FIELD GUIDE 02 / FOR TEAMS WITH A PRODUCT THAT ALREADY WORKS

After It Works

Scaling a product without breaking it. What actually fails first, in what order, and what it costs to fix it late — from running a platform across thousands of live storefronts.

DORON SEGAL
CO-FOUNDER & CTO, PER DIEM (YC W21)
PREVIOUSLY SAILDRONE · OPENTABLE · BEATS MUSIC / APPLE

There is a stage nobody writes guides about. The product works. Customers pay. And everything that made you fast in year one starts quietly working against you.

This is a guide to that stage. It is deliberately concrete, because the failures are concrete: a query that was fine at ten thousand rows, a webhook that arrives twice, an AWS bill growing faster than revenue. None of it is exotic. All of it is expensive if you meet it late.

WHAT IS IN HERE

01

The thing that breaks first is your data model

Founders brace for traffic. Traffic is rarely the problem — cloud infrastructure is genuinely good at more requests. What breaks is the shape of your data, because that was designed when you had one customer type and one country and no idea what the second product would be.

Data model mistakes are the most expensive class of technical debt you can hold, because unlike code, you cannot refactor them in an afternoon. Every fix involves a migration, a backfill, and a window where both shapes must be valid.

— Anything you will later want to report on needs its own column, not a JSON blob.

— Anything that can be soft-deleted will be, so design for it up front.

— Money, time zones and identifiers deserve more thought than they got the first time. All three will bite.

The practical rule: when you feel the urge to add a nullable column to avoid a migration, that is the moment to do the migration properly instead.

02

Multi-tenancy is a safety property, not a feature

If you serve more than one business from one database, you have a class of bug that is categorically worse than an outage: showing one customer another customer's data. An outage costs you a bad day. This costs you the contract, and possibly the category.

Treat tenant scoping as an invariant enforced by the system, not a discipline enforced by code review. Every query gets a tenant identifier. Not most. Every one, including the ad-hoc one you run at midnight during an incident.

> FIELD NOTE
> The rule we hold is simple and unglamorous: no query ships without a merchant or store scope, and the review checks for it explicitly. It sounds like bureaucracy until you imagine explaining the alternative to a customer. Where the framework allows it, push the constraint down into a layer that cannot be forgotten rather than one that must be remembered.

The same logic applies to background jobs, exports, and analytics pipelines — the places where scoping is most often assumed and least often enforced.

03

Postgres stops being magic around here

Postgres will carry you far further than most people expect. It will also stop quietly, and the first symptom is usually a page that got slower rather than an error anyone notices.

— Sequential scans on growing tables. Fine at ten thousand rows. Not fine at a hundred million, and the transition is not gradual from the user's side.

— OFFSET pagination. Page one is instant, page four hundred reads everything before it. Move to keyset pagination before someone builds a report on it.

— N+1 queries. Invisible in development, catastrophic under concurrency.

— Indexes nobody uses. They are not free — every write pays for them.

Read your query plans before you argue about your database. The number of scaling conversations that end the moment someone actually looks at the plan is higher than anyone admits.

04

Migrations that do not take the site down

The migration that works on your laptop is not the migration that runs against a large live table. Adding a column with a default, adding a constraint, changing a type — each can take a lock that stops writes while thousands of people are trying to place orders.

The pattern that works is boring and takes longer: add the new thing, write to both, backfill in batches, read from the new thing, then remove the old one. Four deploys instead of one. Nobody notices, which is the point.

— Every migration has a written rollback, or it does not ship.

— Backfills run in batches with a pause, not as one statement.

— Deploy during a window you understand. Know your traffic shape by hour.

05

Caching is easy; invalidation is the job

Adding a cache is an afternoon. Knowing when to throw it away is the actual engineering, and stale-cache bugs are among the hardest to reproduce because the system is behaving exactly as written.

Two rules have saved us repeatedly. Give every cached key an explicit expiry, even one you think is permanent — unbounded keys are how memory becomes an incident. And make invalidation a deliberate, named step in whatever writes the underlying data, not a side effect someone remembers.

