THE WEEKEND BUILDER

A Life in Code, Ownership, and the Habit of Starting Again

Bhadreshkumar Malankiya

This book is a personal account, written for the author's own record and for anyone building a career of their own.

CONTENTS

Why I'm Writing This Down

1. Roots — Growing Up in Surat

2. The Distinction Years — College and the Placement Season

3. Cracking the Code — DSA, Interviews, and the First Offer

4. Learning to Build — The HQ Infosystem Years

5. The Corona Advantage

6. Kali Linux Nights — A Cybersecurity Detour

7. The Jump — Becoming a Tech Lead at ExpressTech

8. Owning ExtendedForms.io

9. Building Quzo.ai From Nothing

10. Weekends That Mattered — HelpDesk AI and the Side-Project Habit

11. MB Systems — Founding, Failing, and What It Taught Me

12. The Craft — Charts, Maps, and Chasing New Tools

13. Leading Without a Title — Mentorship Lessons

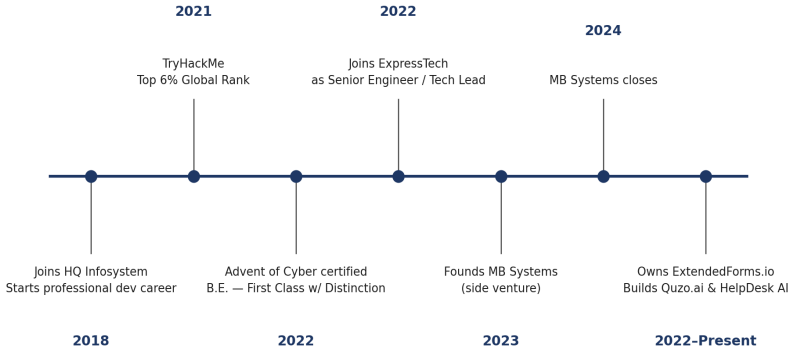
14. A Field Guide to Cracking DSA and Technical Interviews

15. A Field Guide to Landing Your First Job

16. What's Next

A LIFE IN TIMELINE

The Years, at a Glance



FOREWORD

Why I'm Writing This Down

I have spent most of my adult life building things that other people use without ever thinking about the person who built them. That is the nature of the work. A form loads in under a second, an exam runs without a single glitch, a support ticket gets answered before the customer has finished their coffee, and nobody claps. Nobody is supposed to. That is what "it works" means. The absence of complaint is the applause, and you learn, slowly, to hear it that way.

But somewhere between the first line of code I wrote as a student and the platforms I now own and run, alongside a full-time job that most people would already call a full plate, I picked up a habit of writing down what I was learning. Not for anyone else, just so I would not forget it. Notes in a phone, half-finished thoughts in a text file, the kind of thing most engineers accumulate and never look back at. This book is that habit, stretched out into something longer and more honest than a notes app entry ever gets to be.

I am writing this at a point where I have enough distance from most of these events to see the shape of them, but not so much distance that I have forgotten what they felt like while they were happening. I think that is the right distance for a book like this. Close enough to be true, far enough to be

useful. Write it too early and you are still inside the emotion of it, unable to separate what mattered from what merely felt urgent at the time. Write it too late and the specifics blur into a comfortable, softened story that flatters you more than it informs anyone else.

This is not a story about a straight line. There is no straight line in it, and I want to say that plainly before the first chapter even starts, because so much career writing pretends otherwise. There is a college placement season that went better than I expected, a first job that taught me more than any course did, a pandemic that could have stalled everything and instead became the opening I needed, a genuine detour into cybersecurity that never became my career but shaped how I think about every system I have built since, a technical leadership role I grew into rather than arrived in fully formed, two products I built and own outside of anyone else's permission, and one startup that did not survive a year. I am including that startup in as much detail as the ones that worked, because leaving it out would make this book a highlight reel instead of a record, and I did not sit down to write a highlight reel. A highlight reel is easy to write and useless to read. A record is harder to write and, I think, considerably more useful.

I also want to say something about the tone I am aiming for here, because it matters to how you should read what follows. I am not trying to convince you that any of this was

inevitable, or that I had some grand design from the beginning that everything else simply followed. Most of it did not feel like design while it was happening. It felt like a series of reasonable next steps, taken one at a time, by someone who was paying close attention and willing to put in the hours, without any confidence about how the whole thing would eventually add up. It only looks like a coherent arc from here, looking backward. I suspect that is true of almost every career that eventually makes sense on paper. The sense is retrospective. The living of it is much messier, much more uncertain, and much more interesting than the tidy version usually admits.

If you are reading this because you know me, you will recognize the outline, though I hope even you find a few details here you had not heard before, because a surprising amount of this book is material I have never actually said out loud to anyone, only written down for myself in scattered notes over the years. If you are reading this because you are early in your own career and looking for something more specific than generic advice, I have written two chapters near the end specifically for you: one on cracking technical interviews and data structures and algorithms, and one on landing your first job when you have no track record yet to point to. Those chapters are not theory. They are the exact things I wish someone had told me before my own first interview, stripped of the vague encouragement that fills

most advice on this topic and replaced with the specific, sometimes uncomfortable truths I actually needed to hear.

