The Ultimate Personal Project Playbook

Introduction

Personal projects are how most software engineers grow. They're how you get past tutorial hell, how you learn what a real system looks like end-to-end, and they're often what gets you the interview at the company you actually want to work for. Even with how good large language models have gotten at writing code, the engineers who stand out are still the ones who can take an idea, scope it, design it, and actually ship it. LLMs will happily write you 5,000 lines of code for a project that has no clear scope and no working architecture. That's how you end up with a half-finished mess in your GitHub.

Despite how important personal projects are, almost no one teaches you how to plan one. You learn data structures in school. You learn algorithms for interviews. You might even learn system design from my other guide. But planning? That's the part everyone skips, and it's the reason 90% of personal projects die in a folder on your laptop called `cool-app-v2-final` (I have a few of these lol).

This guide is the playbook I wish I had when I was a new grad. It's not a 200-page software engineering textbook. It's not a startup founder's guide to building a unicorn. It's a concise, opinionated walkthrough of how to take an idea in your head and turn it into something real. In about 20 pages, you'll learn:

- How to pick a project that you'll actually finish
- How to scope it without spending a month "planning"
- How to design the system without over-engineering it
- The trade-offs you'll keep running into and how I think about them
- A full worked example you can copy

This guide assumes you can already code. You know what an API is, you've written a function, maybe you've used a database. What you may not have done is taken something from idea to deployed application. That's what we're fixing.

A quick note on style: throughout this guide, I'll be opinionated. I'll tell you what I think you should do and why. None of this is law. If you have good reasons to do something different, do it. But if you're staring at an empty editor and don't know where to start, follow the defaults.

Picking a Project

This is the hardest part for most people I meet. You sit down on a Saturday morning, full of energy, ready to build something cool, and 4 hours later you've watched 3 YouTube videos on Rust, opened 11 tabs of "best side project ideas 2026," and written zero lines of code. I've been there. Everyone has been there. Picking a project is a real skill, and we're going to break it down.

Why Most Personal Projects Die

Before we talk about how to pick a good project, let's talk about why the average personal project ends up as a graveyard repo. In my experience, almost every dead project dies for one of these reasons:

1. **It was too ambitious.** You decided to build "a Spotify competitor" or "a better Twitter." You can not build Spotify. Spotify has 3,000 engineers. You have a weekend.
2. **You couldn't define when it was "done".** You kept adding features. There was always one more thing. The project never reached a state where you could call it shipped.
3. **It solved a problem you don't have.** Building a meal planning app is hard when you don't cook. Building a fitness tracker is hard when you don't work out. Pick something you actually use.

Most of the advice in this section is just trying to help you avoid these traps. If your project idea triggers any of them, you should probably pick something else.

Where Good Ideas Come From

Forget "side project idea generators" or GitHub repos with 100 sample projects. The best project ideas come from one of three places:

Annoyances in your own life. This is the gold standard. Something you do every week that's annoying, something you Google constantly, something where you've thought "there should be a tool for this." If you build it, you have at least one user (you), and you have built-in motivation to finish, because you actually want to use the thing. This is how I've built 80% of my projects.

Tools to help with your hobbies. If you play League of Legends, build a post-game analysis tracker. If you run, build a route planner that does *exactly* what you want. If you read, build a reading log. The advantage here is the same as above: you understand the problem domain better than 99% of engineers.

Curiosity-driven exploration of a technology. This is the riskier path, but there is a time and a place for it if you want to fill a knowledge gap. You read about WebRTC and want to know how it works, so you build a tiny video chat app. You read about vector databases and want to play with them, so you build a personal search engine over your notes. The trick here is to pair the technology with a tiny, real use case. "Learn WebRTC" will fail. "Build a video chat for my D&D group using WebRTC" will probably ship.

What you should *not* do is sit down with the goal "build a cool side project to put on my resume." These almost never finish. The intrinsic motivation just isn't there.

Filtering Your Ideas

Once you have a few candidate ideas, you need to filter them. Here's the rough mental checklist I run through. If your idea fails on more than one or two of these, pick a different idea.

