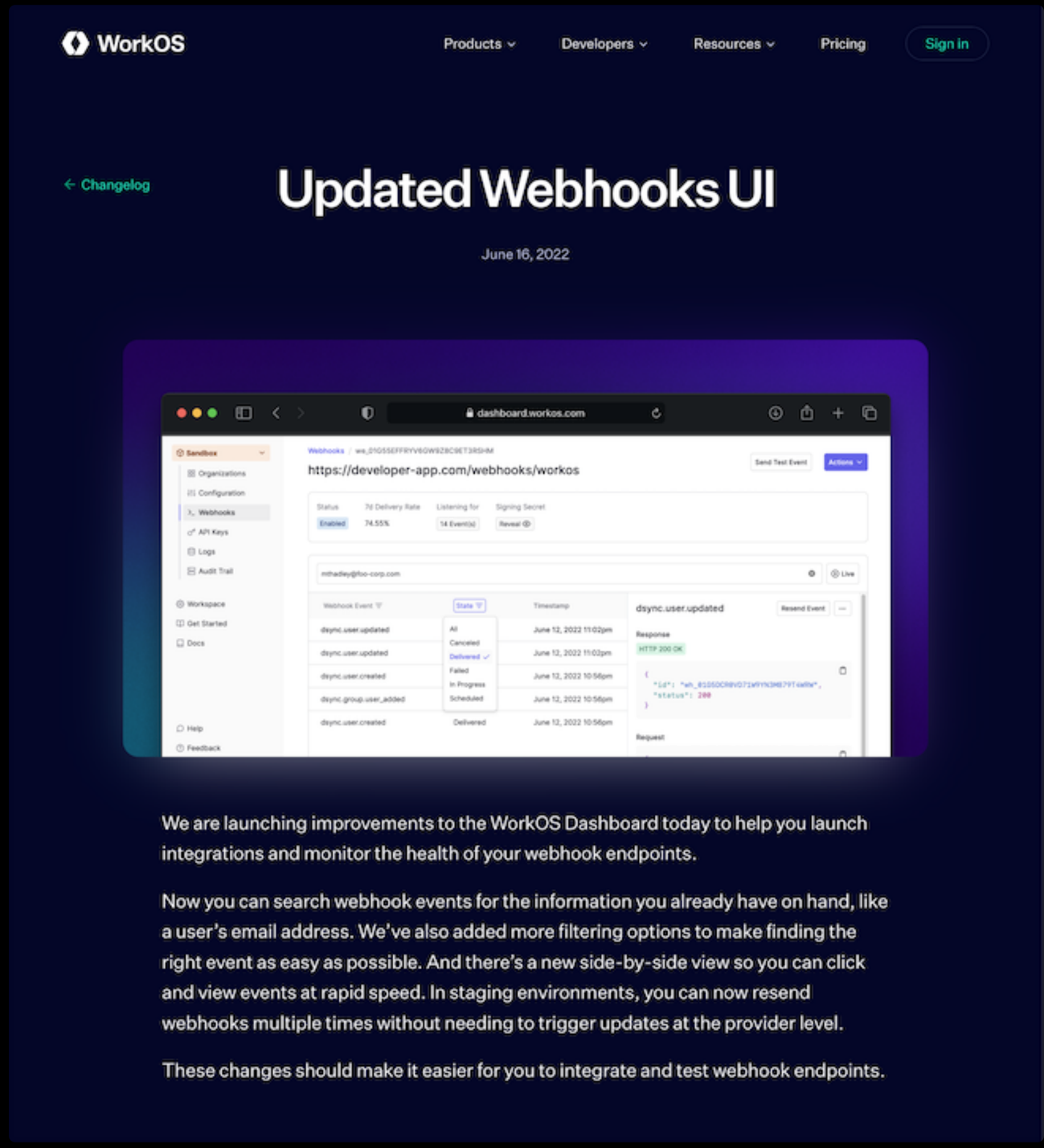
WorkOS Webhooks

Redesigning the integration and debugging experience

Product Designer · Early 2022

A redesigned dashboard turned a flat event list into a real integration-testing tool and introduced a column-inspection pattern to the design system.