And if you are reading this because you are further along and wondering whether to start the side project, take the founder leap, or keep building on weekends when you already have a demanding job during the week, I hope the chapters on Quzo.ai, HelpDesk AI, and MB Systems in particular are useful to you. Two of those went well. One did not. All three taught me something I still use every week, in decisions far smaller and far larger than the ones described in this book. I have tried, throughout, to resist the temptation to sand down the rough edges of the story that did not work. I think the rough edges are exactly the part worth reading.

Let's start at the beginning.

CHAPTER ONE

Roots — Growing Up in Surat

"Surat never taught me that a good product sells itself. It taught me the opposite, every single day."

Surat is not a city that shows up often in the stories people tell about how someone became an engineer. It is known for diamonds and textiles, for a business instinct that runs through the culture like a second language, for a work ethic that treats a slow week as a personal failure rather than bad luck. I grew up inside that instinct before I understood it as an instinct at all. It was just how things were done around me: you build something, you sell it, you improve it, you build the next thing. Nobody sat me down and explained this as a philosophy. It was simply the air of the place, absorbed the way children absorb the rhythms of any household, any neighborhood, any city with a strong enough character of its own.

I did not come from a family of engineers or a school with a famous computer lab. What I had was ordinary access to a computer at a reasonable age, and an unreasonable amount of patience for figuring out why something was not working the way I expected it to. That patience is, in hindsight, most of what the job actually is. Not genius, not talent in the way people imagine it, just a willingness to stare at a broken thing for as long as it takes and refuse to accept

"I don't know" as a final answer. I did not think of this as a special quality at the time. I thought everyone worked this way, and it took years of watching other people give up on a problem far sooner than I would have to realize that this particular stubbornness was not universal, and that it was, in fact, one of the more valuable things about how my mind happened to be wired.

Gujarat's business culture gave me something else I did not fully appreciate until much later: a native comfort with the idea that a product needs a market, not just a clever build. Long before I ever wrote the word "SaaS," I understood, the way you understand things absorbed rather than taught, that nobody owes you attention just because you built something interesting. Somebody has to want it, and wanting it is not guaranteed by effort. That single idea has saved me from a lot of wasted work over the years, and it started forming long before I had any project worth applying it to. I remember, even as a student, being slightly suspicious of the idea that a good enough idea sells itself. Everything around me in Surat suggested the opposite: that even the best product in the world still needs someone tending to it, presenting it, standing behind it, and adjusting it based on what the market actually says back to you, rather than what you hoped it would say.

School gave me the fundamentals in the way school gives most people the fundamentals: unevenly, with a few subjects

that stuck and a few that did not, and a slowly growing sense that the subjects involving logic and structure were the ones where I did not have to force interest. Mathematics, and later the earliest programming exercises, had a property I found calming rather than stressful: there was a correct answer, and if you were patient and careful, you could find it. Most of life does not offer that. Code, at least in its cleaner moments, does. There is a specific comfort in a compiler telling you plainly that you are wrong, rather than the vaguer, softer feedback most of life gives you. I found that comfort early, and I never really lost my taste for it.

The earliest programming I did was small and unimpressive by any objective standard, the kind of thing every self-taught beginner produces: little scripts that did one thing, sometimes badly, and had to be rewritten three or four times before they did it correctly. What mattered was not the quality of that early work. What mattered was the specific feeling of making a computer do something it had not done a moment before, entirely because I had told it to. That feeling did not fade with more experience. If anything, it deepened, because the things I eventually learned to make computers do became more interesting, more consequential, and closer to problems that actually mattered to real people, but the underlying feeling of "I made this happen, on purpose, through reasoning" is exactly the same feeling I had the very first time a small script ran correctly.

By the time I was choosing a stream of study, the decision to move toward engineering, and specifically toward information technology, did not feel like a leap of faith. It felt like formalizing something that had already started. I was not choosing a career out of a guidebook. I was choosing to keep doing, with more structure and more depth, the thing I had already been doing in scattered, unsupervised hours: taking something apart to understand it, and then trying to build a version of it myself. I do not think this decision required any particular courage from me. It required almost no courage at all, because it was not really a decision to become someone new. It was a decision to keep being who I already was, with better tools and more structured guidance than I had managed to give myself.

Looking back, I think the most important thing about this period was not any specific skill I picked up. It was the formation of a default mode that I still operate in today: when something interests me, I do not wait for someone to assign it to me. I start. That single habit, more than any framework or language, is the actual foundation everything else in this book is built on. Every product described in the chapters ahead, every job change, every weekend project, every hard lesson from the one venture that failed, all of it traces back to that same simple default: see something worth doing, and begin, without waiting for permission or perfect conditions.

I want to close this chapter with a small honesty, because I think it is easy for an opening chapter like this to read as though the path was obvious from the start. It was not obvious. I did not know, as a student poking at a computer in Surat, that any of the specific things described later in this book would happen. I did not know I would eventually own two live products, lead a team, spend a serious stretch chasing vulnerabilities on Kali Linux, or found and lose a startup within the space of a few years. What I knew, if I am honest, was much smaller and much less certain: that I liked this, that I was willing to keep doing it even when it was frustrating, and that something in me trusted that persistence with a thing you genuinely enjoy tends to lead somewhere worthwhile, even when you cannot yet see the destination. That trust, more than any specific plan, is what actually carried me out of Surat's classrooms and into everything that came next.

CHAPTER TWO

The Distinction Years — College and the Placement Season

"A 9.0 CGPA didn't make me employable. Four straight years of refusing to cut corners did."