> FIELD NOTE
> If a customer ever sees a price, an item, or an availability window that no longer exists, the cache is not a performance layer any more. It is a correctness bug with a friendly name. Treat it with the seriousness of one.

06

Every third party will fail, so design for it

At scale you are not running one system. You are running yours plus every vendor in the path: payments, messaging, delivery, the point of sale. Each has its own outages, its own retries, and its own opinions about ordering.

Three habits cover most of the damage.

— Idempotency everywhere. Assume every inbound webhook arrives twice, out of order, and possibly weeks late. Key on the provider's identifier, not on yours.

— Reconciliation as a scheduled job. Do not trust that the two systems agree. Check, and alert on the difference.

— Degrade, do not collapse. When a vendor is down, decide in advance which part of your product keeps working.

> FIELD NOTE
> Two failures taught me more than any amount of design review. A duplicated webhook that printed the same ticket twice on a merchant's counter — the bug was invisible in our logs and extremely visible on their receipt roll. And carrier-level message rejections that never touched our code at all, but looked to customers exactly like our bug. Own the experience, not just your slice of the stack.

07

Observability you install before you need it

The worst time to discover you cannot see inside your system is during the incident. By then you are guessing in public.

You need three things, and most teams have one. Errors with enough context to identify the affected customer. Logs you can actually search under pressure. And product analytics that tell you whether the last release made anything worse — because a release can be error-free and still be a regression.

Then set the alerts that matter and delete the ones that do not. An alert everyone ignores is worse than no alert, because it teaches the team that alarms are noise.

08

The cost curve versus the revenue curve

There is a point where infrastructure cost starts growing faster than revenue, and it usually arrives without anyone deciding it should. Nobody owns the bill, so nobody sees the slope until it is a line item someone asks about.

The fix is unglamorous: know your cost per customer, and watch its direction rather than its absolute value. Most of the damage is concentrated in a few familiar places — data egress, over-provisioned instances kept for a launch that ended, log retention nobody chose, and analytics queries running against the production database instead of somewhere built for them.

— Separate analytical workloads from transactional ones before they collide.

— Right-size on measured load, not on the number that felt safe at 2am.

— Give one person the bill and let them ask stupid questions about it monthly.

09

Ownership beats process

Scaling the team is where most technical scaling problems are actually created. The instinct is to add process, because process feels like control. What actually works is clear ownership: one person accountable for each area, with enough context to make the call without a meeting.

My rule is to set the bar and the context, then get out of the way. If I am making the decision, I have either hired wrong or explained badly, and both are my problem rather than theirs.

Keep the gates you need and delete the rest. A quality gate before merge and an acceptance gate before customers see it will catch most of what matters. Everything beyond that tends to add latency without adding safety.

10

What to say no to

At this stage your constraint is no longer ideas. It is attention. The list of reasonable things you could build now exceeds what you can build for years, and every yes is a permanent maintenance obligation.

— The feature one loud customer wants that nobody else has asked for.

— The rewrite that fixes an architecture nobody has actually measured.

— The new platform before the current one is genuinely paying for itself.

— The hire made to fill a gap you have not yet defined.

Engineering serves the business. Architecture that does not move a business number is a hobby with a budget. The best question I know for a roadmap argument is simply: which number does this move, and by when?

CHECKLIST

If you only act on one page, act on this one.

- ■ Every query is scoped by tenant, including the ad-hoc ones.
- ■ I have read the query plan for the three slowest endpoints.
- ■ Pagination on large tables is keyset, not OFFSET.
- ■ Every migration has a written rollback and a batched backfill.
- ■ Every cache key has an expiry, and invalidation is a named step.
- ■ Inbound webhooks are idempotent and keyed on the provider's identifier.
- ■ A scheduled job reconciles our state against each vendor's.
- ■ Errors, searchable logs, and release-over-release analytics are all in place.
- ■ One person owns the infrastructure bill and reviews it monthly.
- ■ Each area has one accountable owner who can decide without a meeting.

Written from building and running the thing, not from advising on it. If you want a second pair of eyes on your specific version of this, the calendar is on doronsegal.com.