A Bootstrapper’s Diary

Visual diary of bootstrapping to 1M ARR

David Boyne

February 2026

About the Author

David Boyne is the founder of EventCatalog, an open-source tool that helps teams document their event-driven architectures. What started as a side project has grown into a bootstrapped business generating over $26K in monthly recurring revenue.

David builds in public, sharing the wins, the losses, and everything in between. A Bootstrapper's Diary is his collection of lessons learned on the road from $0 to $1M ARR.

- Twitter: https://twitter.com/boyney123
- GitHub: https://github.com/boyney123
- LinkedIn: https://linkedin.com/in/david-boyne
- Website: https://diary.boyney.io

Thanks for downloading. Hope you find it useful. Feel free to reach out.

Contents

Why I Quit My Job to Build EventCatalog and What I Learned About Taking the Leap

> What it really looks like to leave your job and go all-in on an open source project.

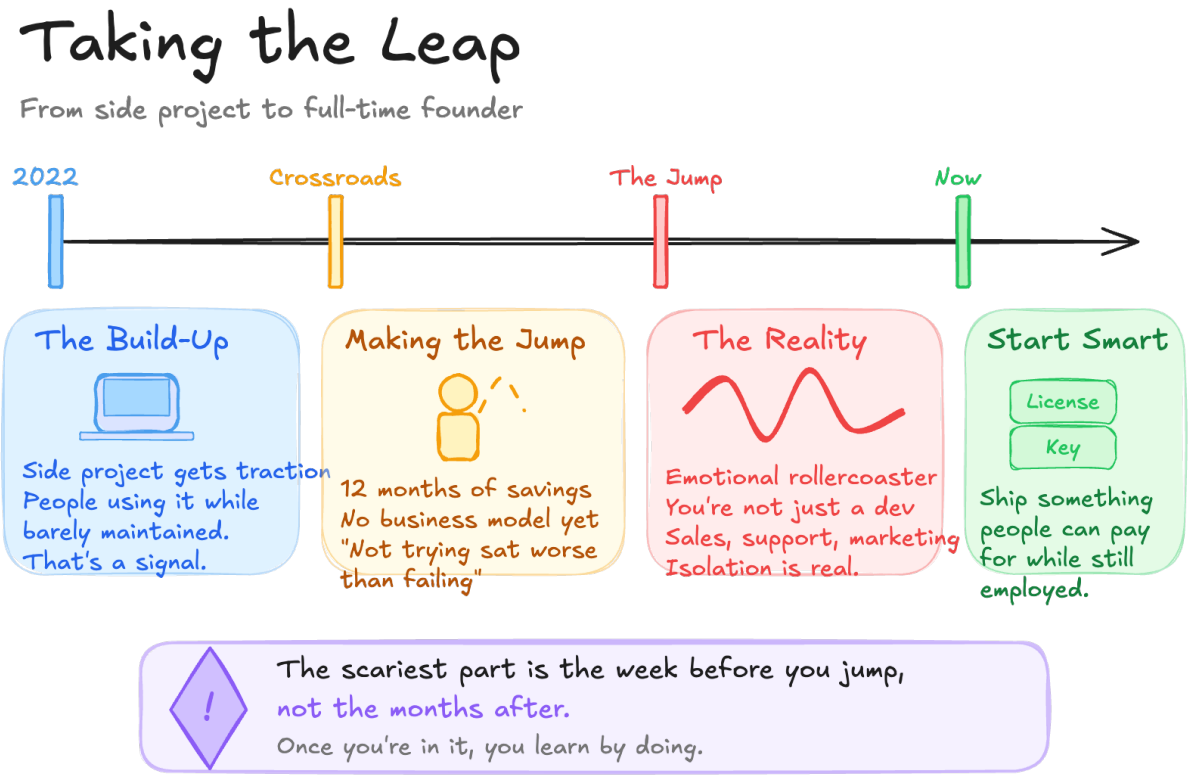

Figure 1: Why I Quit My Job to Build EventCatalog and What I Learned About Taking the Leap

I started EventCatalog as a side project in January 2022. For a couple of years I built it alongside a full-time developer job. I'd go to conferences and people would tell me they were using it, which was wild because I was barely maintaining the thing.

Then circumstances changed and I found myself at a crossroads. I remember sitting there thinking, "Could I actually do this full-time? Could I make open source sustainable?" I had no business model. I had about a year of savings. And honestly, I had no idea if anyone would ever pay for what I was building.

But the thought of not trying sat worse with me than the thought of failing. So I jumped. Looking back, there are things I'd do differently and things I got right by accident. Here's what I've learned so far.

The Build-Up

- **Pay attention to the signals while you still have a salary.** If people are using your project and you're barely maintaining it, that's worth noticing. Don't dismiss slow traction just because it's not dramatic.
- Think about where your best creative energy goes each week. If your side project gets the scraps after your day job drains you, you're probably underestimating what it could become with full attention.
- **Start tracking who's using your thing and why.** Talk to them. Ask what problems it solves. This is market research you can do for free while employed.
- Don't wait for a lightning bolt moment. For most of us the pull towards going full-time is gradual. If the idea keeps coming back, that's the signal.
- Something to consider: traction without attention is not a business. It's a hint. **Treat it like a hypothesis you need to test, not a guarantee.**

Making the Jump

- **Figure out if people will pay before you quit.** I jumped without a business model and I wouldn't recommend it. Even a small experiment (pre-sales, a waitlist, one paying user) gives you something real to build on.
- Save aggressively. I had about a year of runway and that gave me breathing room. **12 months minimum if you can manage it.** You'll make worse decisions under financial pressure.
- Reframe the risk for yourself. My thinking was simple: if it doesn't work, I can go get a job. That framing took the existential weight off it. You're not burning your career. You're taking a detour.
- Think about what you'd regret more. Not trying would have eaten at me. If the answer is the same for you, that tells you something.
- **Be honest about the financial stress.** Running on savings with zero income for months can push you into bad decisions. Desperation is not a business strategy. Plan around it.

The Reality on the Other Side

- **Expect the emotional rollercoaster and build habits to manage it.** Highs and lows come fast. Journal, exercise, talk to other founders. Don't let a bad Tuesday become a bad quarter.
- Accept that you're not just a developer anymore. Sales, support, marketing, community… you're doing all of it. **Block time for non-code work from day one** or it will eat your coding hours.
- Think about who you can lean on. The isolation catches people off guard. Find a community, a co-working space, a founder friend you can message when things feel hard.

- **Let the ethos carry you through the slow months.** I believe in building in public and earning developer trust through transparency. That belief kept me going when revenue was nowhere in sight. Know your "why" before you need it.
- Something people don't talk about enough: the fear fades faster than you'd expect. **The scariest part is the week before you jump, not the months after.** Once you're in it, you start learning by doing.

Start Before You're Ready (But Start Smart)

- **Ship something people can pay for while you still have income.** License keys, a paid tier, a support plan. Anything. I sell license keys for EventCatalog. You don't need a fully-grown SaaS on day one.
- Research open source business models before you commit to one. Open-core, dual licensing, managed services, support contracts. **Read what's working for other maintainers** and pick the one that fits your project.
- While you still have a job, build your audience. Write about what you're learning. Share your progress. Every person who follows your journey now is a potential customer later.
- Think about what "sustainable" means for you specifically. It might not be six figures right away. **Covering your expenses while you build something you're proud of is a perfectly valid first milestone.**
- The best way to learn this stuff is to dive in. Books and blog posts can only take you so far… the real learning happens when it's your money, your time, and your project on the line.

My First Sale and What I Learned About Turning Free Users Into Paying Customers

> How I went from free open source project to someone actually paying me for it.