Can you describe it in one sentence? "It's like Strava but for cyclists who want to plan routes through coffee shops." That's a sentence. "It's a platform for X, Y, Z, with social features and an AI layer and..." that's not a sentence, that's three projects. If you can't compress it down, you don't actually understand what you're building yet.

Do you actually want to use it? If the answer is no, you will quit on week 2 the first time you hit a hard problem. Personal projects are powered almost entirely by intrinsic motivation. You need to actually want the end product to exist.

Are you the dumbest possible user of it? Some projects need expert users. "A trading bot for institutional finance" needs a finance background. "A note-taking app" needs anyone with a brain. Pick projects where you (yes, you) are squarely in the target audience. You will be your own beta tester, your own bug reporter, and your own product manager. If you're not the user, you have no idea what “good” looks like.

What does success look like? This one is killer. Before you write a line of code, you should be able to finish the sentence "this project is done when ____." If your answer is "when it has lots of users" or "when I've learned everything about Kubernetes," you've already lost. A good "done" definition is concrete: "I can log in, add a habit, mark it complete, and see a streak." That's the finish line. Without one, the project never ends, which means it never really starts.

If your idea passes this filter, great. You're ready to move on to scope. If it doesn't, kill it and go again. Killing ideas before you've written code is essentially free. Killing them after you've written 3,000 lines is not.

Defining Scope

In the system design playbook, I talked about how most engineers jump into the design too quickly. The same is true (maybe even *more* true) for personal projects. The difference is that on a personal project, you don't have an interviewer or a PM to pull you back. You have to do it yourself. This section is about being honest with yourself about what you're actually going to build.

The MVP

You've heard "MVP" (minimum viable product) thrown around a million times. Most people use it wrong. They think MVP means "a version with fewer features." It's deeper than that. MVP means: **what is the smallest possible thing I can build that proves the idea works and that I'd actually use?**

For a habit tracker, that might be:

- One user (you)
- One habit at a time
- Mark it complete or not for today

No streaks. No reminders. No login screen. No "social" features. No themes. No mobile app. Just the absolute core loop. You can always add more later, and you should, because "later" is when you actually know what's worth adding. But the most important thing about an MVP is that it teaches you what the real version should look like. You don't know yet. You think you do. You don't.

I personally have never built a personal project where my MVP plan and my final version looked the same. I always cut things, add things, and rearrange things based on what I learned by actually using the v0.1. That's the whole point. An MVP is a learning tool, not a release.

Functional Requirements (Personal Edition)

In the system design playbook, functional requirements were the user-facing features, and we focused on getting them clear before designing anything. The same applies here, just at a much smaller scale.

Example: A habit tracker. Here's how I'd write the functional requirements for the MVP:

Functional Requirements

- I can create an account with email and password
- I can add a habit (just a name)
- I can mark a habit as done for today
- I can see a list of my habits and which ones are done today
- I can delete a habit

Anything beyond that is post-MVP.

Notice what's *not* on there:

- No streaks
- No notifications
- No "share with friends"
- No analytics dashboard
- No mobile app
- No dark mode
- No recurring habit schedules ("only Mondays and Wednesdays")
- No tags or categories

Every single one of those features could be added later, and all of them are nice. But none of them are the core loop. If your MVP requirement list has more than ~7 bullet points, You're building v1.0, not the MVP.

A good question to ask yourself for every feature: "if I removed this, would the project still be useful to me?" If the answer is yes, cut it from the MVP. You can always add it back when you're bored on weekend three.

Non-Functional Requirements (Personal Edition)

Non-functional requirements are where personal projects diverge hard from real-world systems. In the system design playbook, we talked about scalability, latency, availability, durability, consistency, security, and observability. For a personal project that you're going to use yourself? Most of that doesn't matter. Here's how I think about each one.

Requirement	**Real System**	**Personal Project**
Scalability	10M DAU, 1K writes/sec	1 user (you), maybe 10 if your friends try it
Latency	< 200ms	< 2 seconds is fine
Availability	99.99% uptime	"It's up when I check it"

Requirement	Real System	Personal Project
Durability	Never lose a write	Don't lose my actual data, occasional bug is fine
Consistency	Strict ACID across regions	One database, one region, you'll be fine
Security	SOC2, encryption at rest, audit logs	Don't store passwords in plaintext, use HTTPS
Observability	Metrics, logging, alerting	You'll find out it's broken when it's broken

Treat this list like permission to ignore most of what you'd care about at work. You are not Netflix. You do not need a multi-region active-active deployment. You do not need 5 nines. You need an app that works for you and a small handful of users on a single small server. Resisting the urge to over-engineer is one of the most important skills in personal projects, and one of the hardest to learn for engineers coming from big tech, where over-engineering is sometimes literally rewarded.

