns `5xx`, Stripe retries. Lease expires, allowing another instance to try.

On Permanent Failure

Event marked `failed` after max attempts.

How It Works: Flow A — Buying a Credit Pack

1. **User Action:** A user selects a `Credit Pack` in your application's UI.
2. **API Call:** Your backend calls the Monetization Engine API to create a Stripe Checkout Session for the selected project.
3. **Stripe Checkout:** The user is redirected to Stripe to complete the purchase.
4. **Stripe Webhook:** Stripe sends a `checkout.session.completed` event to the `/stripe/webhook` endpoint.
5. **Engine Processing:** The hardened pipeline verifies, dedupes, and processes the event.
6. **Ledger Update:**
   - A `Payment` record is created, linking the Stripe session to the purchase.
   - The project's `Wallet` balance is increased.
   - An idempotent `Transaction` is recorded in the ledger, detailing the credit addition."

✓ **Outcome:** The user's balance is increased, with a complete, auditable trail across the Payment and Transaction ledgers.

How It Works: Flow C — Real-Time Usage Charging

1 **System Event:** Your application's service performs a billable action (e.g., an API call is made, an AI job completes).

2 **Record Usage:** Your backend calls the Monetization Engine API to `recordUsage` with a `metricKey` ('api_calls') and `units` (1). This creates a `UsageMetric` record.

3 **Charge Wallet:** Your backend immediately calls `chargeUsage` to deduct credits from the project's `Wallet`.

4 **Ledger Update:** The `Transaction` service creates an immutable ledger entry for the deduction, decrementing the wallet balance.

5 **Analytics Update:** Overview and aggregation endpoints now reflect the new usage and reduced balance.

Outcome: Pay-per-use is enforced instantly and auditable through both usage records and the financial ledger.

How It Works: Flow D — Automated Scheduled Billing

1 **Scheduler Runs**
The engine's internal scheduler runs on a cron schedule (e.g., every 5 minutes).

2 **Acquire Lock**
The scheduler acquires a database-backed lease. This ensures that even in a multi-server deployment, only one instance will perform the billing run.

3 **Find Due Rules**
The scheduler queries for all `Billing` rules that are due to be executed based on their configuration (e.g., it's the 1st of the month for a `Monthly Flat` rule).

4 **Execute Rule**
For each due rule, the engine executes the logic: it calculates the charge (e.g., base fee + aggregated usage) and deducts the amount from the associated `Wallet`.

5 **Log & Record**

- A `BillingRunLog` is created to record the execution, its status, and outcome.
- An immutable `Transaction` is written to the ledger for the balance deduction.

✓ **Outcome:** Flexible, automated billing runs reliably and safely in a production environment.

A Clean, Maintainable Architecture

Tech Stack & Infrastructure

- **Framework:** NestJS (TypeScript)
- **Database:** MySQL with TypeORM & Migrations
- **Authentication:** JWT (Passport.js) & bcrypt
- **API Docs:** Swagger (OpenAPI)
- **Scheduling:** @nestjs/schedule
- **Security:** Helmet, CORS, Request Throttling
- **Payments:** Stripe SDK

High-Level Module Map

- `auth/` - Authentication & Guards
- `project/` - Tenancy, Members & Invites
- `wallet/` & `transaction/` - Financial Core & Ledger
- `usage/` - Usage Metering
- `payments/` & `subscription/` - Stripe Products
- `billing/` - Rules Engine & Scheduler
- `stripe/` - Hardened Webhook Ingestion
- `audit/` - Audit Logging

The modular design mirrors the feature set, making the system easy to maintain and extend. Source Serif Pro Regular

The Roadmap: The Path to a Fully Productized Platform

The core engine is production-ready. These next steps focus on operator experience and commercial polish.

Developer Experience

- Per-Project API Keys
- Client SDKs (Node.js, Python)
- Tenant-facing Webhooks

Management UI

- Customer & Usage Dashboards
- Billing Rule Configuration UI
- Audit Log & Billing Run Viewer
- Webhook Replay Tooling (Admin)

Commercial Polish

- Coupons & Promotions
- Proration for Plan Changes
- Low Balance & Payment Failure Alerts
- Dunning & Recovery Email Sequences

Why This Architecture Succeeds

Ledger-First Architecture

Ensures financial-grade correctness, auditability, and debuggability. Every change in value is an immutable, traceable event.

Flexible Billing Engine

Decouples pricing logic from application code, allowing you to future-proof your business model and experiment with pricing without re-engineering.

Production-Grade Reliability

Built with an obsessive focus on real-world failure modes, using idempottency keys, hardened webhooks, and distributed locks for operational peace of mind.