What to Code

What to Code

The Indie Hacker's Guide to Finding Profitable Micro-SaaS Ideas in the Vibe Coding Era

By Finxter Publishing

For the builders of this new era-may you choose wisely, create boldly, and shape a future worth coding.

Table of Contents

1. The New Bottleneck

> ***"When code gets cheap, judgment becomes the new moat."***

For most of the software era, the hard part was building.

If you had an idea for a tool, a workflow, a product, or a business, the first real question was whether you or your team could actually make it. That question filtered everything. It filtered who got to participate, which problems were worth pursuing, how quickly products could be tested, and how much risk came before learning anything useful.

Code was expensive. Talent was scarce. Time-to-first-version was long. Even simple products took coordination, technical skill, and patience. The bottleneck lived in implementation.

That is changing fast. Today, a single person can do work that used to require a small team. A developer with modern AI tools can prototype in hours what once took weeks. A non-developer using natural language, templates, and AI-assisted builders can create internal tools, automations, dashboards, agents, and lightweight products without writing much code at all. Even when the output is rough, the cost of getting to rough is collapsing.

This matters more than it first appears. When a constraint weakens, advantage moves elsewhere. When implementation becomes cheaper, the market does not simply reward everyone equally. It shifts its rewards toward the next scarce thing. In the age of AI and vibecoding, that scarce thing is increasingly judgment.

Not just technical judgment. Not just product taste. Broader judgment.

- What problem is worth solving?
- For whom?

- Why now?
- What form should the solution take?
- What should be automated, and what should stay human?
- What looks impressive but creates no value?
- What sounds small but hides a durable need?
- What can you uniquely understand, distribute, sell, or sustain?

These questions used to sit upstream of coding. Now they sit at the center of advantage:

I have spent more than a decade writing code, building side projects, and teaching developers how to think clearly about software. Long before AI became the default lens for every new product, I wrote more than ten programming books, including *The Art of Clean Code*, completed a PhD in computer science, and built Finxter into a large educational platform with an audience of more than 130,000 readers, many of them drawn through Python and now increasingly through AI and live coding. I have watched the center of gravity move firsthand: from learning how to code at all, to deciding what is worth coding when software has become radically easier to produce.

This book is written primarily for indie hackers, technical founders, and AI-native builders who can now ship software quickly but need a better system for choosing what is actually worth building.

You can easily find a million books, tutorials, and bootcamps teaching you exactly *how* to code. But in a world where AI can generate a functional app in seconds, knowing *how* is no longer your competitive advantage. Almost no one is rigorously teaching the much harder, much more lucrative skill: deciding *what* to code. The fundamental question now sits upstream of syntax: what should you build if you want to create real value? Leave the h*ow* to the AIs – focus on the *what*.

It is not a book about how to prompt a coding model, how to become a better engineer, or how to chase the latest AI app wave before it fades. It is about the higher-leverage skill that becomes more important as creation gets easier: deciding what to build. That sounds obvious until you look at how most people still think.

Many builders are using twenty-first century tools with twentieth century assumptions. They still behave as if the main game is execution speed. They obsess over frameworks, model releases, stack choices, and tiny implementation details before they have established that the thing deserves to exist. They overvalue being able to build and undervalue being able to choose.

But when more people can build, building itself stops being a strong filter. If ten thousand people can generate an app, a chatbot, an automation, or a micro-SaaS in a weekend, then the mere existence of that thing means very little. The question is no longer, "Can this be built?" Very often, the answer is yes. The more important question is, "Should this be built, by me, for these users, in this way?"

A mediocre builder with excellent judgment can now go surprisingly far. They can use tools, collaborators, models, and templates to cover many implementation gaps. A brilliant builder with poor judgment can move faster than ever in the wrong direction. They can produce a polished waste of time at historic speed.

You can see it in startup culture. The old fantasy was that if only you had the technical skill, the product could exist. Now technical barriers are lower, but that has not produced a world full of valuable software. It has produced a world full of more software, much of it redundant, shallow, undifferentiated, or disconnected from any real pain. The volume increased. The hit rate did not increase proportionally.

You can see it inside companies. Teams can automate more processes than ever, but many still automate low-value tasks while leaving expensive, recurring bottlenecks untouched. They build dashboards

nobody checks, copilots nobody trusts, workflows that save five seconds but create ten minutes of cleanup, and AI features added because "we need an AI strategy." The tools are stronger. The choices are often weak.

You can see it in individual careers. A person who knows how to ship code used to have a clearer edge because fewer people could do it. Now that edge remains useful, but it is less exclusive. The stronger long-term edge is pairing the ability to make things with the ability to identify leverage. The people who stand out are not just productive. They are pointed in the right direction.

This is not a claim that implementation no longer matters. It still matters. A lot. Bad products still fail. Sloppy systems still break. Reliability, security, design, maintenance, and distribution still matter. Real engineering is still real. AI does not repeal the need for competence.