That said, two non-functional requirements *do* matter, even at this scale:

Security. Don't store passwords in plaintext. Don't expose API keys in your repo. Use HTTPS. If you're going to have other people log in, take auth seriously enough that you don't get hacked. We'll cover this more in the design section.

Durability of your own data. If you spend three months building a journaling app and lose all your entries because you didn't back up the database, you will be sad. Set up a basic backup, even if it's just a cron job that dumps your database to S3 once a day.

Everything else, you can ignore *to start*.

Planning the Build

Now that you have a scoped MVP, you need a plan to actually build it. The good news is that planning a personal project is way simpler than planning a team project. You don't need a Jira board (though I won't judge if you make one). You just need to know what to build first, what to build next, and a rough sense of when each thing is done.

Milestones

The biggest planning mistake I see is people writing down dates. "I'll be done with auth by Saturday. I'll be done with the database by next Thursday." This works in a team setting where you're full-time on the project. It does not work for a personal project that you're squeezing into evenings and weekends, because life is going to happen, you're going to lose a weekend to a wedding, and now your "plan" is broken on day 4.

What works much better is **milestones**. A milestone is a concrete, demonstrable state of the project. It's not "spend 2 hours on the database," it's "I can save and load a habit." That milestone is either done or it isn't, and you don't care if it took you 3 hours or 6.

Here's an example milestone list for the habit tracker MVP:

1. Project skeleton runs locally (frontend, backend, empty DB)
2. I can hit the backend from the frontend (just a "hello world" endpoint)
3. The database has a habits table and I can manually insert a row
4. I can create a habit through the UI and it's saved
5. I can list my habits in the UI
6. I can mark a habit done for today and it's saved
7. I can delete a habit
8. Auth works (basic email/password)
9. It's deployed somewhere I can hit it from my phone
10. I've used it for a week without it breaking

Each of these is binary: done or not done. You can pick up the project at 9pm on a Tuesday, work for an hour, and either move a milestone forward or not. No calendar required.

I'd also note that milestones 1, 2, 3, and 9 are roughly the "walking skeleton," which is so important I want to call it out separately.

The Walking Skeleton

The single best advice I can give you for any personal project is: **build the walking skeleton first.** A walking skeleton is the smallest end-to-end version of your app where every layer is connected, even if no layer does anything useful. For our habit tracker, that means:

- A frontend that renders an empty page
- That talks to a backend that returns a hardcoded response
- That talks to a database that has the schema set up but no real logic
- All of it deployed somewhere on the internet

No features, but every wire is connected, end to end.

Why this matters: most of the pain of building a project is in the connections, not the features. Setting up CORS between your frontend and backend. Getting your database connection string right in production. Figuring out why your deployment is failing. These are the things that will eat your time and energy if you save them for last. If you build the walking skeleton first, you front-load all of that pain in the first few hours of the project, when you have the most energy and motivation.

Once the walking skeleton is up and the wires are connected, adding features is comparatively easy. You're just adding flesh to a body that already walks. Every milestone after the skeleton is "add a feature to a working app," not "wire up a new layer of infrastructure."

Picking a Tech Stack

Personal project tech stack debates are some of the most pointless arguments on the internet. People will tell you to use Rust, Elixir, Haskell, the bleeding-edge framework that came out last week. Ignore them.

For 99% of personal projects, your tech stack should be **boring**. Boring technologies have huge communities, mountains of documentation, working examples for every problem you'll hit on Stack Overflow, they're what companies actually use and hire for, and they don't break unexpectedly.

Here's my default stack for a web-based personal project:

Backend: Whatever you already know best. If you don't know any, use Python (FastAPI) or Node (Express). Both have mountains of tutorials.