Webhooks are how WorkOS pushes real-time updates. When they break, the provider's dashboard is the only debugging window

WorkOS sells developer infrastructure for enterprise auth — SSO, directory sync, user management.

Webhooks are the inverse of REST — instead of customer apps polling for updates, WorkOS pushes events when something changes.

The advantage: real-time reaction without polling overhead.

The cost: when something goes wrong, customers can't debug locally — they have to use whatever surface the provider gives them.

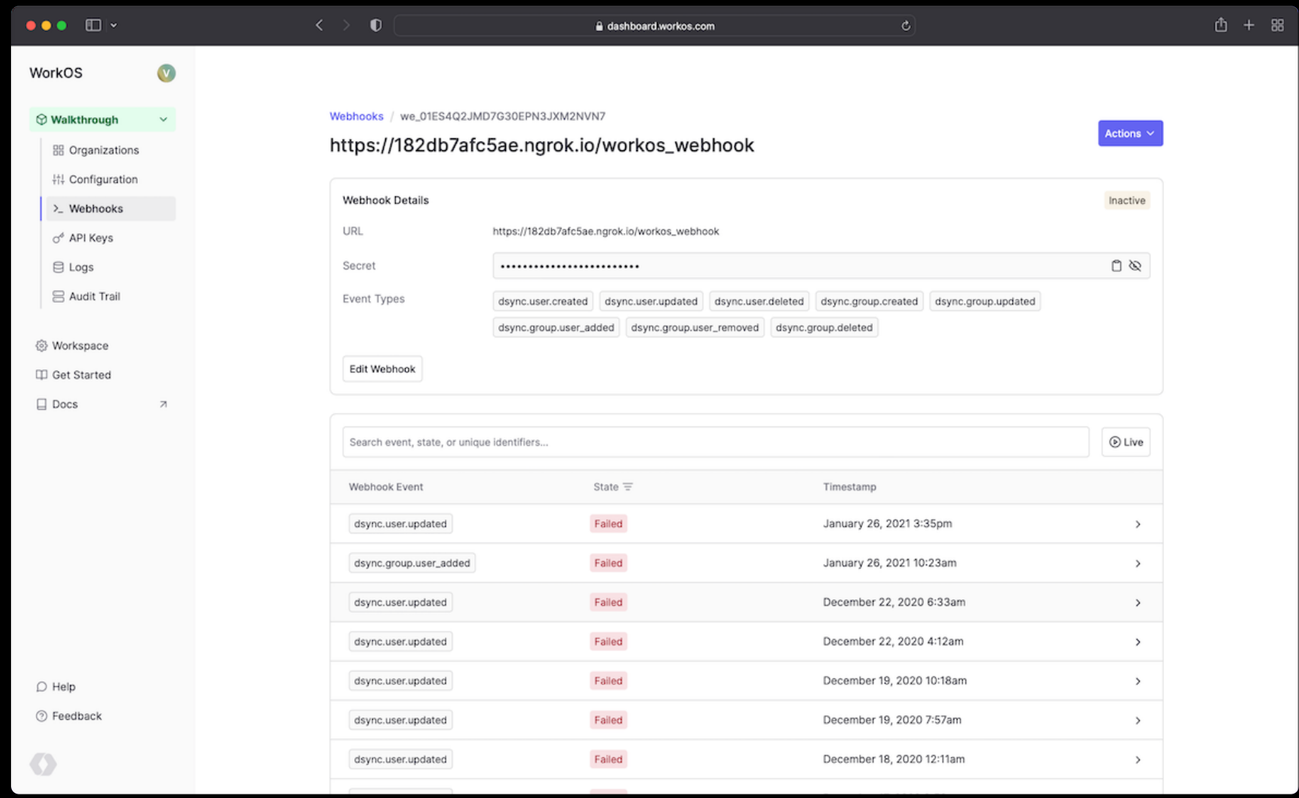
Pull (REST) vs. push (webhooks). When webhooks break, only the provider's dashboard can tell you what happened.

