A SHORT BOOK

The Local Minimum

What failed experiments taught me about how we learn, build and engineer in Nigeria

Chima Emmanuel Akuegbo

Notes from a builder in Nigeria

Before we begin

THIS IS not a book about success. I don't have enough of it yet to write one, and I'm suspicious of people who write those too early. This is a book about things I tried that didn't work, and the uncomfortable patterns I noticed when I stopped blaming the circumstances and looked at how I, and many people like me, were taught to think.

Every chapter moves the same way. First, what was tried. Then why it failed. Then why the problem keeps existing, even though everyone can see it. And finally, what I think a better approach looks like. I'm not certain about all of it. Some of it I'll probably disagree with in five years. But I would rather write down what I think we got wrong now, while it still stings, than wait until it becomes a neat story.

A word on who "we" is. When I say we, I mean young Nigerian engineers and builders, people who learned to code from YouTube and borrowed laptops, who built their first circuit on an Arduino, who started companies before graduating because waiting felt like losing. I'm one of them. Nothing here is a judgement on anyone's intelligence. If anything, the point is the opposite: we are not short of ability. We are short of exposure, and exposure shapes what we think is possible.

1 The engineers who work at the bottom

What I tried. A few weeks ago I made my first serious contribution to a large open-source project: the runtime that lets AI models run on very small chips, the kind inside sensors and wearables. The part of the codebase I worked on belongs to engineers who build processors for a living. The issue looked small. A test in the build system compiled a program for a tiny processor, checked how many bytes it took up, and never actually ran it. My job was to make it run.

To do that, I had to set up a virtual chip on my machine, a simulator of a real processor, and get a bare program to boot on it. Not an app. Not a script. A program with no operating system underneath it, where you decide where memory begins and ends.

What happened. I got it working, but slowly, and with a lot of help. And the reviewers' comments were humbling in a quiet way. They weren't asking about clever features. They were asking about things I'd never been forced to care about: how small could the run be, what exactly did it depend on, could it be stripped to the minimum. Their instincts were built for a world where every byte and every cycle has a cost, because on a chip it does.

Why it felt like a failure. I wasn't bad at it. I was new to it in a way that someone my age in another country often isn't. The engineers I was talking to had grown up in an industry where people write and optimise code that sits right on the silicon. They have colleagues who design the chips, teams that test them, companies that sell them. That knowledge passes between people at lunch, in code reviews, in internships. It's in the air.

We never had that air. Nigeria doesn't design chips. We barely manufacture electronics. So the deepest layer of engineering, where software meets hardware and every abstraction is a choice someone made, is a layer most of us only read about.

Why the problem still exists. Because exposure is built by industry, and industry is built by exposure. Without companies that work at the bottom of the stack, there are no jobs that teach it. Without people who know it, no company will start here. It's a loop, and loops don't break on their own.

A better approach. We can't build a chip industry by wishing for it. But the loop has a crack in it now: open source. The engineers I learned from don't care where I live. The code is public, the simulators are free, and the reviewers will teach you if you show up with real work. The way into the bottom of the stack, for us, is to contribute to the projects that already live there, one small, boring, correct fix at a time. That's not a substitute for an industry. But it's how a few people get the exposure that an industry would one day need.

2 The abstraction ceiling

What we tried. Most of us who got into hardware got into it the same way: an Arduino or an ESP board, a breadboard, some jumper wires, a library from the internet. You write `digitalWrite(13, HIGH)` and an LED turns on. It feels like engineering. Then you build a smart home switch, a weather station, an automatic irrigation system for a final year project.

What failed. None of these projects failed in the sense of not working. They worked. That's the problem. They worked without us ever learning what was under them. The library handled the timing. The board handled the power. The bootloader handled the memory. We connected pre-made parts and called it building.

There's nothing wrong with abstraction. It's how all engineering scales. But there's a difference between choosing an abstraction because you understand what it hides, and living inside one because you've never seen past it. One is a tool. The other is a ceiling.

Why we got it wrong. Your solution set can only be as wide as your experience. If the only tools you've ever touched are high-level boards and drag-and-drop libraries, then every problem starts to look like something you can solve with a high-level board and a library. You don't consider writing your own driver, because you've never written one. You don't consider a cheaper chip with less memory, because you don't know how to fit your program into it. You don't consider a different architecture, because you've only seen one.

This is how a whole generation can be busy and still not inventive. We're very good at assembling. We are rarely forced to originate. And then we sit in panels and ask why we aren't innovative, as if innovation were a mood instead of the result of years of seeing more options than other people see.