Frontend: React with Vite. It's not the prettiest, it's not the fastest, but it works, and the ecosystem is enormous. If you hate React, use Svelte or Vue. Anything else, you're on your own.

Database: PostgreSQL. Always. We'll talk more about why in the design section.

Hosting: Whatever has a generous free tier. As of writing, that's Render, Fly.io, Railway, or AWS if you know what you're doing. Vercel for the frontend if it's purely static or Next.js. We'll cover this later too.

Auth: Use a managed auth provider (Clerk, Auth0, Supabase Auth, Firebase Auth). Do not roll your own. We'll cover this later too.

If you're using this project to learn a *specific* technology, then yes, swap that one piece in. But swap *one* piece. Don't pick "Rust + a custom database + a brand-new frontend framework + my own auth" all at once. You will be debugging your stack instead of building your app, and you will quit. I have seen this kill so many projects.

Estimating (Badly)

Here's a fun fact about software estimation: it's nearly impossible. You will be wrong. Senior engineers with 20 years of experience are wrong about how long things will take. The trick isn't to estimate accurately, it's to **estimate roughly and adjust quickly.**

For a personal project, here's a rough heuristic that has worked for me:

- A small feature (like "add a delete button"): 1-3 hours
- A medium feature (like "add auth"): 1-2 evenings
- A big feature (like "build a real-time sync system"): a full weekend or more
- The whole MVP: roughly 3-5 weekends, if you've cut scope correctly

If your estimate for the whole MVP is more than 5 weekends, you have not cut scope enough. Go back to the previous section and cut more.

A good practice is to write your milestone list, put a rough estimate next to each one, then add up the estimates and double the total.

Designing the System

Now we get to the part that the system design playbook really prepared you for. We've gathered our requirements, we've cut our scope, we've got a milestone plan. It's time to actually decide what the system looks like. This is where most engineers' instincts get them in trouble, because they've spent their careers reading about Netflix, Google, and Meta architectures, and they reach for those patterns on day one of a project that's going to have 3 users.

Design for One User First

This is the single most important rule of personal project architecture: **design for the first user, not the millionth one.**

Specifically: design for *you*, alone, on your laptop. If your design wouldn't be drastically simpler if you knew there'd only ever be one user, your design is wrong. You can always evolve toward

more users later, but starting with a complex distributed system on day one is the fastest way to never ship.

Here's what designing for one user looks like:

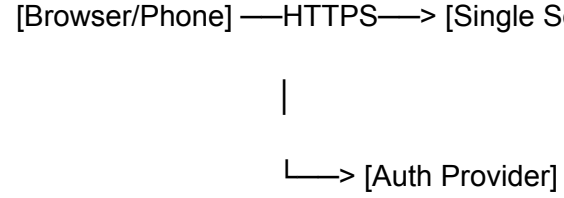
- **One server.** Don’t add the complexity of horizontal scaling yet.
- **One database.** Postgres on the same server (or a tiny managed instance) is fine.
- **No queues.** You're not handling enough traffic to need them.
- **No cache.** If your queries are slow with one user, they'll be slow with a million. Fix the query, not by adding Redis.
- **No microservices.** Bruh.
- **No Kubernetes.**

I know. This list is depressing because it's the opposite of every cool blog post you've read. But all those cool blog posts are about systems that earn millions of dollars and are run by hundreds of engineers. You don't have hundreds of engineers. You have you, two evenings a week, and a free tier. Boring is the right answer.

If your project ever gets popular enough to need any of those scaling tools, that's an *amazing* problem to have. Solve it then. Most of the time, you won't need to, because most personal projects don't go viral, and that's totally fine.

The Default Personal Project Stack

Pretty much every personal project I build starts with the same architecture, and it works for 99% of cases. It looks like this:



That's the whole thing. One server (running your backend), one database, one auth provider. Static frontend assets are usually served by a CDN or a static host (Vercel, Netlify, Cloudflare Pages), so technically there's a fourth box, but it's not running any logic.

Compare this to the architecture diagrams in the system design playbook with load balancers, multiple servers, caches, queues, multi-region replication, and so on. You don't need any of that. The only time you start adding pieces is when something breaks or gets noticeably slow with real users hitting it. And again, that's a great problem to have when you have it. But you almost certainly don't have it on day one.