I completed my Bachelor of Engineering in Information Technology through Gujarat Technological University, studying at SSGEC, and finished with a CGPA of 9.0 out of 10.0 and a First Class with Distinction. I say this plainly and without false modesty, because I think false modesty about numbers like this does a disservice to the people reading this who are still in the middle of their own degree, wondering whether the grind is worth it. It is. Not because a number on a transcript changes who you are, but because the discipline required to earn that number is the same discipline you will need for everything that comes after it, and it is much cheaper to build that discipline as a student than to build it later under real professional pressure, with a paycheck and a manager's expectations attached to every mistake.

College, for me, was less about any single brilliant class and more about a compounding routine. I was not the student who crammed the night before and pulled off a miracle. I was the student who treated consistency as a competitive advantage, because I had noticed early that most people do not sustain effort for very long, and the ones who

do tend to separate from the pack quietly, without any dramatic moment where the separation becomes visible. It just accumulates, week after unremarkable week, until one day the gap between where you are and where a less consistent peer is has become large enough that it finally looks, from the outside, like talent. It was never talent. It was arithmetic: small, repeated deposits of effort, compounding quietly while nobody was measuring them.

I want to describe what that routine actually looked like, because "consistency" is a word that gets used so often in advice writing that it has lost most of its texture. For me it meant treating coursework not as a series of deadlines to survive but as a series of concepts to actually understand, which is a slower way to study in the short term and a dramatically faster way to study across an entire degree, because understanding compounds and memorization does not. A concept properly understood in the second semester makes the sixth semester's harder material easier to absorb. A concept merely memorized for an exam and then discarded leaves nothing behind for the sixth semester to build on. I chose, almost every time the choice presented itself, the slower and more thorough path, even when a faster, shallower path would have produced an identical grade on that particular exam.

The placement season is where that accumulation showed its return. I received the highest package offered to

any student in my college that year. I want to be careful about how I describe that, because it is easy for a fact like this to sound like bragging when what I actually want to communicate is closer to relief and validation. Placement season, for anyone who has been through it, is a genuinely stressful stretch. You are being evaluated on a compressed timeline by people who do not know you, based on a combination of academic record, technical rounds, and a handful of minutes of conversation that are supposed to somehow represent four years of who you are. There is an almost unbearable mismatch between how long it takes to become a strong candidate and how briefly you are actually observed being one. Getting the outcome I got did not feel like winning a competition against my classmates, most of whom I genuinely liked and respected. It felt like the first piece of external evidence that the way I had chosen to spend my college years actually mapped to something the market valued, independent of anyone else's outcome.

I remember the specific texture of the weeks leading up to that season more clearly than I remember the offer letter itself. The uncertainty of not knowing which companies would visit campus that year, the comparison, mostly unspoken but always present, between yourself and your closest peers, the way a single mock interview that went badly could unsettle your confidence for days even when you knew intellectually that one bad mock interview means very

little. None of that stress disappeared because I had prepared well. Preparation does not remove the stress of an uncertain outcome. What it does, and this is the distinction I think matters most, is change what the stress is actually about. An unprepared candidate's stress is largely about whether they know enough. A prepared candidate's stress is mostly about the ordinary unpredictability of any evaluation involving other human beings, which is a lighter and more survivable kind of stress to carry into an interview room.

That distinction, both the academic one and the placement outcome, mattered less to me for what it said about the past and more for what it gave me permission to believe about the future. I walked out of college believing that sustained, unglamorous effort compounds, and that belief has never once let me down, even in the chapters of this book where the outcome itself was a failure. The belief is not "you will always succeed." The belief is "the effort was never wasted, regardless of the outcome," and that is a much sturdier thing to build a career on, because it survives contact with the inevitable setbacks that any long career eventually includes. A belief built on outcomes collapses the first time an outcome goes badly. A belief built on the durability of honest effort survives that collapse and keeps you moving anyway.

I also want to say something here about grades that I think gets lost in a lot of career advice aimed at students,

which tends to swing hard in one of two unhelpful directions: either treating grades as the entire game, or dismissing them entirely as irrelevant once you have a job. Neither is quite right. A strong academic record is not, by itself, evidence that you will be a strong engineer. I have met excellent engineers with unremarkable transcripts and unremarkable engineers with excellent ones. What a strong, honestly earned academic record actually demonstrates, and what I think genuinely transfers, is your relationship to difficulty over a sustained period: whether you tend to meet a hard requirement with real effort or with the search for a shortcut around it. That relationship to difficulty, more than the specific content of any course, is what placement interviewers are actually trying to sense underneath the transcript, whether they would describe it that way or not.

If there is a single piece of advice from this chapter, it is this: do not wait for placement season to start caring about how you spend your time. By the time placement season arrives, the work is already done, one way or the other. What you are actually being evaluated on in those final weeks is a distillation of everything that came before them. Build the distillation early, and the season itself becomes far less frightening than it looks from the outside. It stops being a test you might fail and starts being a formality that confirms something you already know to be true about yourself, which is a completely different experience to walk into, even

though from the outside it looks like the exact same interview room, the exact same panel, the exact same clock on the wall.

CHAPTER THREE

Cracking the Code — DSA, Interviews, and the First Offer

“I didn't beat my first interview with talent. I beat it with the same handful of problems, solved two hundred times, until none of them looked new anymore.”