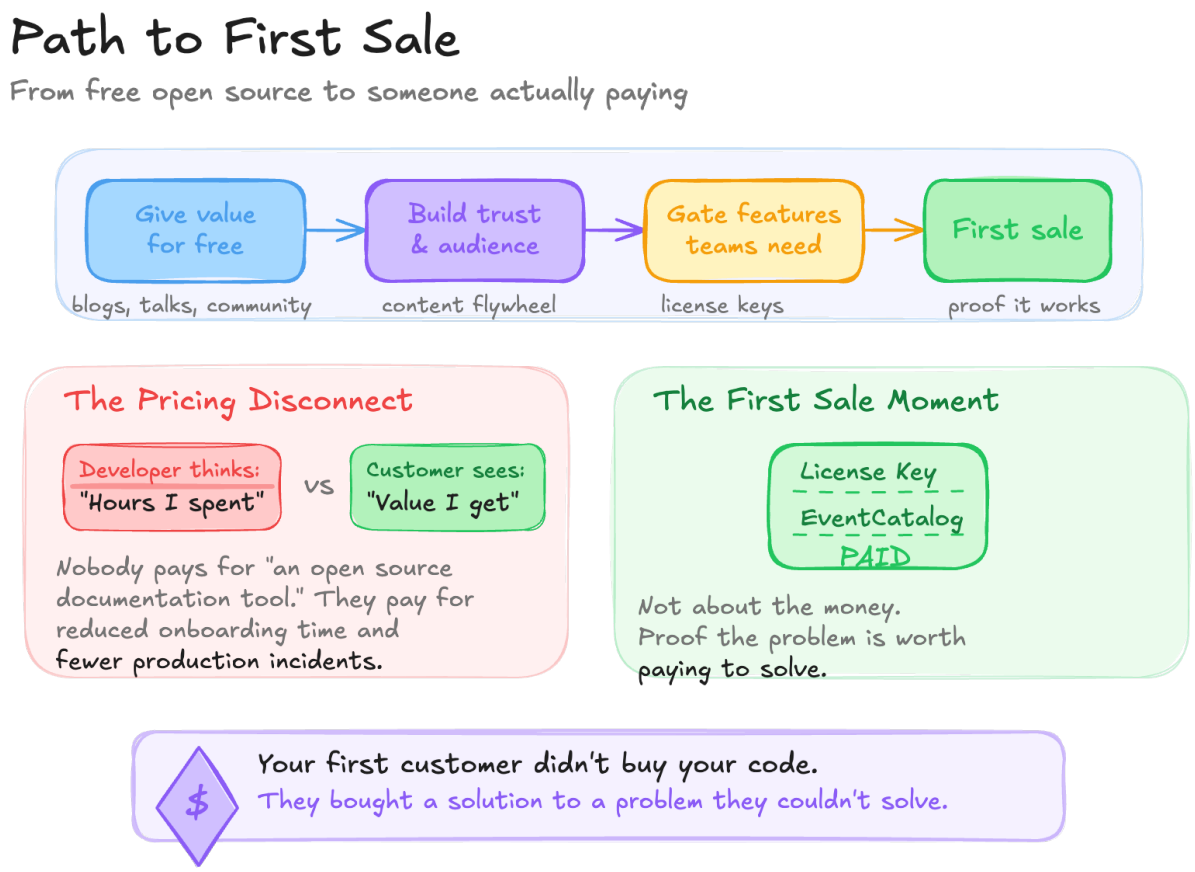

Figure 2: My First Sale and What I Learned About Turning Free Users Into Paying Customers

For months I had an open source project that people were using, but nobody was paying me for it. EventCatalog was free, it was growing, and I had no idea how to turn any of that into revenue. I'm a developer. I build things. I'd never sold anything in my life.

Then about two months after going full-time, someone bought a license key. A real person, at a real company, paying real money for something I built. The feeling was a mix of excitement, relief, and genuine imposter syndrome. I remember thinking, "Someone is actually buying the stuff I'm creating?" It hit different because as engineers, we're so disconnected from the purchasing side of software. We build it. Someone else prices it and sells it.

That first sale changed how I saw everything. It wasn't about the money (it was a small amount). It was proof that the problem I was solving was worth paying for. Here's what I learned getting there, and what I'd tell anyone who's sitting where I was.

Build Trust Before You Build a Checkout Page

- **Give value away before you ever ask for money.** Write blog posts, do talks, help people in the community. Every piece of free content is a deposit into a trust account you'll withdraw from later.
- Think about how you feel when someone tries to sell you a developer tool. You hate it. Your customers feel the same way. **Earn attention through usefulness, not promotion.**
- **Focus on the problem, not your project.** I talk about complexity in event-driven architectures because I genuinely believe it's a problem worth solving. EventCatalog happens to help with that. Lead with the problem and people will find your solution.
- The person who bought my first license key already knew me from my content and talks. That's not a coincidence. **Your content flywheel is your sales engine.** It just works on a longer timeline than you'd like.
- Something to consider: if nobody in your community knows what you're building or why, a checkout page won't fix that. Build the audience first, even if it feels slow.

What to Actually Sell (and When)

- **Start with the simplest possible paid thing.** For me it was license keys that unlock specific features. No complex billing system, no enterprise sales team. Just a key that gates a feature.
- Stop waiting for the perfect pricing page. **Your first version of "paid" can be ugly.** It just needs to exist. You can refine it after someone actually buys.
- Think about which features your power users depend on most. Those are your candidates for the paid tier. **Gate the features that solve pain for teams, not individuals.** Teams have budgets. Individuals don't.
- **Don't try to figure out pricing in a vacuum.** Talk to the people using your project. Ask what problems it solves for them. Ask what they'd pay to keep solving that problem. Their answers will surprise you.
- Something to consider: you don't need to monetize everything. Keep the core open source. Keep the community goodwill. Just find the one or two things that are worth paying for and start there.

The Moment Someone Pays You

- **Let the first sale validate the problem, not your ego.** It's tempting to celebrate the revenue. The real signal is that someone has a problem painful enough to pay for a solution. That's what you're building on.
- Expect a cocktail of feelings. Excitement, relief, and imposter syndrome all at once. That's normal. **Every founder I've talked to felt the same way about their first sale.**
- Think about what that first customer actually bought. They didn't buy your code. **They bought a solution to a problem they couldn't solve themselves.** Remember that when you're pricing, positioning, and building.
- **Reach out to your first customer and ask them why they bought.** Not in a survey. In a real conversation. What they tell you will shape everything that comes after.
- The first sale is tiny in dollar terms but enormous in what it unlocks. It proves the model works. It proves people will pay. **Everything after that is just doing it again.**

The Developer Pricing Disconnect

- **As developers, we're terrible at pricing our own work.** We think in terms of "hours spent building" instead of "value delivered to the customer." Unlearn that as fast as you can.
- Think about what companies pay for other tools in your space. Enterprise software pricing would shock most engineers. **Your tool is probably worth more than you think, especially if it saves a team time or reduces risk.**
- **Sell to the problem, not the project.** Nobody pays for "an open source documentation tool." They pay for "a way to reduce onboarding time and prevent production incidents." Same product, completely different value.
- Don't be afraid to charge. Seriously. **Free users who would never pay are not the audience for your paid tier.** The people who need what you've built will pay for it if you ask.
- Something to consider: pricing is its own skill and it's one most of us never learned. Read about how other open source founders price their products. I went with open-core (free core, paid features behind license keys) and it's been a good fit… but it took me a while to get comfortable with it.

How I Turned Content Into a Growth Engine and Why It Works Better Than Ads

> Why developer advocacy and problem-space thinking created a compounding growth flywheel for me