Let's break down each piece.

Picking a Database

I'm going to make the database decision for you: **use Postgres.** Just use Postgres.

I know I have a section in the system design playbook about all the different database types and when to use each one. That's all true. For personal projects, none of it matters, because Postgres is good at all of it. It does relational data extremely well. It has decent JSON support if you want a document store flavor. It can do full text search well enough. It can do geospatial queries. It can be your queue (we'll talk about why this is fine for you). The only thing it really doesn't do is graph databases at scale, and you don't need that.

Why:

1. **Postgres has the best ecosystem of any open source database.** Tutorials, ORMs, hosting options, debugging tools, all of it.
2. **It scales further than you'll ever need.** Companies serve millions of users on a single Postgres instance. Your 5 users will not be the limit.
3. **You can actually deploy it for free or near-free.** Supabase, Neon, Render, Fly, all have Postgres options on a free or extremely cheap tier.
4. **You're going to learn Postgres at your real job.** This is also useful knowledge.

The one exception I'd make: if your entire project is fundamentally about a different data model (you're literally building a graph traversal app, or you're learning vector databases for an AI project), then yes, pick the database that matches. Otherwise, Postgres.

Authentication

Auth is one of those topics where the gap between "this is easy" and "you have a security disaster" is razor-thin. People hand-roll auth, store passwords with bad hashing, leak session tokens, and end up shipping a project that's actively dangerous. Even if you know what you're doing, hand-rolled auth eats up days that you could be spending on actual product features.

My strong recommendation: use a managed auth provider. I personally like Auth0 and Supabase Auth, but Clerk, Firebase Auth, and a few others work fine too. They handle:

- Password hashing
- Session management
- Email verification
- Password resets
- (Optional) OAuth login with Google, GitHub, etc.
- (Optional) Multi-factor auth

For free or near-free, on a tier that fits any personal project. There is no reason to roll your own.

The one trade-off here is vendor lock-in. If you build your whole app around, say, Clerk, you're going to have a bad time migrating off of Clerk later. For a personal project, I don't think this matters at all. You can always migrate if you really have to. But if you're paranoid about it, use Supabase Auth or a self-hostable option like SuperTokens. Those let you bail later if you really want to.

The one situation I'd hand-roll auth in: if your *entire project* is about learning auth (you're explicitly trying to understand JWTs, OAuth flows, password hashing, etc.). Then yes, do it. But know that you're trading project completion speed for learning, and be okay with that trade.

Hosting and Deployment

I get asked this all the time: "where should I deploy this?" The answer depends on what you're building and how much you want to learn about ops. Here's my mental model.

If you want to ship the fastest: Use a platform-as-a-service (PaaS). Render, Railway, Fly.io, and Vercel are the main ones at the time of writing. You push your code, they run it. You don't deal with servers, networking, certificates, any of it. They all have free tiers that are plenty for a personal project. The trade-off is that they're more expensive at scale, and you have less control over the runtime. For a personal project, who cares.

If you want to learn cloud infrastructure: Use AWS, GCP, or Azure. You'll learn EC2, S3, IAM, VPCs, load balancers, all of it. The trade-off is that you'll spend a *lot* of time on plumbing instead of building your actual app. This is fine if your goal is to learn cloud infra, but it's not fine if your goal is to ship the project. Pick one.

If you want to learn containers: Use AWS ECS, GCP Cloud Run, or just Docker on a small VPS (DigitalOcean, Hetzner, Linode). Cloud Run is honestly a sweet spot for personal projects, because it scales to zero (you pay nothing when no one's using it) and it's a real production-grade tool.

If you have specific requirements (like running a model, GPU work, or something niche): Pick the host that supports it. Most managed PaaS options don't do GPUs.

For frontend hosting specifically, the answer is almost always Vercel, Netlify, or Cloudflare Pages. They're free, fast, and they handle CDN, certificates, and atomic deployments for you.

The Scaling Question

I want to address this directly because I know what you're thinking: "but what if my project goes viral?"

It almost certainly won't. The vast majority of personal projects never get more than a handful of users, and that's completely fine. The point of the project is what you learn building it, the thing it does for you, and (sometimes) the impression it makes in interviews. None of those goals require scale.