Everyone who has been through a technical hiring process in software knows the shorthand: DSA. Data structures and algorithms. Arrays, linked lists, trees, graphs, recursion, dynamic programming, the whole vocabulary that shows up on whiteboards and shared coding screens around the world, regardless of what the actual job will look like once you have it. I want to be honest about something most people are not honest about: a large percentage of the DSA you grind through for interviews will not appear in your daily work in anything like the form you studied it. And it still matters enormously that you study it, because it is not really testing whether you will invert a binary tree on the job. It is testing whether you can hold a problem in your head, break it into smaller correct pieces, and reason clearly about tradeoffs under mild pressure while someone is watching. That skill generalizes to almost everything a working engineer actually does, from debugging a production incident at two in the morning to designing a new feature's data model before a single line of code exists.

My own approach to this period of preparation was not exotic. It was repetition with attention, which is a less exciting phrase than most study advice but a more accurate one. I did not look for a shortcut through data structures and algorithms because I do not believe there is one that survives contact with an actual interview. What I looked for instead was pattern recognition: after enough problems, you start to notice that a huge share of what looks like a novel challenge is actually a familiar shape wearing a different costume. A sliding window problem dressed up as a string question. A graph traversal dressed up as a scheduling question. A dynamic programming problem dressed up as a question about counting paths, or maximizing profit, or minimizing cost, phrasing that changes constantly while the underlying recurrence relation stays recognizably the same. Recognizing the shape underneath the costume is most of the battle, and that recognition only comes from volume, deliberately worked through rather than passively read.

I want to spend a moment on the specific categories, because I think a slightly more concrete breakdown is more useful here than another paragraph about the importance of practice in the abstract. Arrays and strings are the foundation almost everything else builds on, and the two-pointer technique and the sliding window pattern, once genuinely internalized, will quietly solve a surprising fraction of the problems you will ever encounter, far beyond what

their apparent simplicity suggests. Trees and graphs are where most candidates start to feel real friction, because unlike arrays, they require you to hold a mental model of a structure that does not sit neatly in a single line, and breadth-first and depth-first traversal, along with a comfortable intuition for when each is the right tool, are worth practicing until they stop requiring conscious thought. Dynamic programming is the topic most candidates fear most, usually because it is taught badly, as a bag of memorized recurrence formulas rather than as what it actually is: a systematic way of avoiding repeated work by recognizing that a big problem's answer is built out of smaller versions of the exact same problem. Once that reframing clicks, a large share of the fear around dynamic programming clicks away with it.

The interview itself, when it finally arrived, was less about knowledge and more about composure. I had prepared enough that the technical content was not the primary source of stress. What I had to manage instead was the very human experience of being watched while thinking, which is an unnatural way to think and takes its own kind of practice. I made a habit, well before my real interviews, of talking through problems out loud to an empty room, narrating my reasoning the way I would need to narrate it to an interviewer. It felt strange the first several times, genuinely strange, the kind of self-conscious feeling most people

associate with rehearsing a speech alone in front of a mirror. By the time I was doing it in front of an actual person, it no longer felt strange, and that was the entire point. The strangeness had already been spent, safely, in an empty room, where nothing was at stake, rather than saved up and experienced for the first time when something actually was.

I landed my first offer at HQ Infosystem, and I remember the specific relief of that moment more clearly than almost any technical detail from the interview itself. It was not just an offer. It was the first piece of proof, outside of a grade or a professor's opinion, that the way I had spent years preparing produced something the world was willing to pay for. There is a particular kind of validation in a job offer that a grade, however good, cannot fully provide, because a grade is awarded by someone whose job is partly to teach you, and there is an unavoidable generosity built into that relationship. An employer offering you a position has no such generosity built in. They are making a bet with their own money, and the offer is evidence that a stranger, with nothing to gain from being kind to you, looked at what you could do and decided it was worth paying for.

If I were speaking directly to someone preparing for their first technical interview today, here is what I would want them to actually hear, past the generic advice that fills most articles on this topic.

Depth beats breadth in the early stages. It is better to deeply understand arrays, strings, recursion, and basic graph and tree traversal than to have a shallow, panicked familiarity with every advanced topic that exists. Interviewers can tell the difference between someone who has memorized a solution and someone who understands why the solution works, and the second kind of candidate survives follow-up questions the first kind cannot. A follow-up question is not an attack. It is usually the interviewer giving you a chance to demonstrate the depth a single problem cannot show on its own, and candidates who have only memorized a pattern tend to freeze exactly at the moment they are being offered that chance.

Explain your thinking before you write a single line of code. The code is almost the least important part of a technical interview. The reasoning that leads to the code is what is actually being evaluated, and a candidate who writes correct code silently, without narrating the path that led there, has given the interviewer almost nothing to evaluate except the final output, which tells them far less about how you would actually perform on a real, ambiguous problem where the correct approach is not obvious from the start.

Get comfortable being wrong out loud. Nobody solves every problem cleanly on the first attempt in a real interview. What separates a strong candidate from a weak one is not the absence of a wrong turn, it is the visible, calm process of

noticing the wrong turn and correcting it without falling apart. I have watched candidates, in the years since I have been on the other side of interviews as well, lose far more credibility by panicking at a small mistake than they ever lost from the mistake itself.

Treat every mock interview, every practice problem talked through out loud, every rejection along the way, as data rather than judgment. None of it is a verdict on you. All of it is information you can use to prepare better for the next attempt. I failed interviews. I want to say that plainly, because it is easy, from the outside, to assume a strong outcome means a clean record, and that assumption discourages people who are currently in the middle of a string of rejections from believing the pattern can still turn around. It can. Mine did.