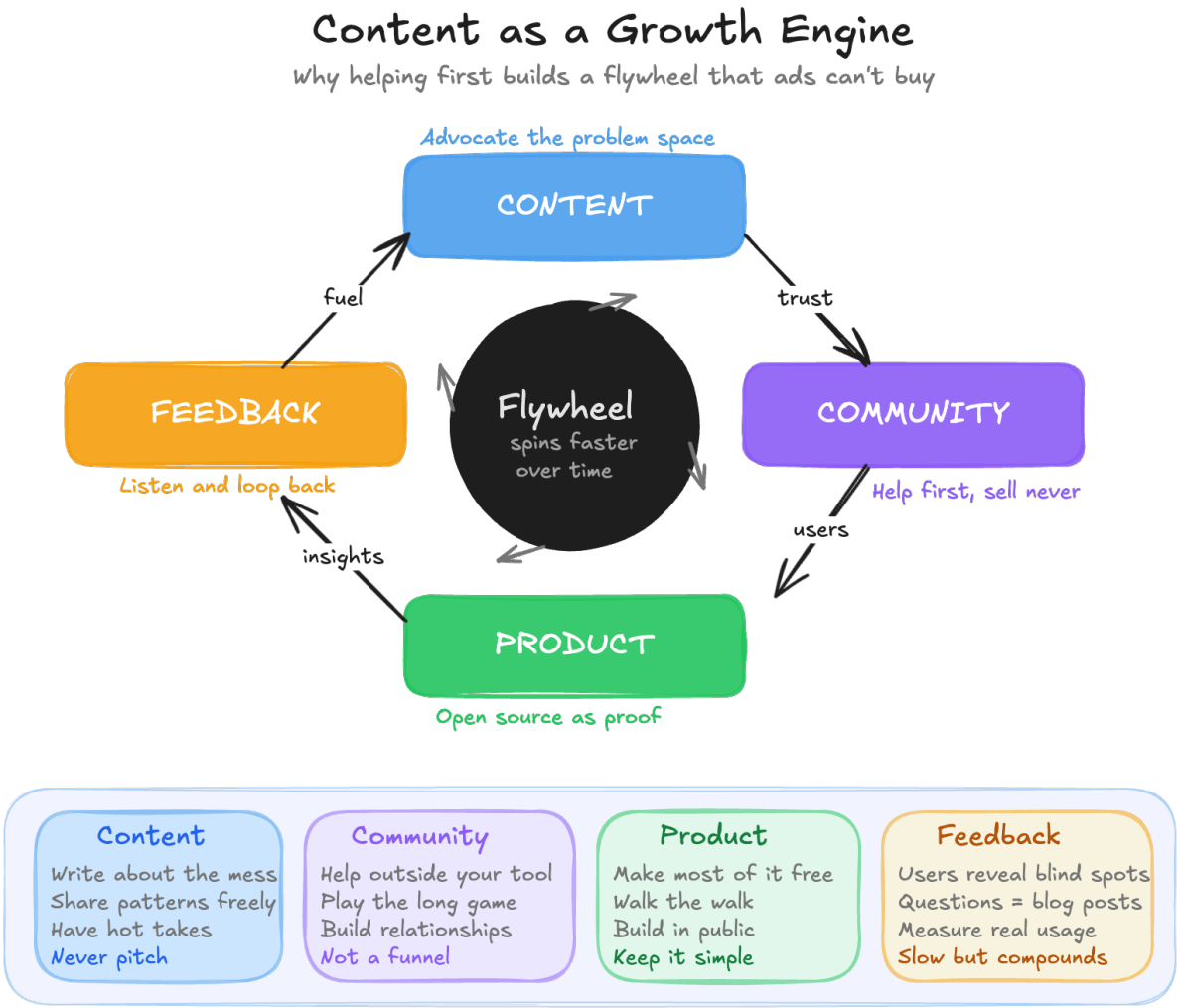

Figure 3: How I Turned Content Into a Growth Engine and Why It Works Better Than Ads

I spent years advocating for event-driven architectures before EventCatalog existed. I wrote about the mess, the complexity, the patterns that work and the ones that don't. I wasn't selling anything. I was just helping people navigate the same problems I'd faced.

When I started building EventCatalog, something clicked. People came to the project because they

already knew me from the problem space. They trusted me because I'd helped them before, not because I pitched them a product. I realized then that content isn't just marketing. It's part of the product itself.

Here's what I've learned about how this flywheel works and why it matters more than you think.

Content: Advocate the Problem Space

- **Write about the mess, not your solution.** Event-driven architectures are complex. Documentation is a pain. That's what I write about. Not "here's why EventCatalog is great" but "here's why this whole space is hard and what I've seen work."
- Think about what would be useful to someone who never uses your product. That's the content worth creating.
- **Share patterns and anti-patterns from the domain.** Teach concepts that apply beyond your tool. This builds trust because you're helping first, selling never.
- Engineers can smell a product pitch from a mile away. Be authentic about the problem space and they'll respect you for it.
- **Your hot takes matter.** Have an opinion. Say what you believe. "I think X is a mess" resonates more than "here are some considerations."

Community: Help First, Sell Never

- **Answer questions even when people aren't using your product.** I spend time on forums, Discord servers, helping people solve problems that have nothing to do with EventCatalog. That compounds.
- Think about the long game. Someone you help today might not use your tool for two years. But when they do need it, or when they change companies, they remember.
- **Engage authentically.** Don't drop product links. Don't redirect every conversation to your solution. Just be helpful. The ROI is invisible at first but it's real.
- Engineers are your ideal customer profile and they're very sensitive to sales tactics. Authenticity wins over conversion optimization every time.
- **Build relationships, not a funnel.** The people in your community become your feedback loop, your advocates, your next hires. This is worth more than any growth hack.

Product: Open Source as Proof

- **Make most of it free.** Open source is how you prove you're serious about helping, not just extracting value. People can see your code, your roadmap, your priorities.

- Think of paid features as sustainability, not the goal. The core value needs to be accessible or the flywheel doesn't spin.
- **Your product backs up what you teach.** If you write about documentation being hard, your tool better make documentation easier. Walk the walk.
- Building in public compounds trust. People see you iterating, taking feedback, making mistakes. That's powerful.
- **Don't over-index on features.** Most developers would rather use something simple that solves the core problem than a feature-bloated tool that tries to do everything.

Feedback: Listen and Loop Back

- **Users give you problem insights you'd never find alone.** Every conversation reveals gaps in your thinking, edge cases you didn't consider, pain points you forgot existed.
- Think about feedback as fuel for your next content. Someone asks a question? That's your next blog post. Multiple people hit the same wall? That's a talk.
- **This is where the flywheel compounds.** Content brings people in. People give you feedback. Feedback shapes your product and your content. Better content brings more people. It's slow but it works.
- Measure what matters. Not just downloads or stars. Are people actually using it? Are they coming back? Are they recommending it?
- **The cycle takes time.** I'm still figuring this out. But I know it works because I've seen it. Content from two years ago still brings people in. Conversations from last month shape what I'm building today.

I Confused Shipping Speed With Product Speed

> Shipping fast doesn't mean shipping smart. Here's what I learned about finding the right release cadence when your users have jobs and limited bandwidth.

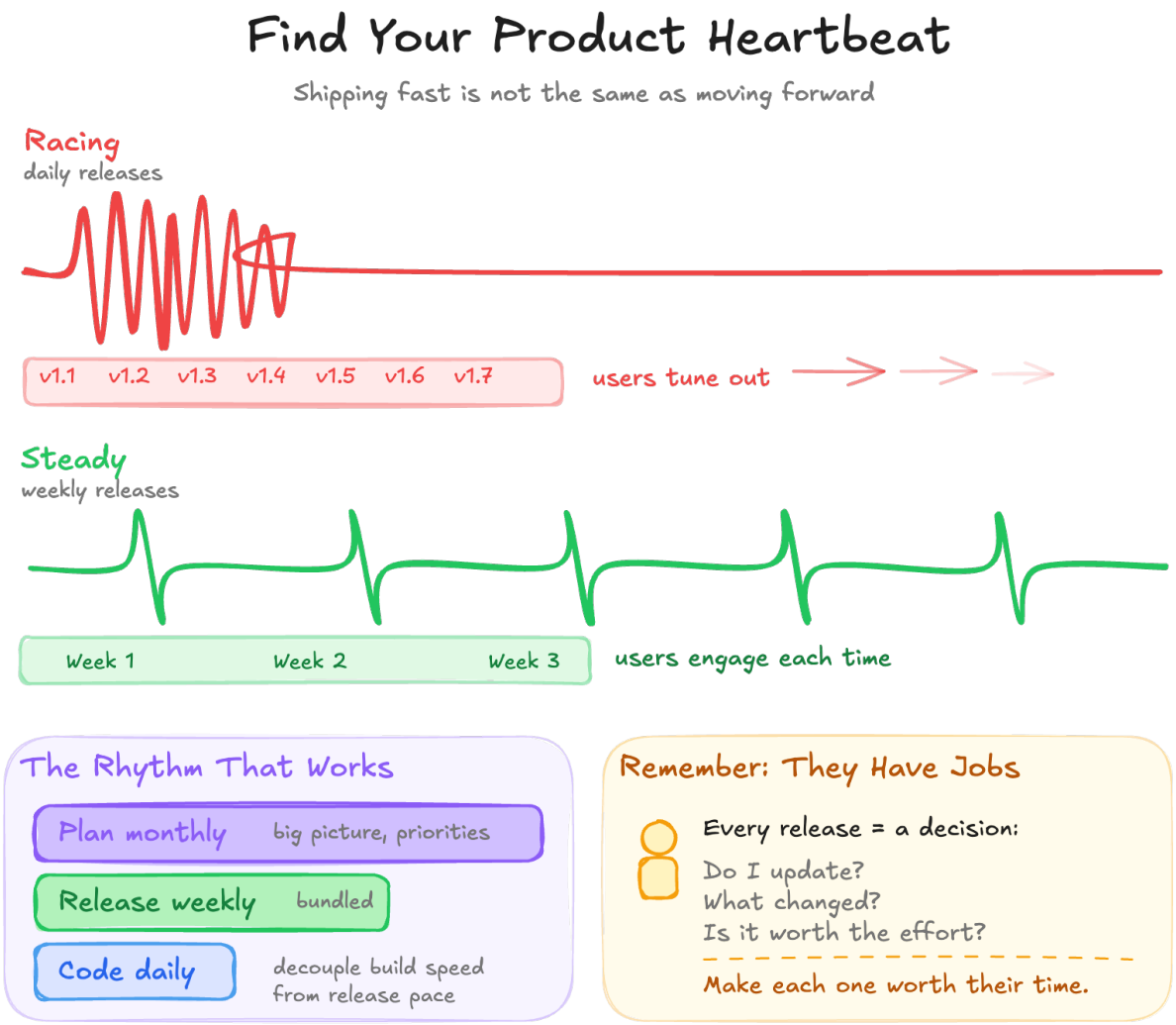

Figure 4: I Confused Shipping Speed With Product Speed

For a while, I was shipping features almost every day. Sometimes multiple times a day. With AI tooling making it easier to move fast, it felt productive. I'd wake up with an idea, have it in production by afternoon, and announce it to the community. Rinse and repeat.

