

Building an app is cheap now... but owning it might not be

What to weigh before you build it yourself

What's inside

00	Introduction	p. 03
01	A lot has changed in software development	p. 04
02	Ownership doesn't start with a coded app	p. 05
03	It's like open source, with a few important caveats	p. 07
04	Ownership gets expensive	p. 08
05	When building it yourself is the right call	p. 11
06	Questions to ask before you build	p. 12
07	Visibility isn't a build-or-buy decision	p. 13

Introduction

AI-coded apps are like a new puppy: cheap to acquire, expensive to keep. Someone hands it to you, it's delightful for about a week (if you're lucky!), and then it needs vet visits, house training, and daily walks and attention that you might have forgotten about when you saw the adorable puppies at the shelter.

Software has its own well-known version of that shelter moment. AI code generation offers real benefits. Teams that used to need weeks and a hiring plan to build a new app can now whiteboard it on Monday and build a working piece of software by Friday.

The part worth more thought is what happens after Friday. It's undoubtedly fun and exciting for a marketing team to spin up their own internal tool to track campaign performance. It works, and they've solved a real pain point for their team. But three months later, the app contains protected information and exposes it publicly. The API it depends on changed its rate limits, so half the dashboard stopped updating weeks ago and nobody noticed. Nobody remembers why it isn't pulling data from two of the five ad platforms it's supposed to cover, and they don't know how to fix it. None of that came up when they built the app. It surfaces later, when you rely on what it delivers, but you don't know how to control security, cost, or functionality.

An app that exists is not the same as one that's maintained, and it makes sense to spend some time on the distance between those two states.

01 A lot has changed in software development

With vibe coding and agentic development, anyone can build a working piece of software fast, even without an engineering team.

Andrej Karpathy coined the term "vibe coding" in early 2025, describing a loose workflow where you accept whatever the AI produces without reading it closely. About a year later, he retired the term, replacing it with "agentic engineering." The basic idea is the same, but it now builds in specs, review, and oversight. He made the distinction clear: vibe coding raises the floor for who can build software. Agentic engineering is what preserves the quality bar once you actually ship it. There are plenty of tools in mainstream use either way (Cursor, Replit, Lovable, Bolt, Copilot Workspace, v0, Claude Code, to name just a few).

“ Vibe coding raises the floor for who can build software. Agentic engineering is what preserves the quality bar once you actually ship it. ”

One satirical site went so far as to rate 649 real RSA 2026 security exhibitors on how easily they could be replaced with a weekend of vibe coding. It's a joke, but 649 companies is a lot of targets for a joke. We hear versions of this in real conversations. One prospect told us, "I had Claude look at your website and estimate how long it would take me to build Flux." That statement isn't wrong about how easy building a version of something has become, but it does skip past the challenges of maintaining an app.

02 Ownership doesn't start with a coded app

Writing the application code is just part of the work. Even producing it well takes judgment. AI coding tools don't supply on their own. First, you need to understand the problem well enough to know what needs to get built. You need to make the architectural calls, too: how information gets organized and stored, how pieces talk to each other, where the system needs room to grow. Skip that thinking and the code can still run. It's just more likely to be brittle in unanticipated ways.

Once your app exists, someone still has to own:



Product decisions and domain understanding



Feature scoping



Ongoing maintenance



Support when something breaks

Product decisions don't stop once the code ships, either. A vibe-coded tool that solves today's problem will get requests for more capabilities: another view, an additional integration, a new team asking for the same thing with a few small changes. Someone has to decide which of those requests are worth building and which aren't, understand the domain well enough to know the difference, and hold that line when a stakeholder pushes back. That's product management, whether or not anyone on the team has that title. Not accounting for it doesn't remove the work. It just means the tool grows in whatever direction the loudest request pulls it, until it's solving five problems adequately instead of one problem well.

The other half of the effort is less visible until it isn't, but it's still important. It's called keeping the lights on. Dependencies need updating before they become security liabilities. You need to secure sensitive data at rest, in use, and in motion. The platform or API that the tool runs on changes, sometimes without warning. Something that worked great with ten users starts timing out at a hundred. None of this is apparent when the app first ships. These issues show up steadily afterward, and they must be identified and addressed.

```
● ● ● keeping the lights on

$ flux watch --repo internal/campaign-dash

dependencies behind latest      14
upstream api change, rate limits detected 41d ago
data sources reporting         3 of 5
owner                          unassigned
// none of this threw an error
```

A human doesn't have to be the one patching each dependency. Agents can open the PR, retry the call, or scale the timeout. But someone still has to decide if this app warrants watching, and know that the dashboard is missing two of its five sources even when nothing throws an error. And most importantly, that person needs to answer for the fix when the agent makes it worse. You can delegate the manual toil, but not the ownership. Teams that budget for building an AI-coded app rarely budget resources for this, because it wasn't part of the original request.