The offer, when it comes, will not feel like the end of a test. It will feel like the beginning of a much longer and more interesting one. That is exactly how it should feel, and if it does not feel that way, if it feels instead like a finish line you have crossed and can now rest behind, that is usually a sign you have misunderstood what the interview was actually preparing you for.

CHAPTER FOUR

Learning to Build — The HQ Infosystem Years

"Fifteen client projects taught me the same lesson fifteen times: edtech, logistics, and fintech are the same problem, wearing three different costumes."

I spent from 2018 to 2022 at HQ Infosystem, starting as a Full Stack Developer and growing into a Senior Full Stack Developer over that stretch. If the college years were about proving I could learn, the HQ Infosystem years were about proving I could ship, and those are not the same skill. A huge number of people who are excellent at learning in a structured academic setting struggle badly with the much messier discipline of shipping something real, on a deadline, to a client who does not care how elegant your code is if the feature does not work the way they expected. College rewards understanding. A job rewards understanding that arrives on time, survives contact with a real user, and keeps working after you have moved on to the next task.

Over those four years I delivered more than fifteen full-stack applications across a genuinely wide spread of industries: edtech, logistics, fintech, e-commerce. I want to underline how valuable that spread turned out to be, because at the time it did not feel like a strategy, it felt like whatever the business needed that quarter. There was no grand plan

behind the variety. A client in logistics needed something, and then a different client in fintech needed something else, and I simply did the work in front of me as well as I could, one project at a time, without any particular sense that the variety itself was building something valuable. In retrospect, that breadth across industries taught me something a single deep specialization could not have: almost every domain, once you strip away the industry-specific vocabulary, is solving a smaller number of underlying problems than it appears to be. Edtech and logistics look nothing alike from the outside. From the inside, at the level of "how do we move data reliably, authenticate the right people, and present the right information at the right time," they rhyme constantly.

This period is also where the integrations that now sit comfortably in my everyday toolkit first became real to me: Stripe, Razorpay, and Twilio for payments and communication, Mapbox for anything involving location, and the Binance API for an early taste of what working with financial market data actually demands in terms of precision and error handling. I remember the first time I integrated a payment provider properly, rather than just following a tutorial's happy path, and realizing how much of the real work in a payment integration has almost nothing to do with the successful transaction. The successful transaction is the easy ten percent. The other ninety percent is every way the transaction can fail, partially fail, time out, get retried

incorrectly and charged twice, or succeed on the provider's side while your own system never receives confirmation. Building genuine comfort with that ninety percent, rather than the easy demo-ready ten percent, was one of the more important technical educations of this entire period, and it is not the kind of thing any course teaches well, because courses are built around the happy path almost by necessity.

I built REST APIs with Node.js, Express, and Laravel that handled fifty thousand or more daily requests, which was the first time scale stopped being an abstract concept from a systems design textbook and became a real constraint I had to design around. There is a specific moment, familiar to most engineers who eventually work at any meaningful scale, where a query that was instant against a small test dataset suddenly takes several seconds against real production data, and you are forced to actually understand indexing, query planning, and caching rather than simply reading about them. I hit that moment more than once during these years, and each time it happened, it left behind a permanent, hard-won intuition that no amount of reading ever fully replicates.

I built Stripe Connect multi-vendor payout systems, which forced me to think carefully about money moving between parties, about the kind of error that cannot simply be logged and shrugged off because it involves someone's actual income. Working on systems that move real money

changes how you write code in a way that is hard to describe to someone who has only ever worked on systems where a bug means an inconvenient page reload rather than a wrong number in someone's bank account. Every edge case stops being theoretical. A payout calculated incorrectly is not an abstract test failure, it is a vendor who did not get paid the right amount for work they already did, and that reframing made me a more careful, more defensively minded engineer across every subsequent project, well beyond the specific domain of payments.

I cut API response times by 45 percent through caching and query optimization, and held Jest coverage above 85 percent on the modules that mattered most, both of which taught me that performance and reliability are not features you add at the end, they are habits you either build in from the start or spend far more time retrofitting later. Retrofitting performance into a system that was never designed with performance in mind is genuinely one of the more frustrating kinds of engineering work, because you are constantly fighting decisions made earlier, sometimes by yourself, sometimes by someone who has already moved on, decisions that made sense at the time and now stand directly in the way of the fix you need. I learned, during this period, to think about performance and testability at design time rather than as a cleanup exercise scheduled for later, because

"later" in a real, ongoing product almost never arrives with the free time it was promised in.

None of this happened because I was assigned a growth plan that walked me through it in order. It happened because the work itself kept presenting harder versions of the same fundamental questions, and I kept choosing to meet them rather than route around them. There is a specific moment, common to most engineers who eventually grow past their first role, where you stop asking "how do I get this feature working" and start asking "how will this decision look in six months when the data has grown ten times over." I do not remember the exact week that shift happened for me during these years. I remember that it did happen, and that once it happened, I could not go back to thinking about code any other way. It is a little like learning to see a second layer underneath the first: the feature you are building is still the feature, but underneath it now sits a constant, quiet awareness of the system the feature is being added to, and what that system will need to tolerate as it grows.