But something started to shift. Social engagement was declining. People were still using EventCatalog, but the excitement around releases felt… muted. Then a few users started giving me feedback,

gently at first. "The pace is a bit much." "Hard to keep up with all the changes." It hit me: I was creating noise, not momentum.

I had to relearn something fundamental. My users have jobs. They have their own projects. EventCatalog is self-hosted, which means every release is a decision they have to make. Do I update? What changed? Is it worth the effort? I was moving at my pace as a builder, not their pace as users trying to get work done. Here's what I've learned about finding the right heartbeat for a product.

The Trap of Shipping Too Fast

- **Just because you can build fast doesn't mean you should release fast.** AI tooling and modern dev workflows make it easy to ship features daily. That speed is a tool, not a strategy.
- Think about what "momentum" actually means. It's not how many features you ship. It's whether people notice, care, and adopt what you're building. **Constant releases create noise that drowns out the signal.**
- **Engineering speed and product speed are different things.** You can code every day and still release weekly. Use the time between releases to bundle features, write better docs, and craft updates that are easier to digest.
- Something to consider: when everything is urgent and new, nothing feels important. **Your users tune out when the release cadence becomes background noise.**
- Developers love to build. It's what we do. But product work is just as important as coding. Planning, bundling, positioning, communicating… that's the stuff that turns features into value.

The Signal You're Overdoing It

- **Watch your engagement metrics.** If people used to react to your releases and now they don't, that's not a content problem. That's a frequency problem.
- Listen when users tell you the pace is too much. They're being polite. What they mean is: "I can't keep up and it's starting to feel like work to follow your project."
- **Self-hosted products have a different rhythm than SaaS.** Every release asks your users to take action. Update the dependency. Test it. Deploy it. Respect that friction.
- Think about how you feel when a tool you use releases updates constantly. It's exhausting. Your users feel the same way about your product.
- Something to consider: if you're releasing daily and getting less engagement than when you released weekly, the market is telling you something. **Slow down and let each release land.**

Finding Your Heartbeat

- **The rhythm that's working for me: plan monthly, release weekly, code daily.** Plan the bigger picture once a month. Reconcile priorities every week. Ship one meaningful update per week. Code and fix issues as they come up.
- Think about bundling features instead of shipping them one at a time. Three small features in one release feels like progress. Three releases in three days feels like chaos.
- **Weekly releases create a predictable rhythm.** Your users know when to expect updates. You have time to write good changelogs and explain what changed. Everyone wins.
- Something to consider: consistency beats intensity every time. A heartbeat is steady. It's reliable. **It doesn't race, and it doesn't flatline.**
- Don't confuse building with shipping. You can work on five things at once. Just don't release all five at once. **Use your velocity to build a backlog of polished features, then release them at a sustainable pace.**
- The "plan weekly, release weekly" idea resonates because it decouples your internal pace from your external communication. **Ship on a schedule that respects your users' bandwidth, not your build speed.**

What Your Users Actually Need

- **High-quality releases beat high-frequency releases.** Spend the extra time writing better docs, better changelogs, better examples. Your users will engage more when each release feels complete.
- Think about the cognitive load you're putting on your audience. Every release is a micro-decision for them. Do I update? Do I read the changelog? Do I test it? **Make those decisions worth their time.**
- **Your users have jobs, side projects, and limited attention.** Especially in the developer tools space, your audience is busy. They want to see progress, but they can't track constant change.
- Something to consider: self-hosting means your users are responsible for keeping your product running. **The more you release, the more work you're creating for them.** Be thoughtful about that trade-off.
- **Developers want simple solutions to hard problems.** Spend your time making the product simpler, not just adding features. That's where the real value is, and it takes longer than you think.
- Remember: you're building for the long term. **A sustainable pace now is better than burning out your users (or yourself) with unsustainable intensity.**

I Got to $23K MRR Without Stripe and What That Taught Me About Starting Simple

> You don't need half the infrastructure you think you need. Here's what actually matters when you're building something from zero.

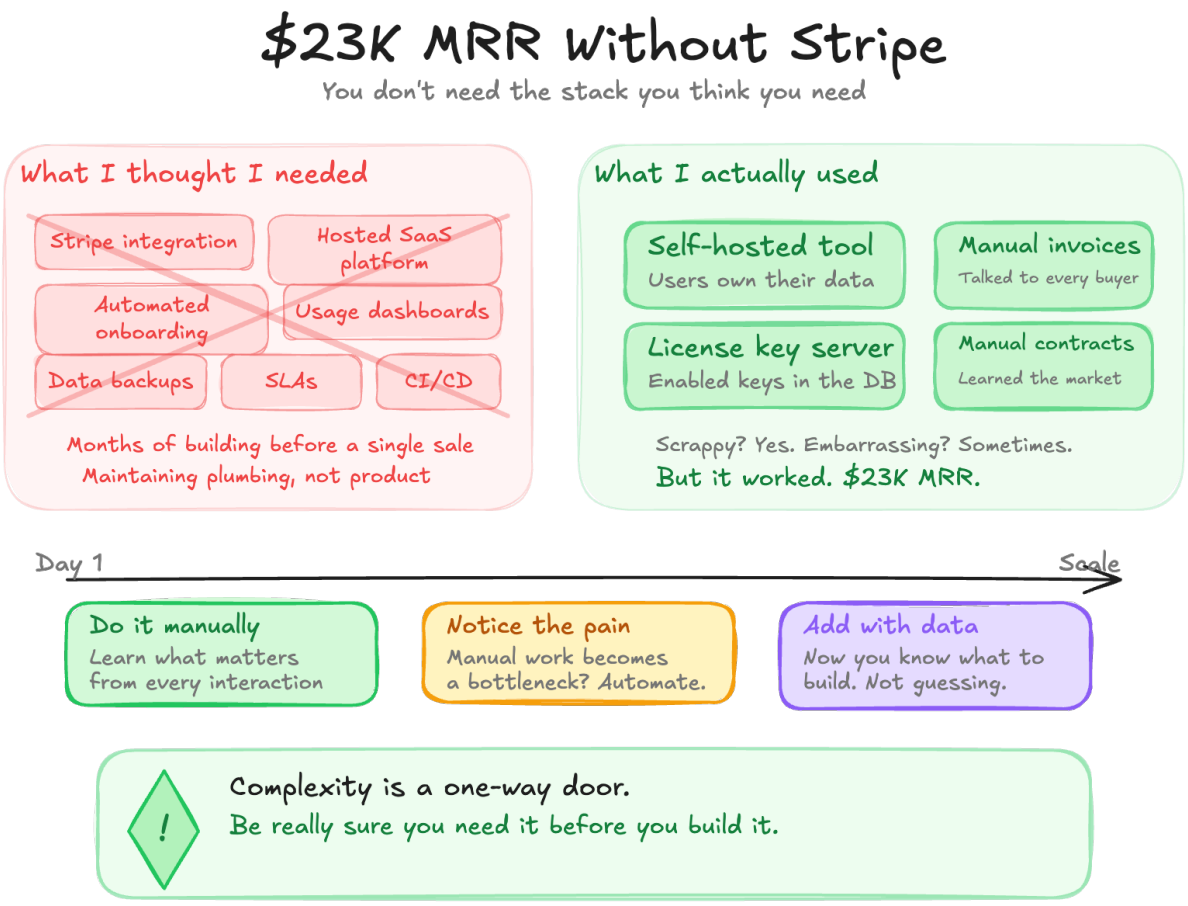

Figure 5: I Got to $23K MRR Without Stripe and What That Taught Me About Starting Simple

When I started building EventCatalog, I had this picture in my head of what a "real" product looks like. Stripe integration. Hosted SaaS platform. Automated onboarding. Dashboards. The works. And I almost went down that road. But something stopped me. I looked at what I actually needed to prove and realized… none of that stuff mattered yet.