Why it persists. Because the abstraction is cheap and the depth is expensive. An Arduino costs a few thousand naira and works in an afternoon. Learning how a microcontroller boots, how memory is laid out, how an interrupt really works, takes months and gives you nothing to show your lecturer at the end of the semester. Our schools reward the demo, not the understanding. Our clients reward the demo too.

A better approach. Keep the Arduino, but use it as a starting point, not a destination. After the LED blinks, remove the library and blink it by writing to the register yourself. Read the datasheet, even the parts you don't understand. Then take one thing you built and port it to a smaller, cheaper chip, with less memory than you think you need. Constraints are the best teachers we have access to. They force you to see what the abstraction was doing for you, and once you've seen it, you can never unsee it.

3 Stuck at a local minimum

What we tried. This one is about how we study, and I'll use a picture from my own field to explain it.

When you train a machine learning model, you choose a model first, a shape with millions of adjustable numbers called parameters. Then an algorithm called gradient descent nudges those numbers, step by step, to make the model's errors smaller. It works incredibly well. So well that it's easy to forget what it's actually doing. It's not searching for the best possible model. It's searching for the best numbers inside the model you already picked.

If you picked the wrong shape, or the wrong goal, or fed it the wrong data, gradient descent will still dutifully improve. It will find the best version of a bad idea. In the language of optimisation, it settles into a local minimum: a valley that looks like the bottom from inside, but isn't the lowest point on the map.

What failed. I think a lot of our education, and a lot of our engineering, behaves like gradient descent. We are very good at improving within a frame someone else chose. Read the past questions, learn the marking scheme, get the grade up. Take the framework, add the feature, ship it. We tune and tune. What we rarely do is step back and ask whether the frame itself is right: whether the question is the right question, whether the method is the right method, whether a completely different approach exists.

Why we got it wrong. Tuning works, and when something works it becomes the only thing we try. Gradient descent isn't the problem. Optimisation isn't the problem. The problem is that the big leaps in any field rarely come from tuning harder. They come from changing the model, changing the goal, or changing the data, the decisions made before the tuning starts. And those decisions require you to know what other models exist, what other goals are possible, what other data could be gathered. That knowledge comes from breadth and from foundations, which is exactly what our study habits skip.

Why it persists. Because the system measures how far down the valley you got, not which valley you chose. Exams, grades, client feedback and investor metrics are mostly local measures. They reward improvement inside the frame. Nobody hands out marks for stepping outside it, especially when stepping outside means your numbers get worse for a while before they get better.

A better approach. Learn like someone who has to choose the model, not just tune it. That means studying the reasons behind methods, not only the methods. When you learn a technique, also learn what came before it and why it lost. When you build something, write down two other ways you could have built it, and why you didn't. And every so often, deliberately take a step that makes your numbers worse, a new tool, a lower-level language, an unfamiliar field, because that's the only way out of a valley.

4 We skip the boring parts

What we tried. Speed. Everyone around me wants to move fast, and so do I. Learn the hot framework this month, build the demo next week, raise money next quarter. In many ways that hunger is our advantage.

What failed. Speed without foundations has a ceiling, and you hit it suddenly. The first time a bug doesn't match any Stack Overflow answer. The first time a model's performance drops and you can't tell whether the cause is the data, the code or the measurement. The first time a client asks you why something is slow and you have no way to find out, because you've never used a profiler, never read a stack trace end to end, never looked at memory.

The boring parts are things like linear algebra, probability, how compilers work, how operating systems schedule work, how networks really move packets, how to read a datasheet, how to write a test. Nobody posts them on social media. They don't make for a good demo day. But they are what lets you reason about a problem you've never seen before, instead of searching for someone who has.

Why it gets worse now. AI tools have made this gap wider, not narrower. If you have the foundations, AI is extraordinary: it's a tireless tutor, a fast typist, a second opinion. You can check what it gives you, catch its mistakes, and ask it better questions. If you don't have the foundations, AI becomes a crutch you can't evaluate. It writes code you can't debug and explanations you can't verify. You feel productive while your understanding stays exactly where it was.

I think this is why so many of us feel cognitively robbed in this era. Not because AI is taking something from us, but because it's exposing that we never built the thing it's supposed to amplify. A multiplier on zero is still zero.

Why it persists. Because foundations are slow, invisible and unrewarded in the short term. No one gets hired for understanding cache lines. People get hired for the portfolio. And when everyone around you is shipping, spending three months on fundamentals feels like falling behind.