**“ You can delegate the manual
toil, but not the ownership. ”**

That someone is often one specific person, which creates an additional risk: when the person who built or led the internal tool leaves the company, changes teams, or gets pulled onto something else, the knowledge of how it works often leaves with them. In some cases, even the person who built it can't fully explain their own reasoning, especially if they sit outside your engineering org. The app itself is still there. But the context for why it makes certain assumptions, what breaks if you touch a particular file, which workaround exists for which edge case, none of that is written down, because it never needed to be documented before. Whoever inherits the tool has to reverse-engineer decisions someone else made and no one realized they needed to be explained.

03 It's like open source, with a few important caveats

Self-built AI software is a lot like open source: there's similar math in terms of ownership. In open source, the tradeoffs are well understood and often worth it. AI-generated software carries similar tradeoffs, but they're less acknowledged so far.

One useful example comes from an engineering leader whose company pitched an open-core platform to customers as free. Because it was open source, they could just run it themselves. But it wasn't really free. Running the platform at any real scale meant infrastructure costs that scaled per user, in addition to the ongoing work of maintaining it, patching it, and keeping it running. The software itself cost nothing. Operating it cost plenty.

Self-built AI apps run into the same issues, with one important difference. Open source solutions still give you a community. Other people (sometimes lots of them) are finding and disclosing security holes, then writing and sharing patches for them. They are patching the edge case your team never thought to test. Someone else hit a bug and already fixed it. The people guiding the open source project are making the difficult feature-selection decisions for you, and you get to take advantage of their years of collective work every time you pull the latest version.

AI-coded apps give you none of that. You might have a great looking (and working) app. But there's no one upstream. No one else has ever run this exact code, hit these edge cases, or found its flaws. Whatever goes wrong, you find out first, and you're the only one fixing it.

**“ You're not skipping the community's work.
You're doing it all yourself without realizing it. ”**

That's the real cost of free. You're not skipping the community's work. You're doing it all yourself without realizing it. And you probably won't notice that loss until something breaks.

04 Ownership gets expensive

How do you know the small tool you built has become a load-bearing app? Sometimes you don't, until something breaks. The first four examples below are documented incidents.



Security debt

Cybersecurity firm RedAccess found more than 380,000 publicly accessible apps built on Lovable, Replit, Base44, and Netlify. About 5,000 had no meaningful authentication, and roughly 2,000 exposed real corporate or personal data, financial records, medical information, internal strategy docs, and unredacted customer conversations. None of the teams behind those apps intended to expose customer data. Most probably didn't think about authentication at all, because the tool was small enough that it didn't seem worth worrying about.

380,000

publicly accessible apps

5,000

had no meaningful
authentication

2,000

exposed real corporate or
personal data



A security incident

Replit's AI agent deleted a live production database during an active code freeze, affecting records tied to over 1,200 executives and 1,100+ companies, then fabricated data about what happened. Replit's CEO publicly apologized. A code freeze exists specifically to prevent exactly this kind of change. The agent didn't ignore that safeguard maliciously, it simply didn't treat the freeze as a boundary. That's an example of the kind of failure that doesn't look like a bug until it happens.



Legal exposure

A civil resolution tribunal in British Columbia held Air Canada liable for inaccurate guidance its AI chatbot gave a customer, ruling the company responsible for what the bot said. The ruling didn't distinguish between a human agent and an AI one. The company owned the output either way. That precedent is important: if your organization owns the outcome of what your software tells people, it's worth knowing that before you deploy it.



Ongoing cost

Token spend at companies actively using AI coding agents rose roughly 10x in six months at some large firms, and up to 15x at one AI infra startup (up from \$200/month per developer to \$3,000/month per developer). Several companies now report individual developers spending \$500 to \$1,400 in a single day. The pitch for AI-assisted building is usually about speed, and it delivers on that. What it doesn't advertise is that the meter keeps running long past the first ship date, and it tends to run faster as the tool gets wider adoption and increased capabilities.

10x

in six months at some large firms

\$200 → \$3,000

per month, per developer

\$500 – \$1,400

in a single day



Succession problems

Ownership of your app is important. What happens if the person who built the tool leaves, gets promoted, or moves to a different team, and takes the context for that app with them? Inevitably, a new hire or former teammate inherits a working piece of software with none of the reasoning behind it. Every fix takes longer, because it starts with figuring out what the original build was really doing before anyone can safely change it. Unlike the other four risks in this section, this one has no external source to cite, because it rarely makes the news. It just slows a team down for months. That may be why it gets talked about less than a headline-worthy breach, but it certainly costs you time.

None of these risks are independent of each other. A tool nobody's watching closely is also a tool where feature creep goes unchecked, because internal builds rarely have anyone whose job is saying no to the next request. Focused companies have that discipline; it's what keeps them from drowning in their own roadmap. Internal builds usually don't have it, so the tool keeps growing. Whoever's left after the original builder moves on ends up maintaining that growth. And the token bill climbs past what anyone forecasted (if they even considered it).

And the more complex the tool gets, the more likely agents, and the people reviewing their output, are to miss something at scale. This is the same mechanism behind the RedAccess and Replit examples above, just playing out inside a single internal tool. Underneath all five risks in this section, two familiar scenarios keep showing up: work that never touches a ticket and so never becomes visible until it's already a problem, and risk that accumulates as volume grows, undetected until it surfaces as an incident.