So I started simple. Really simple. A self-hosted tool with a license key server. No Stripe. No hosting. No SLAs. No data backups. I handled contracts manually. I sent invoices by hand. I enabled license keys directly in the database. It felt scrappy, and honestly, it felt a bit embarrassing at times. But it

worked. I got to $23K MRR doing things that way.

The lesson I keep coming back to is this: most of us are building for a version of our product that doesn't exist yet, for users we don't have yet, solving problems we haven't confirmed are real. Here's what I've learned about keeping things simple and why it might be the most important thing you do early on.

You Don't Need the Stack You Think You Need

- **Start by asking what you actually need to prove.** Is it that people will pay? That the product solves a real problem? You can validate both without Stripe, without a SaaS platform, without any of the "standard" infrastructure.
- Think about what your product looks like if you strip away everything except the core value. That's your starting point. **Everything else is a distraction until you've proven the core works.**
- **I ran without Stripe for the first $23K of MRR.** Manual invoices. Manual license keys. It was slow, but it forced me to talk to every single customer. That was worth more than any payment integration.
- Something to consider: every piece of infrastructure you add is something you have to maintain. As a solo founder, your time is the scarcest resource you have. **Spend it on the product, not the plumbing.**
- **Self-hosting can be a superpower.** No servers to manage. No uptime guarantees. No data liability. Your users own their data, and you get to focus purely on making the tool better. There's a huge market for self-hosted software, especially if you're bootstrapping alone.

Do Things That Don't Scale

- **Handle things manually until it hurts.** Contracts, invoicing, onboarding, license key provisioning. Do all of it by hand at first. You'll learn things about your customers that no dashboard will ever tell you.
- Think about what you'd automate on day one if you were following the "standard" playbook. Now ask yourself: do I actually need that right now, or am I just building it because it feels professional?
- **Every manual interaction is a chance to learn.** When I was enabling license keys in the database myself, I knew exactly who my customers were, what they needed, and why they were paying. That's invaluable early on.
- Something to consider: the things that don't scale are often the things that build the deepest understanding of your market. **Automation is great, but not before you understand what**

you're automating.

- **You can always add infrastructure later.** I just added Stripe recently. The business didn't suffer without it. If anything, the delay helped me understand my pricing and customers better before locking into a system.

Simple Is Enough (Until It Isn't)

- **Stop waiting for your product to feel "ready."** It won't. Ship the simplest version that solves a real problem for real people. You can iterate from there.
- Think about the difference between "simple" and "incomplete." A product with one well-solved use case is simple. A product with ten half-built features is incomplete. **Aim for simple.**
- **Your first version doesn't need to look like anyone else's product.** It doesn't need a flashy homepage. It doesn't need a polished onboarding flow. It needs to work, and it needs to solve a problem someone is willing to pay for.
- Something to consider: most people building products are learning on the fly. That's normal. You don't need to have it all figured out. **You just need to start, and starting simple is the fastest way to learn what actually matters.**
- **The bar for "good enough" is lower than you think.** Especially in developer tools, people care about whether your product solves their problem. They'll forgive rough edges if the core value is there.

How to Know When to Add Complexity

- **Add infrastructure when the manual version becomes a bottleneck, not before.** If you're spending more time on invoicing than building features, that's when you automate invoicing. Not a day sooner.
- Think about what your customers are actually asking for. Are they asking for Stripe? Or are they asking for the product to do more? **Build what they're asking for, not what you think a product "should" have.**
- **Watch for the signals.** When manual processes start eating into your product development time, when customers start expecting faster turnaround, when you're dropping balls because there's too much to juggle. Those are the signs.
- Something to consider: complexity is a one-way door. Once you add a hosted option, a SaaS tier, or an automated billing system, you're maintaining it forever. **Be really sure you need it before you build it.**
- **The best time to add complexity is when you have data, not opinions.** After months of manual work, you'll know exactly what to automate, what to skip, and what your customers actually

value. That’s a much better position to build from than guessing on day one.

How AI Gave Me the Capacity of a Small Team and Why That Changes Everything for Solo Founders

> The bottleneck used to be building. Now it's deciding what to build. Here's how AI shifted what's possible when you're bootstrapping alone.

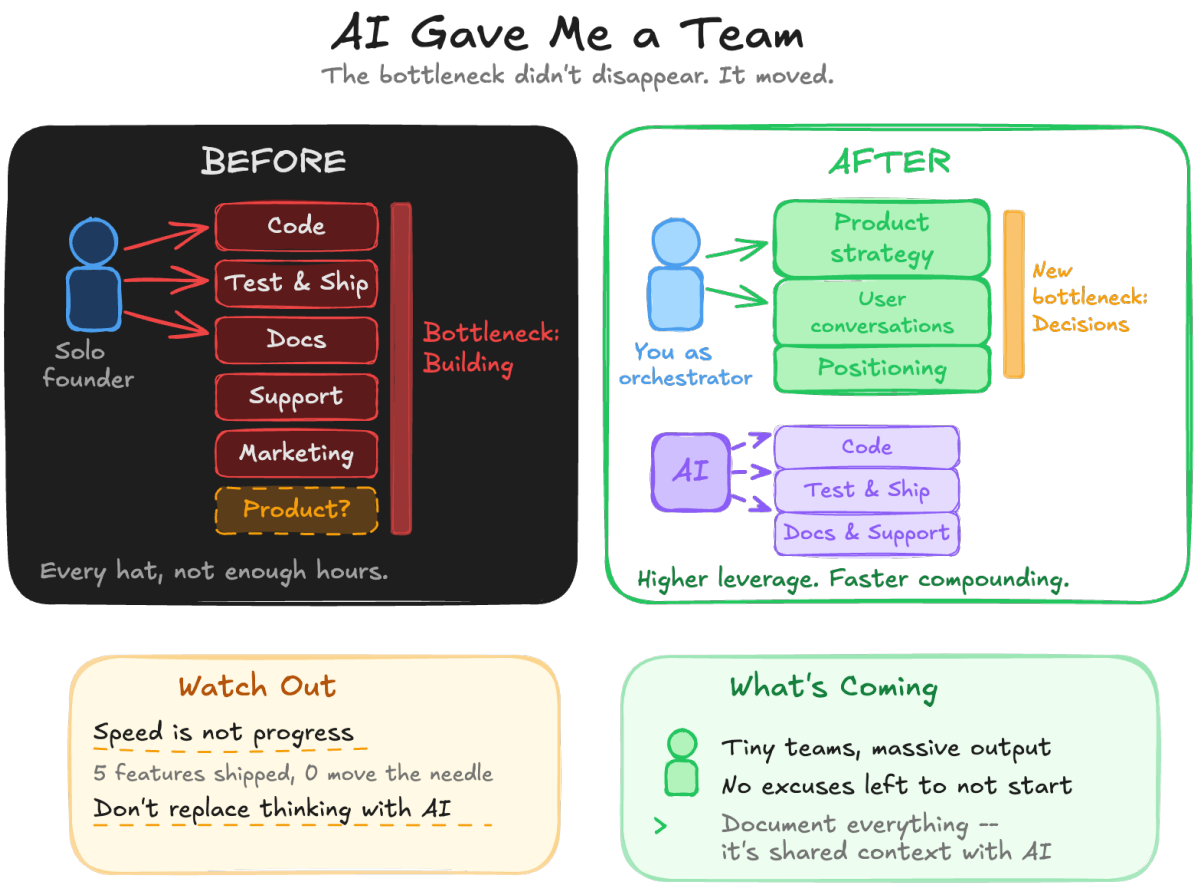

Figure 6: How AI Gave Me the Capacity of a Small Team and Why That Changes Everything for Solo Founders

I've been building EventCatalog on and off for four years. For most of that time, I was the bottleneck. PR reviews piled up. Code took days to ship. Invoices, contracts, GitHub issues, releases… it all sat on my plate, and there was only so much I could get through in a day. That's just the reality of being a solo founder. You wear every hat, and some of them don't fit.

But over the past 14 months of going full-time, something shifted. AI tooling got genuinely good. Not "helpful autocomplete" good. More like "this thing just reviewed my pull request, brainstormed a fea-

ture with me, and helped me ship it in a few hours" good. The bottleneck I'd been living with for years started to dissolve. Not completely, but enough that it changed how I think about what's possible alone.