Developers couldn't find broken events — and testing handlers required recreating upstream state every time

Three independent signals were pointing at the same problem:

- **Customer Support tickets** — developers couldn't locate the relevant event when a customer reported an issue by email or user ID
- **Dogfood friction** — internal devs hit the same wall during integration; recreating upstream state to test was tiresome and produced false negatives
- **Integration funnel** — time-to-first-successful-webhook was longer than it should have been

The pre-redesign dashboard surfaced a flat event list and high-level connection status. That was it.



Sole designer on a team of four — partnering directly with Customer Support for discovery

My role

- Design end-to-end: discovery, prototyping, UX copy, production-ready assets
- Led discovery by partnering with Customer Support
- Pushed for the column-inspection pattern against the design-system default

The team

- **PM** — Stephen Haney (stakeholder alignment, engineering scoping)
- **Engineering** — David Liu, Vitor Capretz
- **Research partner** — Customer Support (no dedicated researcher on the project)

Customer Support surfaced the key insight

Discovery had three parts:

1. **Surface-area inventory** — every place webhook information appeared: dashboard, error emails, transactional emails, error states, docs, the creation flow
2. **Pattern research** — established webhook debugging interfaces (Stripe, Svix, others in dev-tools)
3. **JTBD with Customer Support** — actual jobs developers were trying to do

The key finding: resend. Developers were repeatedly recreating upstream states to debug integration. The friction was visible in support tickets — but not in pattern research.

"Easier to develop" lost to a JTBD argument backed by precedent

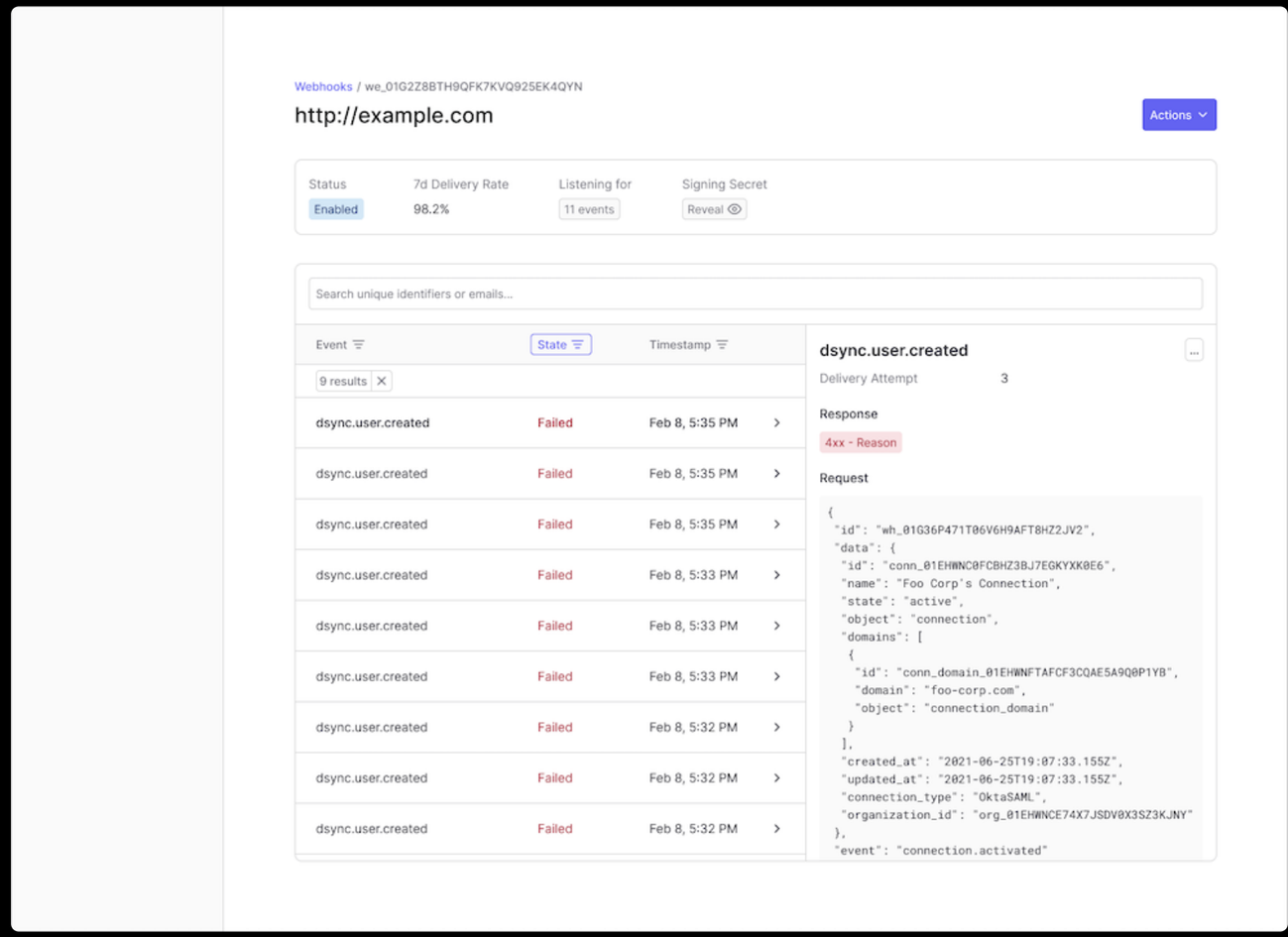
The dashboard's default detail pattern was a **drawer modal**. Cheap to build, well-supported.