I also want to say something about the human side of this period, because it is easy for a chapter framed around technical growth to leave out the fact that almost every one of these fifteen-plus applications involved real clients with real expectations, real deadlines, and real frustration when something did not go the way they had pictured it. Learning to manage that relationship, to communicate honestly about

a delay before it became a crisis, to push back on an unreasonable request without damaging the relationship, to say plainly when something would take longer than hoped rather than promising a date I privately doubted, was its own curriculum, running in parallel with the purely technical one. Nobody graded that curriculum. It graded itself, in the form of which clients came back with more work and which relationships quietly ended after a single strained project.

By the end of this period I was no longer the same developer who had walked in as a fresh graduate. I had built enterprise backend systems handling millions of transactions, and more importantly, I had built the instinct for what "handling millions of transactions" actually requires in practice, which no amount of reading about it in advance can substitute for. That instinct is the real inheritance HQ Infosystem left me with, and it is the foundation everything I built afterward, including the products I now own outright, was constructed on top of. I do not think I could have built ExtendedForms.io or Quzo.ai to the scale they eventually reached without these four years underneath them, quietly doing the unglamorous work of turning theoretical knowledge into practical, load-tested instinct.

CHAPTER FIVE

The Corona Advantage

"A crisis doesn't ask if you're ready. It only asks what you do next."

I want to write carefully about this chapter, because the period the world now shorthands as "COVID" or "corona" was, for an enormous number of people, a period of loss, fear, and disruption that had nothing redemptive about it at all. I do not want anything in this chapter to read as if I am minimizing that. Families lost people. Businesses that took years to build disappeared in months. Entire industries were reshaped in ways that are still being felt. What I can speak to honestly is my own experience of that period as a working engineer in the middle of building a career, and in that specific, narrow sense, it became one of the more important turning points I have had. I want to hold both of those truths at once throughout this chapter, without letting the second one drown out the first.

The honest version of the story is this: the job market slowed down dramatically during that stretch, the way it slowed down for almost everyone, everywhere. Hiring froze in places, projects got paused, and a lot of people I knew found themselves suddenly with far more uncertainty about their professional future than they had a few months earlier. Conversations that had once been about which offer to

accept became conversations about whether an offer would exist at all. The easy response to that kind of environment is to freeze along with it, to treat the slowdown as a signal to wait quietly until conditions improve, to conserve energy the way you might during any period that feels dangerous and uncertain. I made a different choice, and I think it is the choice that mattered most during that window.

I treated the slowdown as an unusual, temporary gift of time and attention that I would not normally have. With fewer external distractions competing for hours in the day, fewer commutes, fewer of the small social obligations that quietly consume an evening without ever feeling like a decision, I turned that time toward deepening exactly the kind of skills that this book keeps circling back to: shipping real things, learning new tools quickly, and building a portfolio of proof rather than waiting for someone else to hand me an opportunity to prove myself. Where the market was slow, I used the slowness as room to get better rather than as an excuse to get quieter. I did not have a grand plan for this at the outset. I simply noticed, fairly early into the disruption, that the hours were suddenly there in a way they had not been before, and it felt wasteful, almost disrespectful to the difficulty everyone was going through, to let that unusual quiet pass without using some of it deliberately.

I want to be specific about what "using the time" actually looked like, because vague encouragement to "upskill during

a downturn" is not much more useful than no advice at all. It meant going deeper into frameworks and tools I had only used at a surface level under normal deadline pressure, the kind of deep, unhurried exploration that a live client project rarely allows time for. It meant building small things purely to understand them, with no deadline and no client waiting, which is a completely different kind of learning than building something under pressure to ship. It meant reading documentation cover to cover instead of skimming it for the one method call I urgently needed that week. None of this produced anything dramatic in the moment. All of it produced a depth of understanding that later showed up, quietly, in how quickly I could move once real projects picked back up, and it is also, as the next chapter describes, the same appetite that pulled me sideways for a while into a completely different discipline before pulling me back.

The compounding effect of that choice did not show up immediately. It showed up later, when the market began to recover and the difference between people who had spent the slow period building and people who had spent it waiting became visible in a very concrete way: some of us were simply further along than we had been going into the crisis, and others were exactly where they had started. I do not say this to claim any special virtue. Plenty of people who spent that period focused entirely on simply surviving it, emotionally and financially, had every reason to do so, and I

do not think less of anyone who used that time differently than I did. I say it because I think it is one of the more transferable lessons in this entire book, applicable to far more than a global pandemic: any period that looks like a career setback, if you are still employed and still have your hands and your evenings, is also a period nobody is watching closely. That lack of scrutiny is not a curse. It is room to work on the things you would otherwise never have time to prioritize, without anyone's expectations attached to the outcome.

There is also a quieter lesson in this chapter about the relationship between fear and action that I did not fully understand until much later. Fear, during that period, was everywhere, reasonable, and largely outside anyone's control. What was within my control was much smaller and much more specific: whether I spent the next hour building something or scrolling anxiously through the same uncertain headlines everyone else was scrolling through. Choosing the former, repeatedly, over many months, did not make the broader fear disappear. It simply gave me something productive to place next to the fear, rather than instead of it, and I think that coexistence, doing useful work while still being afraid of the same uncertain future everyone else was afraid of, is a more honest description of resilience than the version of resilience that pretends fear was never present at all.