The thing is, the bottleneck didn't disappear. It moved. And that's worth understanding if you're building something solo. Here's what I've learned about working with AI as a bootstrapper, what it unlocks, and the traps that come with it.

The Bottleneck Moved

- **For years, the constraint was building.** I had more ideas than I could ship. Features sat in my head for weeks because there weren't enough hours to code, test, document, and release them. That's not the constraint anymore.
- Think about where your time actually goes. If you're still spending most of it writing code, AI can probably take a big chunk of that off your plate. **The question shifts from "can I build this?" to "should I build this?"**
- **The new bottleneck is product thinking.** What to build, what to prioritize, how to position it, how to communicate it. These are the decisions that matter most now, and they're harder to outsource to an AI.
- Something to consider: if AI removes the building constraint, what's left is the stuff that requires your judgement, your taste, your understanding of your users. **That's where your value is as a founder.** Lean into it.
- **This is actually good news.** Product decisions are higher-leverage than code. Every hour you spend thinking about what to build (instead of how to build it) compounds faster.

What I Actually Do Now

- **Most of my coding time is spent orchestrating agents, not writing code myself.** I describe what I want, review what comes back, steer the direction. What used to take me days now takes hours. It's a completely different workflow.
- **I use AI across the whole business, not just code.** SEO audits, copywriting, product strategy, brainstorming. I've built custom skills and agents that understand my project's vision and mission. They reference my docs. They stay aligned with where I'm headed.
- Think about documenting your vision and keeping it in your codebase. **When your AI tools can reference your mission, your brand voice, your product direction, the output quality goes way up.** Context is everything.
- **The time I save on building goes straight into product and business work.** Roadmaps, user conversations, positioning, pricing. The stuff I used to squeeze into the gaps between coding

sessions now gets real attention.
- Something to consider: the skill of being a solo founder is shifting. It's less about being a great coder and more about being a great orchestrator. **Learn to direct AI well and you'll move faster than most small teams.**

The Traps That Come With More Capacity

- **More capacity means you can ship too much, too fast.** I learned this the hard way. If your users can't keep up with your changes, you're creating noise, not value. AI makes it easy to build. That doesn't mean you should release everything you build.
- **Watch out for losing your ability to think critically.** AI feels like a cheat code sometimes. But if you skip the hard thinking, you skip the learning. You'll make worse product decisions because you never developed the muscle.
- Think about how much trust you're putting in AI output. **We naturally give it too much credit.** It sounds confident even when it's wrong. Review everything. Question the suggestions. The moment you stop thinking for yourself is the moment your product starts drifting.
- **Don't confuse speed with progress.** You can ship five features in a week with AI help. But if none of them move the needle, you just created five things to maintain. **Velocity without direction is just busy work.**
- Something to consider: the founders who will do best with AI are the ones who use it to free up thinking time, not replace thinking entirely. **Stay in the loop. Stay curious. Stay critical.**

Smaller Teams, Bigger Output

- **We're going to see a wave of tiny companies doing things that used to require 20-person teams.** One or two people with the right AI tooling can build, ship, and support products at a scale that wasn't possible two years ago. This is just getting started.
- Think about what this means for you personally. **If you've been waiting to start something because you thought you needed a team, that barrier is lower than it's ever been.** The excuses are running out.
- **Self-hosted, bootstrapped products are perfectly positioned for this.** No infrastructure to manage. No SLAs. No ops team. Just a product people install and run themselves. AI handles the building, you handle the product and the customers.
- **Document everything about your project.** Your vision, your tone, your architecture, your decisions. The better your documentation, the better AI can help you. **Your codebase becomes the shared context between you and your AI collaborators.**

- Something to consider: the playing field is levelling out. Big companies have more people, but solo founders have more speed and more focus. AI amplifies both. **If you can move fast and stay focused, you can compete with anyone.**

Why I Spent All My Time Chasing New Customers and What I Learned When Renewals Hit

> What happens when you focus all your energy on acquisition and forget the people already paying you.

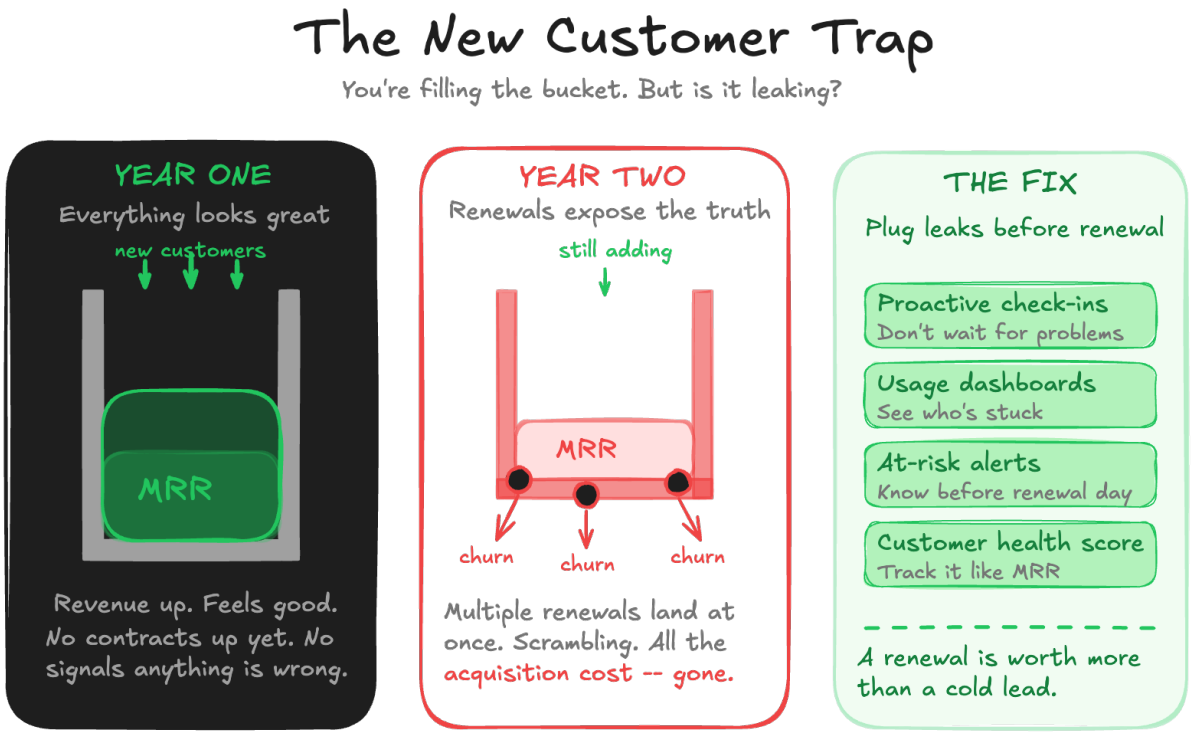

Figure 7: Why I Spent All My Time Chasing New Customers and What I Learned When Renewals Hit

The new customer trap

I'll be honest, getting a new paying customer is one of the best feelings in bootstrapping. Someone sees what you've built, pulls out their card, and says "yes, this is worth paying for." That dopamine hit is real. And when you're early on, you need that revenue. You need the runway. So you chase the next one. And the next one. It makes total sense.

The problem is, I did this for a long time. Every week was about who's in the funnel, who's close to signing, how do I get the next logo. And it worked… for a while. Revenue was going up. Things felt good.

Then year two arrived. Renewals started coming up. And suddenly I realized I had spent almost no

time making sure the people already paying me were actually getting value from EventCatalog. I'd been so focused on filling the bucket that I forgot to check if it was leaking.

The Dopamine Hit of New Customers

- **Recognize that acquisition is addictive.** Every new customer feels like validation. That's great, but it can pull all your attention away from the people who already believe in you.
- Think about where your time actually goes each week. If you're spending most of it on prospects and almost none on existing customers, that's a pattern worth noticing early.
- **New customer revenue is exciting, but renewal revenue is what keeps you alive.** One builds momentum. The other builds a business.
- Something to consider… the skills that win new customers (marketing, demos, sales) are completely different from the skills that keep them (support, onboarding, making sure they get ROI). You have to invest in both.