But what if it does get popular? Then your simple architecture will start to show cracks. Your single server will hit its CPU or memory limit. Your database will get slow. Pages will start timing out. *This is a great problem to have.* It means real people are using your thing. When that happens, here's the order I'd add complexity in:

1. **Vertical scaling first.** Bump up the size of your server and your database. This will get you a long way and doesn't require any code changes.
2. **Add a cache.** If certain queries are expensive and getting hit repeatedly, drop in Redis (or Memcached) and cache the responses.
3. **Add a CDN for static assets.** Most platforms do this for you, but if not, Cloudflare is free and excellent.
4. **Move heavy work to a queue.** If certain endpoints are slow because they're doing big jobs synchronously, move that work to a background queue. The queues section of the system design playbook covers this.
5. **Horizontally scale your backend.** Add more instances behind a load balancer. This requires your backend to be stateless, which it should be anyway.
6. **Read replicas for the database.** If your reads are slow because the database is overloaded, add read replicas.
7. **Eventually, sharding, microservices, etc.** This is where you're a real grownup with real users and real money. Welcome to the next stage.

You will probably never need anything past step 3. That's fine. The point is that the simple architecture you start with can evolve gracefully. You're not going to "have to rewrite everything" if it gets popular. You're just going to add pieces, one at a time, as needed.

Trade-Offs You'll Actually Make

The system design playbook is full of trade-offs: consistency vs availability, latency vs durability, monolith vs microservices. Those are all real. But for personal projects, you're going to hit a different set of trade-offs, ones you don't really see in interview prep.

Build vs Buy

For every feature you're building, you have a choice: write it yourself, or pay (or use a free tier) for someone who's already written it. Auth is the classic example, but it shows up everywhere:

- Email sending: write your own SMTP integration, or use Resend / Postmark / Mailgun
- Payments: build your own checkout, or use Stripe
- Analytics: build your own, or use Amplitude / Posthog
- File storage: run your own, or use S3 / Cloudflare R2
- Push notifications: build your own, or use OneSignal / Pusher

In almost every case, **buy.** Or, more accurately, "use the free tier of someone else's service." Building these things yourself is a tar pit. Even if you "finish," you'll be 80% as good as the dedicated tool, and you'll have spent a month not building your actual product.

The one caveat: if a specific piece is *core* to your product, build it yourself. If you're building a competitor to Stripe, fine, build your own payments. But if payments are a means to an end, just use Stripe.

There's also a learning angle here. Sometimes you want to build a thing yourself because you want to know how it works. That's totally fine, but be honest with yourself about the trade-off: building it yourself will make the project take longer and probably less polished.

Custom vs Boilerplate

Closely related is the question of starting from a template (boilerplate) versus starting from scratch.

The boilerplate world is full of starter kits: Next.js + Supabase + Stripe + Tailwind, all wired up, with auth and payments and a landing page. You can clone one of these and have a "real" app running in 10 minutes.

The trade-off: you're starting with a codebase you don't fully understand. There will be code in there for things you don't need, configurations you didn't choose, and patterns that may not

match how you'd naturally write the project. The first time something breaks, you're debugging unfamiliar code.

My take: **for a project you actually intend to ship, use a boilerplate.** It saves you days of grunt work, and most of the initial setup is the same boring stuff every project needs (auth, deploy, basic styling). For a project where the *point* is learning, build from scratch. You'll learn way more, even if it takes longer.

Either way, do not spend more than one weekend on initial setup. If you're past that, you're probably yak shaving. Get to the actual product.

Polish vs Ship

The instinct is to keep polishing: tweak the colors, redesign the landing page, refactor the API, write better tests. None of that is *bad*, but it's not what makes your project succeed or fail.

The thing that determines success is whether you ship. A janky shipped project is worth a hundred "almost done" projects sitting on a hard drive. I ship janky stuff ALL THE TIME, and my projects have thousands of users. People DM with bugs, I fix them. Shipping is a forcing function. It exposes bugs, gives you feedback, and (most importantly) gets you to the next project, where you'll apply what you learned.