But competence is not the same thing as scarcity. Imagine a city where everyone suddenly gets access to power tools. Carpentry does not stop mattering. Skill still matters. But one source of advantage shifts. The person who can recognize which furniture people actually need, what fits their homes, what price they will pay, what materials hold up, and where demand is quietly rising, gains relative importance. Lower construction friction increases the value of choosing well.

That is what AI is doing to software. It is turning software from a relatively scarce act of production into a much more abundant medium of expression and experimentation. When a medium becomes easier to use, the bottleneck moves from production toward selection.

Think about writing. When publishing was physically expensive, gatekeeping centered on production and distribution. When publishing became cheap on the internet, anyone could post. That did not make attention worthless. It made selection more important. What to say, to whom, with what credibility, in what format, became more decisive.

The same pattern now applies to software: As making gets easier, deciding gets harder, because there are more possible things to make than

anyone can pursue. The menu expands faster than your time, attention, and energy. Opportunity cost becomes sharper. Every weekend spent shipping a clever but irrelevant tool is a weekend not spent solving a painful problem for a reachable customer.

Abundance raises the price of discernment so many capable people feel strangely stuck right now. They are not blocked by lack of tools. They are blocked by too many options. They can build almost anything small. They do not know which almost-anything matters.

That confusion creates two common traps:

The first trap is idea paralysis. Because the possibility space is so wide, builders drift. They bookmark problems, collect prompts, save threads, and sketch products in notes apps, but never commit. The freedom feels energizing at first, then dissolves into vagueness. If everything is buildable, nothing feels clearly urgent.

The second trap is compulsive shipping without discrimination. This looks productive. A builder launches one tool after another, each technically competent, each lightly promoted, each ignored. They tell themselves they are learning by doing, and they are, but often they are learning too slowly because they are practicing implementation more than opportunity selection. They get reps, but not the right reps.

In both cases, the missing skill is not effort. It is judgment under abundance.

Judgment sounds fuzzy, so let us make it concrete. In this book, judgment means the ability to repeatedly make better bets about what to build. It includes noticing real problems, estimating their seriousness, identifying who feels them strongly enough to care, understanding whether software is the right intervention, spotting timing advantages, matching ideas to your own access and strengths, filtering out seductive but weak opportunities, and choosing scopes that create learning quickly without collapsing into toy projects.

In an older world, the person who could implement a decent idea had a meaningful moat because fewer people could cross the gap from idea to product. In the new world, crossing that gap is still work, but it is no longer the rarest work. The rarest work is choosing ideas that survive contact with reality.

A useful way to see this is to separate three layers:

1. Can it be built?
2. Can it be used?
3. Can it matter?

For a long time, many projects died at layer one. Now far more of them clear layer one. Some even clear layer two, at least in a narrow sense. They function. A user can click around. A workflow completes.

But layer three remains brutal.

Does it matter enough to create repeated use? Does it save time, make money, reduce pain, reduce risk, improve status, increase control, or unlock something that was previously hard to do? Does it fit into existing behavior, budgets, incentives, and trust? Does someone care enough to switch, pay, adopt, recommend, or depend on it?

These are not coding questions. They are judgment questions. The market has always cared about them. What is different now is that they dominate earlier. Because prototypes are cheap, the world gives you less credit for making one. It wants to know whether the thing changes anything meaningful.

This has several consequences.

First, idea quality matters more than it used to relative to implementation effort. Not in the abstract sense of a magical startup idea, but in the practical sense of selecting a painful, recurring, reachable problem with a plausible path to adoption.

Second, distribution and insight become more important than raw building ability alone. If many people can build, then unfair advantages come from seeing what others do not see, reaching who others cannot reach, and understanding workflows others ignore.

Third, taste becomes economically relevant. By taste, I do not mean aesthetics alone. I mean the ability to distinguish signal from noise, core from feature creep, urgent pain from mild preference, durable need from temporary fascination.

Fourth, small advantages in framing compound. The builder who starts with a sharper problem statement makes better product choices, writes better copy, targets better users, and learns faster from feedback. Good judgment upstream simplifies execution downstream.

This is one reason experienced operators often outperform pure technologists in new tooling waves. They may be slower to adopt every shiny model release, but they have spent years developing intuitions about customer pain, budgets, incentives, buying behavior, and organizational dysfunction. They know that an ugly solution to a costly problem beats a beautiful solution to a trivial one.

The good news is that judgment is learnable. It can be improved through better filters, better observation, better idea sourcing, better evaluation, and better honesty about your own advantages and blind spots.

Before we get there, it helps to clear away one misunderstanding. The phrase "what to code" does not mean the coding part no longer matters, or that every future builder should stop learning technical skills. It means that technical skill is becoming downstream of a more important strategic choice.

The person who can both choose well and build well will remain dangerous. The person who can choose well but build imperfectly can now still get very far. The person who builds well but chooses poorly faces a harder future than before.