A better approach. Pay a small, steady tax to the boring parts. An hour a day, every day, on one fundamental subject, with no deliverable attached. Use AI as a tutor for that hour, not a ghostwriter: ask it to explain, then close it and redo the work yourself. And when you hit a bug, resist the urge to paste it somewhere for an answer until you've tried to explain to yourself what is actually happening. That struggle is not wasted time. It is the foundation being poured.

5 The model that lost

This chapter is the most personal, because it's about something I built, believed in, and watched lose.

What I tried. In a short, intense stretch, I built a text-to-speech system for Nigerian languages. The goal was ambitious: one model that could speak Igbo, Hausa, Yoruba, Pidgin and English, and switch between them naturally in the middle of a sentence, the way we actually talk. We used only data we were legally allowed to use. We rented computing power from abroad by the minute, paid for out of a small prepaid credit with an expiry date. We went through several versions, each one better than the last. We published it openly.

What failed. It worked. The switching between languages was genuinely good. But when I compared it with a proprietary Nigerian system built by a better-funded team, theirs sounded more natural in Igbo and Hausa. Not a little more natural. Noticeably. I decided not to launch ours publicly with any fanfare, because I couldn't honestly say it was the best.

There were smaller failures along the way too. Our first attempt, a model built from scratch, produced nothing but babble. The cause turned out to be one setting in the part of the model that lines up text with sound, set to the wrong range. Later, when I tried to improve our speech recognition with synthetic speech generated by our own voice model, the new data made the system much better at understanding our synthetic voice, and no better at understanding real people. It had learned the voice, not the skill.

Why we got it wrong. The easy explanation is money and data, and it's partly true. The other team had more of both, and in speech, data quality is destiny. But I don't want to stop at the easy explanation, because it lets me off the hook.

The honest version is this: I optimised what I could measure. I had automatic scores for accuracy and pacing, so I pushed those numbers down, phase after phase. What I didn't have was a reliable way to measure what a native speaker hears as natural, so I under-invested in it. I tuned inside the frame my measurements gave me. The local minimum again. And the synthetic speech experiment was the same mistake in miniature: I created a shortcut that improved a number without improving the thing the number was meant to represent.

The babbling model taught me something else. A single misunderstood setting, buried in a part of the system I'd treated as a black box, can sink days of work. The less you understand the layers under you, the more fragile everything above them becomes.

Why it persists. Because in places like ours, the infrastructure for knowing whether you're right barely exists. There are few paid panels of native-speaker listeners, few trusted test sets, few calibrated tools built for our languages. So builders use the measurements available and quietly hope they mean what we want them to mean.

A better approach. Before building anything, decide how you'll know if it's actually good, and spend real money and time on that, even if it's just ten native speakers listening carefully and voting. Treat any shortcut that improves a number as suspicious until it improves the real outcome too. Understand at least one layer below the one you work on. And when you lose, don't hide behind the budget gap. Find the part of the loss that was yours, because that's the only part you can fix.

6 What work exposes

What we tried. Many of us go into our first jobs, internships or client contracts confident. We've built things. We've won hackathons. We have certificates and GitHub repositories. We think we're ready.

What failed. Then someone more experienced looks at our work, and in a single meeting, sometimes in a single question, they find the gap we didn't know was there. They ask how something behaves under load, and we don't know. They ask why we chose this design over another one, and we realise we never considered another one. They ask how we tested it, and the honest answer is that we ran it and it seemed to work.

I've had more of these moments than I'll write about here. Most of them are private, and they should stay that way. But the pattern across them was the same, and the pattern is what matters.

Why it stings. A good boss, client or senior engineer doesn't expose your weakness by attacking you. They expose it by asking the question your preparation never asked. School tested whether you could produce the answer. Work tests whether you understand the problem well enough to know which questions matter. Those are different skills, and only one of them was ever graded.

It's tempting to treat these moments as verdicts: I'm not good enough, I'm a fraud, I don't belong here. I've felt all of that. But I've come to think of them as diagnoses instead. The question you couldn't answer is a map of exactly where your foundation is thin. Nobody else will ever give you a map that precise, for free.

Why it persists. Because most of us never get feedback this direct until it's expensive. Our schools rarely do deep code reviews. Our side projects have no reviewers at all. So the first real critique comes when there is money, reputation or a job on the line, which makes it feel like an attack rather than a lesson.