A heuristic I've used: **if a feature isn't core to the value of the app, and it can be added later without touching everything, defer it.** Loading states? Defer. Empty state illustrations? Defer. Onboarding flow? Defer. Custom error pages? Defer. Beautiful animations? Defer.

The trap is that all of these things feel like "almost done" work. You're 90% there, you just need to clean it up. But cleanup expands to fill all available time. At some point, you have to declare it shipped, even if it's ugly.

Learning vs Shipping

The last big trade-off is between learning and shipping. These are sometimes the same goal, but often they're different.

If your goal is **learning**, your project might never be "finished" in the product sense, and that's fine. You're optimizing for the depth of what you understand at the end. Maybe you write a custom auth system, build your own ORM, hand-roll a queue. None of these will help you ship faster, but they'll all teach you a ton.

If your goal is **shipping**, your project needs to actually work and be usable. You should be using every shortcut available, every managed service, every boilerplate. The shipped product is what matters, not how much of it you wrote yourself.

Most personal projects are some mix of the two. The mistake is not being honest about which one you're prioritizing in any given decision. When you reach for the harder, from-scratch path, ask yourself: am I doing this because it's the right call for the product, or because I want to learn the thing? Both answers are valid, but they lead to different decisions. Be honest about which one you're chasing.

Worked Example: Designing a Habit Tracker

Let's walk through everything we just covered, end to end, on a real example. I'm going to design a habit tracker. Why a habit tracker? Because it's small enough to actually finish, useful enough that I'd actually use it, and complex enough to involve auth, a database, and some interesting product decisions.

Step 1: Pick the Project

Idea: **Track daily habits.** I want to mark whether I did the thing today. I want to see a streak (because streaks are motivating). I want it to be on my phone, so I can mark stuff before bed.

Filter check:

- One sentence: "A simple daily habit tracker with streaks." ✓
- Buildable in 2-4 weekends: Yes, if I cut hard. ✓
- Would I use it: Yes. ✓
- Am I the user: Yes. ✓
- What does done look like: "I can log in, add a habit, mark it done for today, and see my streaks." ✓

Step 2: Define Scope

Functional Requirements (MVP)

- I can sign up and log in (email/password)
- I can add a habit (just a name)
- I can see all my habits
- I can mark a habit as done for today
- I can see the current streak for each habit
- I can delete a habit

Non-Functional Requirements

- Works on my phone (responsive web is fine)
- Loads in under 3 seconds
- My data isn't lost
- HTTPS, no plaintext passwords

Things explicitly NOT in scope (but would be nice later):

- Reminders/notifications
- Sharing with friends
- Charts/analytics
- Recurring schedules ("only Mondays and Wednesdays")
- Tags/categories
- Mobile app (just responsive web)
- Multiple users on one account
- Import/export

I had 6 MVP features. That's at the upper end. I'm sticking with it because they're core to the loop.

Step 3: Plan the Build

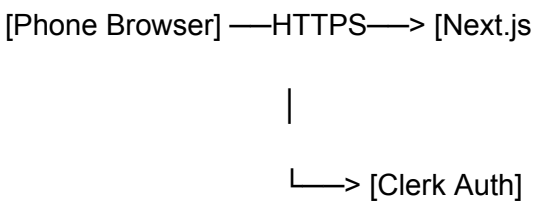
Milestones:

1. Empty Next.js + Postgres skeleton runs locally
2. I can call a "hello world" API from the frontend
3. Postgres has a `habits` table with the right schema
4. Auth works (using Clerk)
5. I can create a habit through the UI
6. I can list my habits
7. I can mark a habit done for today
8. The streak calculation works
9. I can delete a habit
10. It's deployed (Vercel + Supabase)
11. I can log in from my phone and use it
12. I've used it for a week without it breaking

Estimate: roughly 2 weekends, with a buffer for deployment and bugs. Let's see how this holds up.

Step 4: Design the System

Architecture:



That's it. One server (Vercel running Next.js API routes), one database (Supabse), one auth provider (Clerk).