What Happens When Year Two Arrives

- **Renewals will expose every shortcut you took.** If customers aren't getting real value from your product, you'll find out when it's time to renew. Not before.
- The painful thing is retention problems are invisible early on. Everything looks fine because nobody's contract is up yet. Then suddenly, multiple renewals land at once and you're scrambling.
- **Ask yourself right now: do your current customers know how to get the most out of your product?** If you're not sure, that's your answer.
- Think about this… a customer who churns after year one isn't just lost revenue. It's all the time you spent acquiring them, onboarding them, and supporting them. Gone.

What I'm Doing Differently Now

- **I started reaching out to customers proactively.** Not waiting for them to come to me with problems. Regular check-ins, even quick ones, go a long way.
- **I built usage tracking and custom dashboards** so I can actually see if customers are using EventCatalog, how often, and where they might be stuck. You can't improve what you can't see.
- **I set up automation to flag at-risk accounts.** If usage drops, I want to know before the renewal conversation, not during it. Early signals give you time to act.
- Something worth exploring… make "customer health" as visible as your MRR dashboard. If you track new revenue obsessively but don't track engagement, you're flying blind on retention.

How to Avoid This Trap From Day One

- **Your first 5-10 customers are your most important asset.** Resist the urge to immediately chase number 11. Make sure 1 through 10 are thriving first.
- **Get on calls with your existing customers regularly.** Ask for feedback. Move fast on it. The power of being a solo founder or small team is that you can ship what they need this week, not next quarter. That surprises people in a good way.
- Think about building retention into your workflow from the start. Don't wait until renewals force you to figure it out. A simple monthly check-in habit costs you nothing and catches problems early.
- **Growth isn't just new logos.** A customer who renews, expands, or refers someone is worth more than a cold lead. Some of the best "acquisition" you can do is making sure your existing customers love what you've built.

Why I Undercharged for EventCatalog and What I Learned About How Developers Get Pricing Wrong

> Why developers are wired to undersell their products and how to start pricing based on the problem you solve, not the code you wrote.

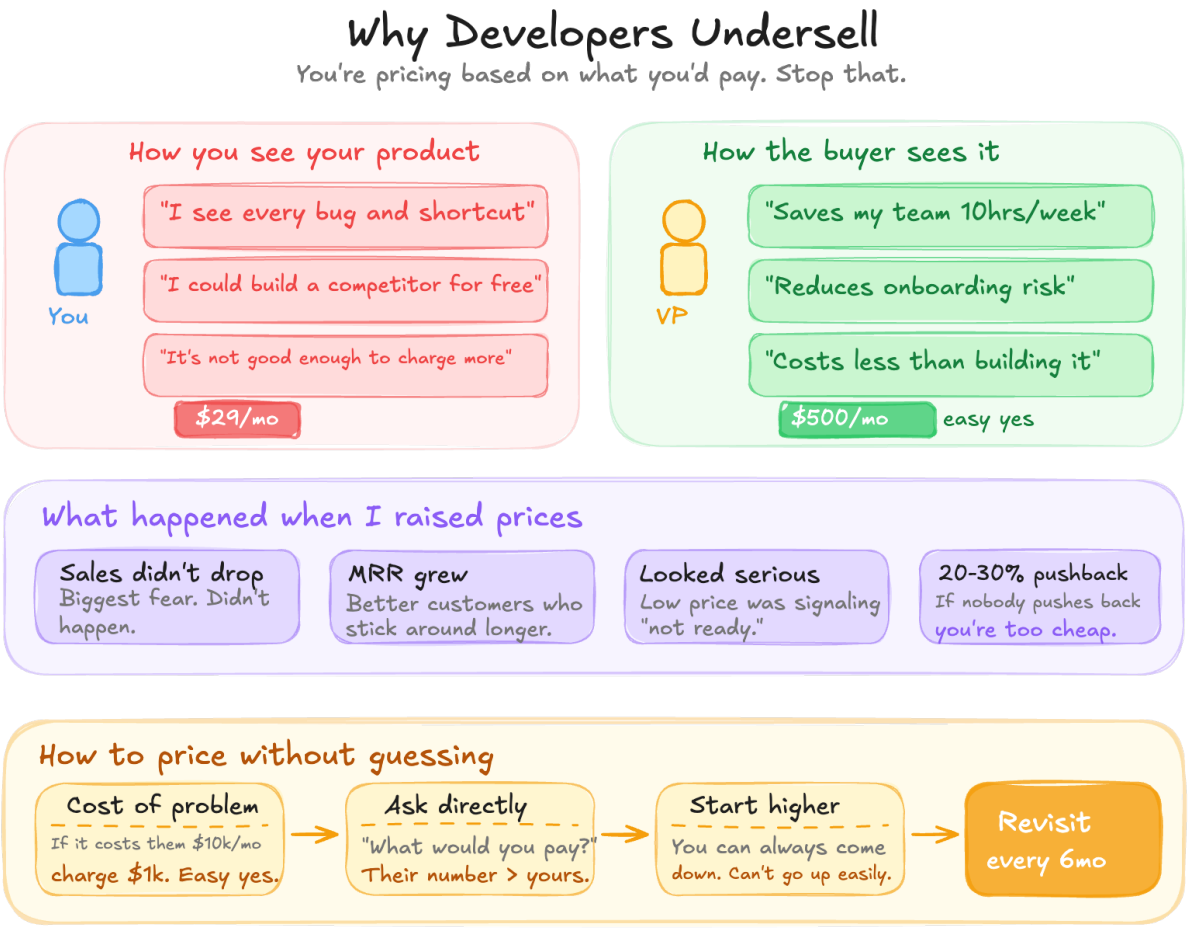

Figure 8: Why I Undercharged for EventCatalog and What I Learned About How Developers Get Pricing Wrong

Developers and pricing

Someone told me something that stuck with me. "Developers are terrible at pricing." At first I pushed back on it. I'd spent years thinking about what things cost, comparing tools, hunting for deals. But that's exactly the problem. We're price sensitive. We compare everything to free. We know we could

probably build it ourselves. And when it comes time to price our own products, all of that baggage comes with us.

Here's the thing I didn't fully understand until I raised my prices on EventCatalog. Developers are rarely the ones buying software in a company. The budget holder is a VP, a team lead, a director. They're solving a business problem and they have a budget for it. Software sells for thousands in enterprises. But as developers, we're disconnected from that entire flow. So we price based on what we'd pay, not what the product is worth.

I raised my prices, and nothing bad happened. MRR grew. Nobody left. If anything, the product started looking more serious. That was a wake-up call. Here's what I've learned about this trap and how to think about it differently.

Why Developers Are Terrible at Pricing

- **We price based on what we'd personally pay.** That's the wrong reference point. You're not your customer. The person buying your product is solving a business problem with a real budget, not a developer comparing it to a free GitHub repo.
- **We anchor against open source and free tools.** If you've spent your career using free software, it's hard to charge $500/month for yours. But your buyer isn't comparing you to a free tool. They're comparing you to the cost of building it themselves or the cost of not solving the problem.
- **Imposter syndrome sneaks into pricing.** We see every bug, every shortcut, every feature we haven't built yet. So we think "it's not good enough to charge that much." Meanwhile, the customer sees a product that solves their problem and is happy to pay.
- Think about this… if your product feels too cheap, it might actually look cheap. There's real psychology behind pricing. A low price can signal "this isn't serious" to a buyer with a real budget.

You're Not Your Customer

- **Figure out who is actually paying.** In most companies, the person using your tool and the person approving the purchase are different people. The buyer cares about the business outcome, not the tech stack.
- **Ask your customers directly what they'd be willing to pay.** This sounds scary but it's one of the most valuable conversations you can have. Most of us guess at pricing instead of just asking. You might be surprised how much higher their number is than yours.
- **Find out what the problem is costing them today.** If your product saves a team 10 hours a week, that's thousands per month in developer time. Your $200/month price looks like a bargain against that… not an expense.

- Something to consider… when you downplay your product on calls or apologize for missing features, you're training the buyer to think it's worth less. Talk about the problem you solve, not the code you wrote.

What Happened When I Raised My Prices

- **Sales didn't drop.** This was my biggest fear and it just didn't happen. The people who see value in your product will pay what it's worth. The ones who won't were probably going to churn anyway.
- **MRR actually grew.** Fewer tire-kickers, more serious buyers. Higher prices can attract better customers who stick around longer and get more value from the product.
- **I read that you want about 20-30% of prospects pushing back on price.** If nobody is pushing back, you're probably too cheap. That pushback is actually a healthy signal that you're in the right range.
- **Stop giving too much away for free.** It's tempting to be generous with free tiers and discounts because charging feels uncomfortable. But every discount trains your market to expect lower prices. Be thoughtful about what's free and what's paid.