I emerged from that period with more depth than I went in with, and with a durable belief that slow markets are not the enemy of a career, only the enemy of a passive one. That belief has stayed with me through every downturn, hesitation, and uncertain stretch since, including some of the ones described later in this book, particularly the period around MB Systems. A market being difficult has never, in my experience, been a good reason to stop building. It has only ever been a good reason to be quieter and more deliberate about what I build, and to trust that the work compounds even when nobody is applauding it in real time. When the founder chapter of this book arrives, and the venture inside it eventually does not survive, this same belief is part of what let me treat that outcome as a lesson to carry forward rather than a verdict to be crushed by. I had already lived through one period that felt, from the inside, like it could end a career before it properly started, and I had already learned, the hard way, that it does not have to.

CHAPTER SIX

Kali Linux Nights — A Cybersecurity Detour

"I fell in love with breaking things a full year before it occurred to me that someone might pay me to do it."

Somewhere in the middle of my time at HQ Infosystem, roughly across 2021 and into 2022, I went through a period that never shows up on my resume in any clean, official way, but that shaped how I think about every system I have built since. I fell, almost by accident, into a serious interest in cybersecurity. Not a course. Not an assignment. A genuine, late-night, self-directed pull toward understanding how things break, running alongside a full-time development job that had no formal connection to security work at all.

It started the way most of these detours start: curiosity about the other side of the systems I was already building. I had spent years by that point making things work. At some point I became almost equally interested in the opposite question, how things get made to fail, and once that question took hold, it did not let go easily. I installed Kali Linux and started working through its tooling seriously, not as a novelty but as an actual practice, learning to think the way an attacker thinks: where does this system trust input it should not trust, where does this authentication flow have a gap, where does this API leak more than it should to

someone patient enough to look. I spent real hours on vulnerability testing, methodically, the way I had already learned to approach any hard technical problem: not by looking for a shortcut, but by building genuine, repeated familiarity with the terrain until the patterns started to reveal themselves.

TryHackMe became a central part of this period, and I am proud of what came out of it: I reached a Top 6% Global Rank on the platform in 2021, working through their structured rooms and challenges the same way I had once worked through data structures problems, pattern by pattern, until the underlying logic of common vulnerability classes stopped feeling mysterious and started feeling recognizable. In 2022, I completed and was certified through their Advent of Cyber event, an annual, calendar-style set of hands-on security challenges that walks through a wide range of real-world attack and defense scenarios one day at a time. Working through it felt, in a strange way, like a security-flavored echo of the DSA grinding I had already done years earlier for interviews: the same discipline of repetition with attention, applied to an entirely different domain.

Alongside the hands-on practice, I read seriously into the discipline, the kind of books and material that take you from "I can follow a tutorial" to "I understand why this class of vulnerability exists at all, independent of any specific tool." I

wanted to understand the theory underneath the practice, not just accumulate a collection of memorized exploit techniques that would go stale the moment the tooling changed. That distinction, understanding the underlying principle versus memorizing a specific technique, is one I had already learned to value in software development, and it transferred directly into how I approached security research. A specific exploit for a specific outdated library will stop working the moment that library is patched. An understanding of why that class of vulnerability existed in the first place keeps paying off indefinitely, against systems nobody has written a specific writeup for yet.

I want to be honest about something that I think a lot of career narratives smooth over: I never fully crossed over into cybersecurity or bug bounty hunting as an actual career path, even though I genuinely loved the work and was getting good enough results to make that a plausible direction. My career as a software developer was already underway by this point, several years into HQ Infosystem, with real momentum and real responsibilities attached to it. And my home situation at the time required something specific from me: steady, continuously growing income, arriving on a dependable schedule, not the far more uncertain and front-loaded-with-risk economics that bug bounty work and independent security research realistically involve, especially before you have built a reputation

substantial enough to make that income reliable. Bug bounty payouts are genuinely unpredictable, arriving in bursts rather than a steady rhythm, and building the kind of reputation that makes them dependable takes years most people cannot always afford to spend without other income underneath them. I could not, in good conscience, treat that uncertain path as my primary one while people at home were counting on the more predictable trajectory I had already started building in software development. So I made a decision, not without some real reluctance, to keep cybersecurity as a serious, skilled interest rather than a career pivot, and to keep growing rapidly and deliberately in development instead, since that was the path that could actually support what needed supporting.

I do not think of that decision as a loss, even though I want to be honest that it did involve giving something up. What I gained instead was a genuinely unusual advantage as a developer: I now think about the systems I build the way an attacker would look at them, by default, without needing to consciously switch into a different mode. When I built HelpDesk AI's mailbox credential handling with AES-256 encryption and a carefully isolated WatcherManager, that design did not come purely from software engineering best practice. It came from months of Kali Linux nights spent thinking specifically about how credential storage gets exploited when it is built carelessly. When I think about

JWT-based tenant isolation across WebStack and HelpDesk AI, or when I approach OWASP-style security testing on any project I touch, I am not applying a checklist I memorized from a compliance document. I am applying an instinct built during a period when finding the gap in someone else's system was the entire goal, for its own sake, for a solid year and a half, independent of any deadline or client.

Looking back, I think this detour is one of the more important chapters in this book precisely because it did not become the headline career. It became something quieter and, in its own way, more valuable: a permanent second lens layered underneath every architecture decision I have made since, in every product described in the chapters around this one. I build things now, and somewhere in the process, a version of myself that once spent nights on Kali Linux, working through TryHackMe rooms for no reason except that I wanted to understand how systems actually fail, is quietly checking the same build for the exact gaps I once spent a year and a half learning to look for. That is not a bad outcome for a path I ultimately chose not to walk all the way down. In some ways, it might be a better one than walking it all the way down would have been.