I pushed for **column inspection** instead — persistent list, persistent inspector, click-through without losing context.

Why it mattered: Debugging webhooks is high-volume scanning, not single-event reading. Drawers add a navigation tax on every click. At the scale of events discovery surfaced, that tax was prohibitive.

How the argument was won: JTBD signal (volume of inspections) + pattern precedent from adjacent dev tools (Stripe, Svix). Engineering agreed to the more expensive path.

The durable outcome: The frontend team later codified column-inspection into the design system. The pattern outlived the feature.

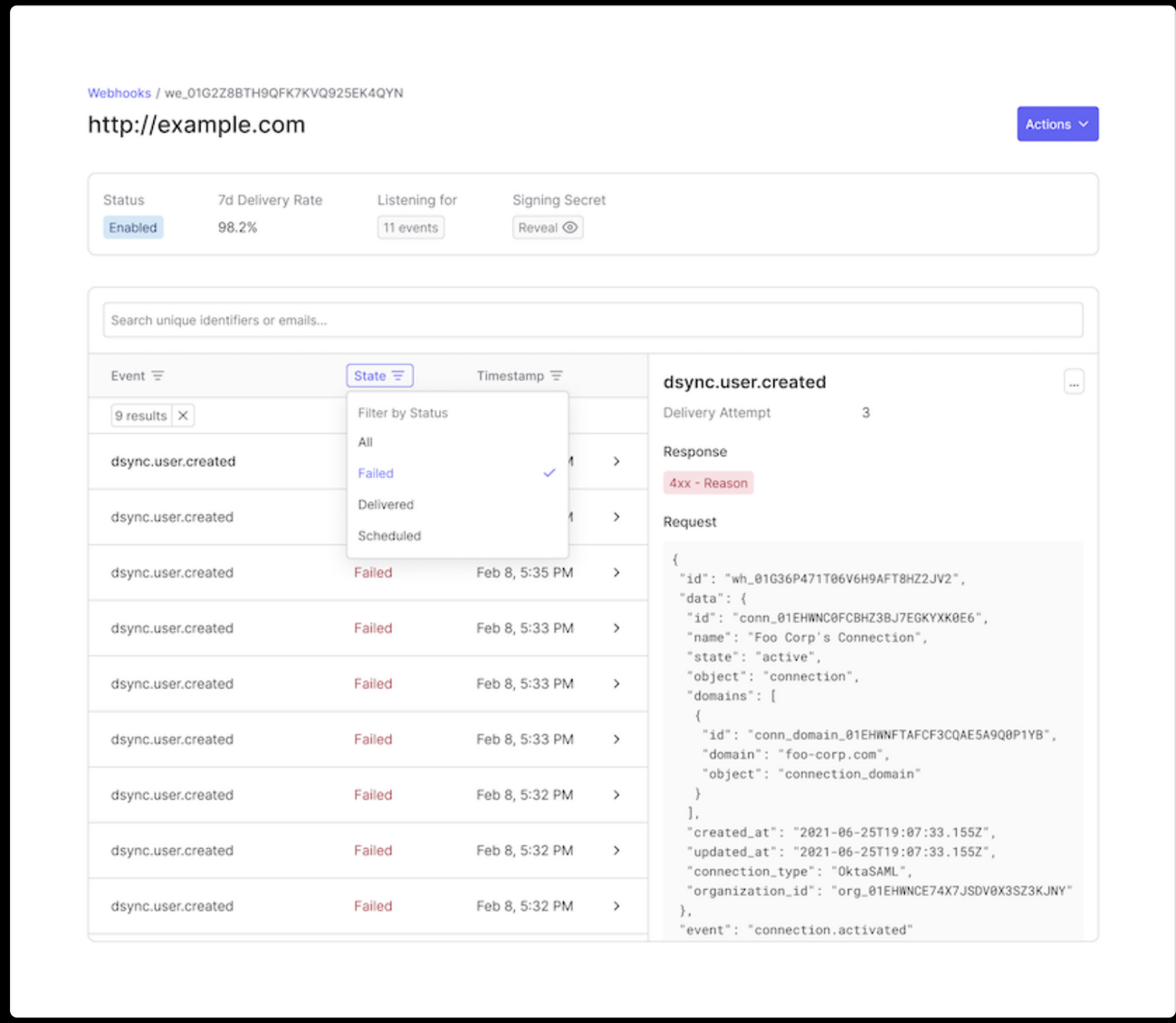


WorkOS Case Study

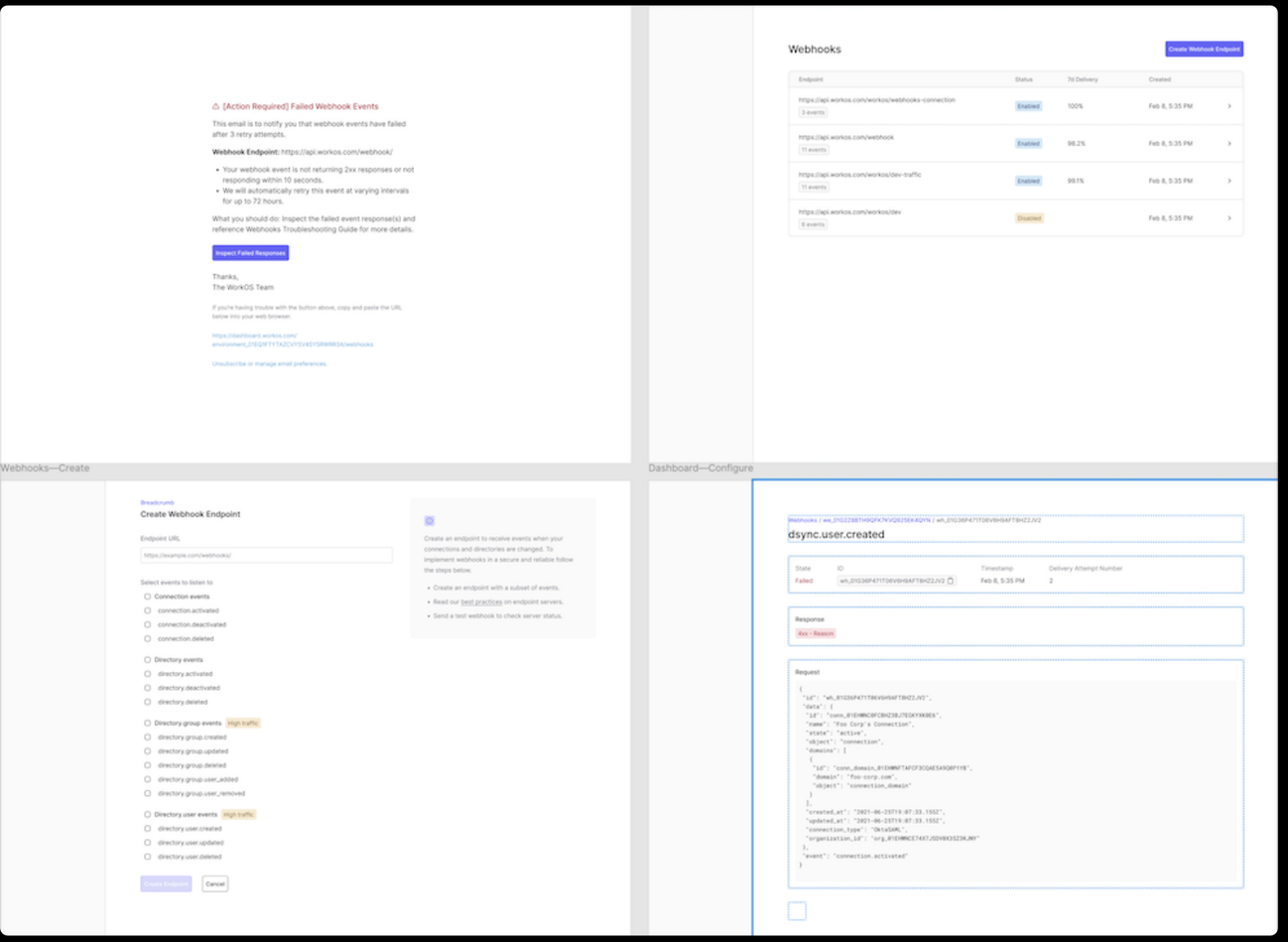
Search, filters, inspector, and resend — designed as one connected system

Each capability narrowed the developer's path to the event they needed:

- **Search by user data** — scoped to email, ID, event state and type. CS's "find by who" need anchored the scope; I resisted overbuilding into a generic search engine.