How to Price Without Guessing

- **Start by understanding the cost of the problem.** How much is this costing your customer in time, money, or risk right now? Price your product as a fraction of that cost. If you save them $10,000/month, charging $1,000 is an easy yes for them.
- **Have honest pricing conversations early.** Ask prospects "what would you expect to pay for this?" and "what's this problem costing you today?" These two questions will teach you more about pricing than any blog post.
- **Start higher than feels comfortable, then watch the signals.** You can always come down, but raising prices on existing customers is much harder. Give yourself room to learn.
- Think about revisiting your pricing every 6 months. Your product gets better over time. Your understanding of the market deepens. Your prices should reflect that growth… not stay frozen at whatever you guessed on day one.

How AI Made Scope Creep My Biggest Threat and Why Speed Is Not a Strategy

> AI makes building faster than ever. That's exactly why it's more important than ever to build less.

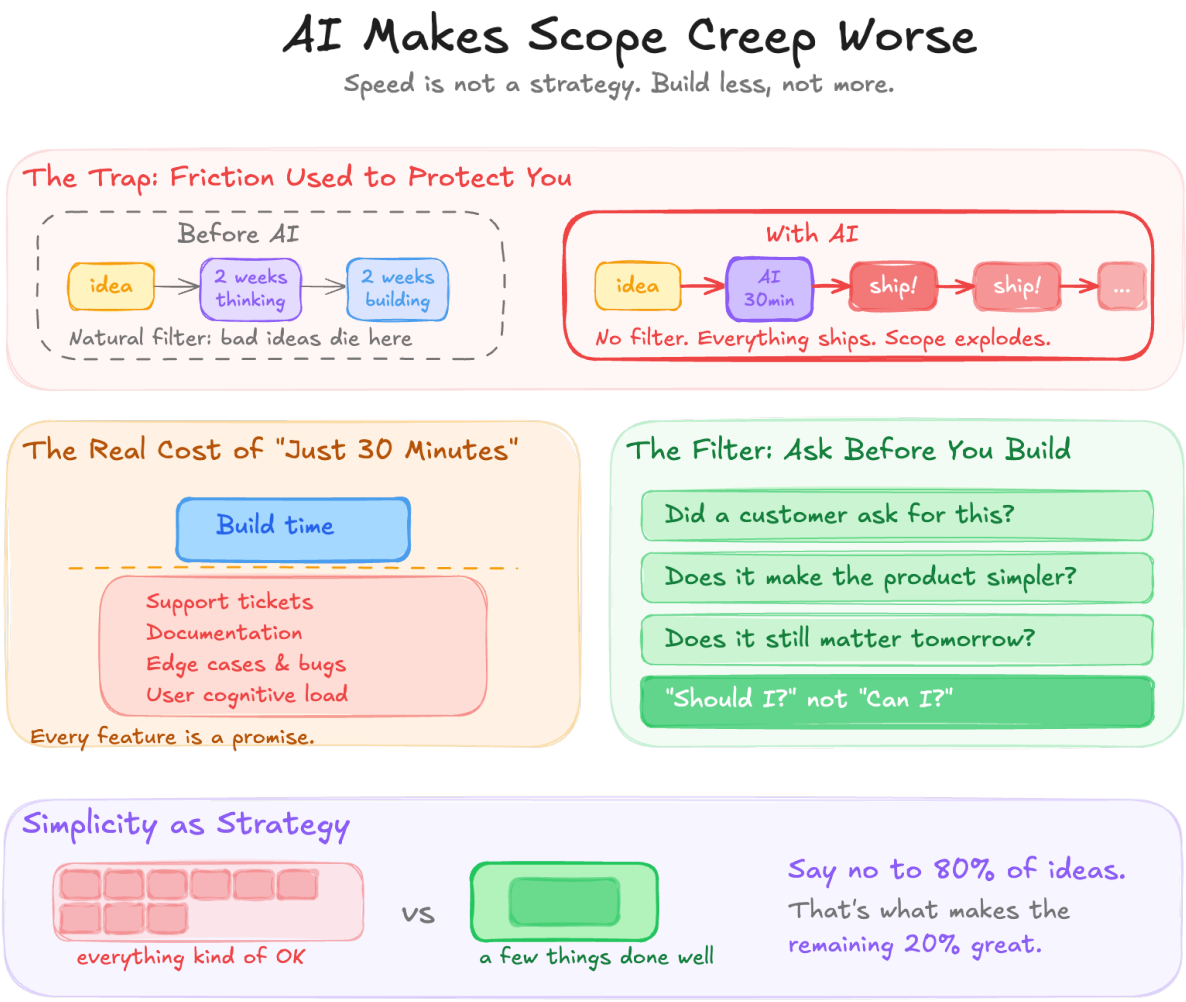

Figure 9: How AI Made Scope Creep My Biggest Threat and Why Speed Is Not a Strategy

Here's something I didn't expect. The better AI tools get, the harder it is to stay disciplined about what I build. Fourteen months ago, I had a natural filter. If a feature was going to take two weeks to build, I had to really want it. I had to justify the time. Now? I can describe an idea to Claude Code faster than I can sketch a mockup. And that changes everything about how scope creep sneaks in.

I've fallen into this trap more times than I'd like to admit. "Just add this one thing." "It'll only take an hour." The problem isn't that the feature is bad. The problem is that every feature you ship is a feature

your users have to learn, and a feature you have to maintain. I've watched users struggle to keep up with changes because I was shipping faster than they could absorb.

The truth is, your users are craving simplicity. They want a product that feels good, that's easy to get started with, that doesn't overwhelm them. And the only way to give them that is to build less, not more. Here's what I've learned about keeping scope creep in check when AI makes everything feel possible.

The Trap

- **It's faster to describe an idea than to evaluate it.** That's the core problem. The friction that used to protect you from bad ideas is gone. You used to have time to think while you built. Now the building happens before the thinking.
- **"It only took 30 minutes" is the most dangerous sentence in your vocabulary.** The cost of a feature is never the time it took to build. It's the support tickets, the documentation, the edge cases, the cognitive load on your users.
- **AI doesn't have product sense.** It'll build whatever you ask for, brilliantly. But it can't tell you whether you should be asking in the first place. That's still your job, and it's more important than ever.
- **Experimentation feels productive but can be a distraction.** Building cool things is fun. I get it. But fun and valuable aren't the same thing. Something to consider next time you're about to ship something "just because you can."

The Real Cost

- **Users can't absorb features as fast as you can ship them.** I've seen this firsthand. You push three updates in a week and people start feeling lost. The product they liked yesterday feels different today.
- **Every feature is a promise.** Once it's out there, someone depends on it. Now you're maintaining it, fixing bugs in it, making sure it works with the next thing you build. That compounds fast.
- **Think about what your product feels like to someone using it for the first time.** If they're overwhelmed, it doesn't matter how powerful it is. They'll leave before they get to the good stuff.
- **Feature bloat is the slow death of simple products.** The products people love most are the ones that do a few things really well. Not the ones that do everything kind of OK.

The Filter

- **Before you build, ask: did a customer ask for this?** If the answer is no, that's a red flag. You might be building for yourself, not for them.
- **Keep UX and DX at the front of your mind.** How does the product feel? Is it easy to get up and running? Are there barriers to entry? What's the onboarding like? These questions won't go away no matter how good AI gets.
- **Sleep on it.** Seriously. If the idea still feels important tomorrow, maybe it's worth building. If you've already forgotten about it, you just saved yourself a maintenance headache.
- **Think about whether this makes the product simpler or more complex.** The best features remove friction. The worst ones add options. There's a difference.
- **"Can I build this?" is the wrong question. "Should I build this?" is the right one.** AI shifted the bottleneck from capability to judgment. Your judgment is now the most valuable thing you bring to your product.

Simplicity as Strategy

- **Simple products win.** Not because they're easy to build, but because they're easy to use. That's what people pay for. That's what they recommend to others.
- **Your role is changing.** AI can be the builder now. Your job is to be the product person. To decide what not to build. To protect the experience. That's harder than writing code, and it matters more.
- **Constraints are a feature, not a bug.** Saying no to 80% of ideas is what makes the remaining 20% great. Don't let AI's speed erode your ability to say no.
- **Think about the products you love most.** Chances are they feel focused. They do one thing or a few things really well. That's not an accident. Someone said no to a thousand other ideas to get there.
- **Simplicity is a competitive advantage that compounds.** While others are drowning in features, a focused product becomes easier to use, easier to support, and easier to grow. That's a position worth protecting.