05 When building it yourself is the right call

Not every decision to build an in-house app needs a high level of scrutiny. Plenty of internal software is a great fit for AI-assisted building. Here are a few examples illustrating when it makes sense.



An app with a small surface area and low stakes

For example, a script that reformats a weekly report, an internal dashboard three people on one team check once a day, or a one-off tool that replaces your clunky spreadsheet macro. If that app breaks, someone notices, fixes it, and moves on. No customer sees it. No other system depends on it. No regulator cares about what it's doing.



An app built for a small internal team

A tool built and used by the same two or three people rarely needs a real maintenance plan, because the people who built it are the people who'll (hopefully) notice when something goes wrong. A finance analyst who vibe codes a script to reconcile two spreadsheets faster is a good example: it either produces the right number or it doesn't, the analyst checks it against the source data either way, and if it breaks next quarter, the fix is a quick rewrite, but it doesn't result in an incident in production.

**How the software was built doesn't matter.
What matters is what happens if it fails.**

Low blast radius, low complexity, easy to replace: build it, use it, move on. When those conditions change, it might be worth revisiting whether your in-house app is still the right choice.

06 Questions to ask before you build

Before you decide you want to own and maintain an app in-house, consider these four questions. The more high risk answers you have, the stronger the case for buying a solution to meet your ongoing needs.

Questions to consider	Lower risk to build & maintain	Higher risk to build & maintain
System criticality. What happens if this breaks?	Nobody outside the team notices	Customers, revenue, or compliance are affected
Complexity growth. Will this stay simple, or grow?	Scope is fixed and narrow	New features and edge cases keep getting requested
In-house domain expertise. Does your team understand this problem deeply?	Yes, this is close to what you already do	No, this is a new domain for your team
Maintenance appetite. Who owns this a year from now?	A clear owner, with time and token cost budgeted for it	No one specific and there's no line item in the budget for ongoing costs

If most of your answers land in the left-hand column, it's reasonable to build a tool and keep maintenance within your company. If most fall in the right-hand column, building it yourself may cost more than purchasing from an outside vendor over the life of the solution.

07 Visibility isn't a build-or-buy decision

Everything so far has been about apps other teams build. The same question applies to the one tool you might build for yourself as an engineering leader: the engineering-intelligence dashboard meant to tell you whether your AI bets are paying off across the org. Plenty of engineering teams have the skill to build a dashboard pulling from GitHub, or a script tracking PR velocity. But a self-built dashboard has the same weakness as any tool that aggregates signals from across your toolchain: it's only as reliable as its shakiest input. For most homegrown tracking, that input is ticket data, the exact source most engineering leaders already don't fully trust.

The real gap is what most teams don't build. It is the same ongoing work already described, never finished, always changing, except now it's aimed at estate-wide visibility instead of one app. And it has to hold up as your codebase, your tools, and your AI adoption keep changing. Unless someone owns it as an ongoing part of their job, it's unlikely you'll be able to keep maintaining it to meet your evolving needs. Picture the same tool six months from now, after two more teams adopted AI coding agents and the original scope expanded significantly. It has the same maintenance and succession problems addressed earlier, but aimed at the one tool you're relying on to track everything else.

That's true no matter which way you go on the build or buy decision. Whoever, or whatever, is generating the code, someone still has to see what's actually happening inside your codebase. You need to know which parts of the codebase nobody's touched since the person who understood them left. You need to know when a script that started as a two-person tool picked up dozens of dependencies, and you need to know it before a security researcher finds it for you. Build the app, and that need is still there. Buy the app, and it's still there.

Flux exists to answer a question that building or buying the app, on its own, never answers. What is happening in this code right now, and is anyone watching it?

After the app works, who owns it?

Building software has never been easier. But owning it, like owning a puppy, hasn't gotten any simpler.

The ownership decision doesn't end at deployment. Every app you build or vibe code into existence still needs someone watching what happens to it next, the way every puppy needs someone's time and dedication long after the cute stage is over.

You don't inherit an upstream community when you use AI tooling to code an app. No one is watching for what breaks except you. That's who owns it: you.

Flux doesn't own your apps for you. It shows you the file nobody's touched since the person who understood it left, the dependency that turned into a security hole, the tool whose token bill nobody's watching. You still do the work. Flux makes sure you know what's happening in your codebase.

Flux: Conquer the chaos. Lead with code-first clarity.

About Flux

Flux is a code-first engineering intelligence platform that helps engineering leaders make better decisions with ground-truth visibility into the work actually happening across today's complex, AI-accelerated codebases. Instead of relying on tickets, Flux reveals the work teams are doing, surfaces risk and technical debt, and connects engineering activity to business outcomes. With this visibility, leaders can distinguish innovation from maintenance, spot emerging issues before they become incidents, and understand collaboration patterns to build healthier, higher-performing teams across their engineering organization.

Learn more at www.askflux.ai, explore the resources, and follow Flux on LinkedIn.