- **Filters by webhook type and state** — composed with search. Search narrows *who*, filters narrow *what kind*.
- **Side-by-side inspection** — the column pattern, with state persisting across clicks for fast comparison.
- **Manual resend in staging** — closing the integration-testing loop without auto-retrying behind the developer.



Discovery extended beyond the dashboard — endpoint creation, error emails, best-practices docs



The surfaces a developer actually touches during integration. The dashboard wasn't the integration experience — it was the most visible piece.

Endpoint creation flow

- Highlighted high-traffic events so customers could prepare
- Enabled multiple events at once
- Linked to a best-practices guide for handling, security, and testing

Error notification emails

- Restructured around making retry behavior visible
- "Right information at the right time" — is this retry ongoing, exhausted, or successful?

Support volume dropped, integration time improved — and the pattern outlived the feature

What changed

- Support ticket volume on webhook-related issues dropped
- Time-to-first-successful-webhook improved
- Updated documentation, linked from the creation flow, compounded the onboarding effect

Directional signals, not quantified — but consistent across CS, the integration funnel, and customer feedback.

The durable outcome: the column-inspection pattern entered the design system and propagated to other dashboard surfaces.

What I'd do differently

Push for a larger scope sooner. The resend feature surfaced late from JTBD work and was added under time pressure with scope reduced. The integration-testing workflow deserved to be front-loaded.

What this taught me

For developer infrastructure, the integration and debugging experience is a primary surface of the product — not a quality-of-life concern. It either elevates the product over competitors or stands as a barrier to customer success.

Debugging is the product.