A better approach. Go looking for those questions early, when they're cheap. Put your code where experienced people will review it: open source, technical communities, mentors who won't flatter you. After every hard review, write down the question that caught you, and spend the next week learning the answer properly, not just enough to survive the next meeting. And when you're the senior person someday, ask the hard question kindly. The person in front of you is probably where you once were.

7 Why fintech and edtech

What we noticed. If you look at Nigeria's most successful software products, a pattern jumps out. Most of them move money or teach people. Payments, transfers, lending, savings, exam preparation, online courses. It's fair to ask why almost nothing else breaks through.

The easy answer. It's tempting to say this is purely because of the engineering gap I've been describing: that we can only build what sits on top of other people's platforms, so we build apps, not deep technology. There's truth in that. But it's not the whole truth, and I want to be fair about it.

The fuller answer. Fintech and edtech also succeeded because the demand was enormous and urgent. Moving money was painful, so solving it paid. Exams decide futures, so people pay to prepare. Both could be built mostly in software, distributed through phones that people already had, and scaled without factories, laboratories or long hardware cycles. Investors understood them. Talent could be trained for them quickly. They were the shortest path between a real problem and a working business.

That's not shallow. Some of the engineering behind Nigerian payment systems is serious work: reliability, security and scale under difficult conditions. So the lesson isn't that fintech and edtech are bad. It's that they're where our solution set naturally points, given what we've been exposed to and what our environment rewards.

Why it's still a problem. Because a country whose engineers can only see one kind of opportunity will keep finding only that kind. Deep technology in health, energy, manufacturing, chips, materials and research tools needs people who are comfortable at lower layers, with longer timelines and harder measurements. We don't lack problems in those areas. We lack people who were trained to see them as engineering problems they could personally solve.

A better approach. Keep building the businesses that work, but stop assuming that's all we can build. If you're a student, pick one deep area and go further than your syllabus asks. If you're a founder, ask what layer your product depends on that nobody here understands, and consider learning it. If you're an investor or a school, fund and reward the slow work that doesn't demo well yet. The next category of Nigerian success won't come from doing the same thing faster. It will come from people who were exposed to something different.

8 A better approach

If I had to compress everything above into one idea, it would be this: we've been very good at tuning inside the frames we were given, and much less practised at choosing better frames. That's not a character flaw. It's what happens when your exposure is narrow, your feedback comes late, and your environment rewards the visible over the foundational.

So here's what I'm trying to do differently, and what I'd suggest to anyone who recognised themselves in these pages.

Go one layer down. Whatever you work on, learn the layer beneath it well enough to debug it. If you build web apps, learn how browsers and networks work. If you train models, learn how the hardware runs them. If you use boards, learn the chip. You don't have to live there. You just have to have visited.

Contribute where the experts already are. Open source is the closest thing we have to an industry we were never given. Pick a serious project, start with something small and boring, and let people who know more than you review your work. It's free mentorship, if you're humble enough to take it.

Pay the foundation tax daily. A little time, every day, on fundamentals with no deliverable attached. Use AI to explain, never to replace the struggle.

Measure the real thing. Before you build, decide how you'll know it's actually good, from the point of view of the person who will use it. Be suspicious of any shortcut that improves a number faster than it improves reality.

Search, don't just tune. Every so often, try a completely different approach, even if it looks worse at first. Write down the alternatives you rejected and why. The valley you're in is not the only one on the map.

Treat exposure as a diagnosis. When a boss, client or reviewer finds your gap, write it down and close it. That's the most useful feedback you'll ever get.

Ask for the shared things. Some of what holds us back no single person can fix: reliable power, local computing, data we're allowed to use, and test sets and expert panels that tell us honestly whether our work is good. Builders should ask for these loudly, from universities, government, donors and companies, because they're foundations everyone builds on.

THE SHORT VERSION

- Exposure, not ability, limits what we think we can build.
- Abstractions are tools; living inside one becomes a ceiling.
- We tune well inside frames we were given, and rarely question the frame.
- The boring parts are the foundation, and AI only multiplies what's already there.
- Measure what users actually experience, not what's easy to score.
- Hard questions from experienced people are diagnoses, not verdicts.
- Fintech and edtech won for real reasons, but they shouldn't be the only thing we can see.

I started writing this because a model I built lost, and a few engineers on the other side of the world showed me how much of engineering I'd never touched. Neither of those things feels good. But both of them showed me where the valley ends. The point isn't to feel bad about where we started. It's to stop mistaking the bottom of the valley we're in for the bottom of the map, and to start climbing out.