Trade-offs I made:

- **Postgres for everything**
- **Clerk for auth**, which means vendor lock-in, but saves me probably 2 weekends of work.
- **Vercel + Supabase**, both free tier. I'm betting the project stays small enough to stay on free tiers, and if not, I have a great problem.
- **No caching**. With < 100 users, no query is going to be slow enough to need it.
- **No queue**. Nothing here is async.
- **Server-rendered with Next.js API routes**, so I have one codebase, not two. Trade-off: more vendor lock-in to Vercel, but again, who cares for now.

Step 5: Build It

I won't go through the whole code, but the order I'd build in:

1. **Walking skeleton.** Next.js project, deployed empty to Vercel, connected to a Supabase Postgres with the schema applied.
2. **Auth.** Wire up Clerk, get login working, protect a single page.
3. **Habits CRUD without streak.** Just the basic create/list/delete flow, no streak math. hours.
4. **Mark complete.** The UI to mark a habit done for today, write to the `habit_completions` table.
5. **Streak calculation.**
6. **Polish for daily use.** Make the UI not embarrassing on my phone. Add a date check so "today" works correctly with timezones.
7. **Use it for a few days.** Find bugs. Fix them.

Step 6: Ship

At some point, you have to call it shipped. For me, "shipped" on a personal project means:

- The walking skeleton works in production
- All MVP features work
- I've personally used it for at least a couple days
- I haven't pushed a fix in 3 days (means it's stable enough)

When all four are true, I declare it shipped, write a quick post about what I learned, and move on. I might come back and add features later (the habit tracker is a great candidate for adding charts, recurring schedules, etc.), but the project is "done" in the sense that there's a working version I'm using.

That's the whole worked example. You can apply this same flow to anything: a recipe organizer, a workout logger, a budget tracker, a reading list, whatever. The pattern doesn't change. Pick a project, scope it, plan in milestones, design for one user, ship.

Telling People About It

If you want anyone else to use it, or you want to use it for job applications, you need to tell people about it. This part feels uncomfortable for most engineers, but I think it’s worth doing in almost all cases. This can let recruiters discover you, and it’s how you learn how much of a pain in the ass it is to maintain software for actual users.

Some lightweight options:

- **Write a post about what you built.** What you learned, what trade-offs you made, what you'd do differently. A surprisingly good number of recruiters and engineers read these.
- **Post it on Reddit, Hacker News, or Indie Hackers.** Be honest about what it is and isn't. Don't oversell. The hardest part of these communities is being authentic, not promotional.
- **Add it to your resume / GitHub README.** Don't just say "built a habit tracker." Say what trade-offs you made, what you learned, and why you chose what you chose. That's what hiring managers are actually looking for.
- **Show it to your friends.** They probably won't use it forever, but they'll give you feedback that improves it.

You don't have to do any of this. A project just for you is also valid. But if you're hoping for the project to do work for you (resume, audience, learning in public), shipping it visibly is the next step after shipping it technically.

Maintenance (or Not)

Once you've shipped, you have a choice: keep maintaining it, or let it sit.

Maintenance is real work. Servers get OS updates, libraries get vulnerabilities, hosts deprecate features, free tiers shrink. If you have 0 users besides yourself, none of this is critical, you can patch it when you feel like it. If you have real users, especially paying ones, maintenance becomes mandatory.

For most personal projects, my honest recommendation is: **maintain it for as long as you're personally using it, then let it go.** When you stop using it, archive the repo, take down the deployment, and move on. There's no shame in retiring a project that taught you what it needed to teach you.

If you do want to keep it alive long-term, build maintenance into your routine. Once a month, pull dependency updates, run the tests, deploy a fresh build, click through the app, fix anything that's broken. An hour a month, and the project stays healthy.

What's Next?

If you've made it this far, you have everything you need to plan and design a personal project that you'll actually finish. The next step is to pick an idea (a small one), apply this playbook, and start building. The whole point is to ship, and the only way to do that is to start.

If you want to keep going from here, feel like you need more guidance, want to get some eyeballs on your system design doc, or want to chat with me (and other people like you) directly about your personal project check out **The Daily Dev**, my system design app and private community for software engineers and builders who want to keep getting better at this stuff together.

I'd also strongly recommend reading the Ultimate System Design Playbook if you haven't already. The two are designed to be companions: this guide gets you to a working personal project, and that one gets you ready to design the kinds of systems you'll work on at scale, both at your job and in interviews.