CHAPTER SEVEN

The Jump — Becoming a Tech Lead at ExpressTech

"Nobody promoted me into leadership. I just started making the decisions nobody else wanted to own, and the title eventually agreed with me."

In 2022 I moved to ExpressTech Systems, and I am still there today, now as a Senior Software Engineer and Technical Lead. This is the role that has given me the most sustained scope of any point in my career, and it is also the role that taught me the clearest lesson about the difference between being a strong individual contributor and being someone a team can actually depend on for direction. The move itself did not feel dramatic at the time. It felt like a reasonable next step after four years of accumulated experience at HQ Infosystem, and after a year and a half spent, in parallel, learning to see systems the way an attacker sees them. It was only well into the new role that I understood how different the actual day-to-day texture of the work would become, and how much that security-minded lens from the previous chapter would end up shaping the architecture decisions I was now responsible for.

The shift from developer to technical lead is not primarily a shift in the difficulty of the code you write. It is a

shift in the number of decisions other people are waiting on you to make, and in how far the consequences of a wrong decision travel before someone notices. As an individual contributor, a bad choice mostly costs you time. As a technical lead, a bad choice can cost your whole team time, and if it reaches production, it can cost the business real money and real trust with customers. That widening blast radius changes how you think before you act, and it took me real, deliberate effort to build the habit of pausing at exactly the moments when, as a junior developer, I would have simply started typing. I noticed, fairly early in this role, that the engineers who struggle most with the transition to leadership are often the ones who were fastest and most confident as individual contributors, precisely because that speed and confidence had never been trained to pause and consider a wider blast radius. I had to actively unlearn some of my own instinctive speed in order to build the more deliberate judgment the new scope required.

Over the years in this role I have mentored more than five engineers directly, which is a number I am genuinely proud of in a quieter way than any performance metric, because mentoring well requires a kind of patience that does not come naturally to someone who is used to solving problems fast and alone. I helped establish testing discipline across the team, pushing Jest coverage above ninety percent and layering in Cypress end-to-end tests, not because

coverage numbers are inherently meaningful, but because the process of getting a team to actually care about coverage numbers is really a process of getting a team to care about the person who will maintain this code after you have moved on to something else. That reduced our bug escape rate by 75 percent, and the number that actually matters behind that statistic is the number of Friday afternoons nobody had to spend firefighting a preventable production issue instead of going home on time. I think about that specific tradeoff often: a testing culture is, at its heart, a decision to spend a little more time now so that a future version of the team, possibly not even including the people who wrote the original code, inherits fewer avoidable emergencies.

Technically, this role has also been where I did some of my most demanding infrastructure work: scaling backend systems to support 100,000 concurrent users at 99.8 percent uptime, architecting server-rendered applications with Next.js that cut page load times by 60 percent, and building a CDN strategy that delivers sub-100 millisecond response times globally. Behind each of those numbers is a long list of smaller decisions, most of them unglamorous, about caching strategy, query optimization, and load balancing under real, unpredictable traffic rather than the polite, predictable traffic of a staging environment. Scaling to 100,000 concurrent users did not happen through one heroic redesign. It happened through dozens of smaller decisions,

made over an extended period, each one closing a gap that would have become the actual bottleneck once the previous one was solved. Systems at scale rarely fail at their weakest theoretical point. They fail at whichever point happens to be weakest once every stronger point has already been reinforced, and chasing that constantly moving target has been one of the more genuinely interesting parts of this entire role.

But the part of this chapter I think is genuinely worth passing on is less technical. Becoming a technical lead taught me that leadership in engineering is mostly the discipline of making the invisible work visible enough that your team can trust it exists. Nobody can see the architecture decision that prevented a scaling problem eighteen months before it would have happened. Nobody can see the mentoring conversation that helped a junior engineer stop being afraid to ask a question in a stand-up. The job, in large part, is doing that invisible work anyway, consistently, without needing it to be visible to feel worth doing. That is a very different motivation than the one that got me through DSA prep or my first job interview, and building it took time. It is, I think, the actual definition of growing from a developer into a lead: the reward stops being "did I solve the problem" and becomes "did the team get stronger because I was here."

I also learned, during this period, a specific lesson about the limits of my own certainty that I did not expect

leadership to teach me. As an individual contributor, being wrong about a technical decision is a private, contained event, corrected quietly, often before anyone else even notices the initial mistake. As a lead, being visibly, confidently wrong in front of a team carries a different cost, not just to the immediate decision but to how much the team trusts your future judgment. This did not make me more timid about making decisions. It made me more careful about the specific confidence with which I stated them, learning to distinguish out loud between "I am certain about this" and "this is my best current read, and I want us to check it against reality quickly rather than commit to it irreversibly." That distinction, communicated honestly and often, turned out to build more trust over time than false certainty ever did, even when the false certainty would have sounded more impressive in the moment.

There is one more thing about this role I want to record honestly, because it connects directly to a later chapter of this book. Being trusted with this much technical and organizational scope inside an established company is part of what gave me the confidence, a year or so later, to believe I could found and run something of my own. I do not think that confidence was misplaced, even though the venture it eventually led to did not survive. What ExpressTech taught me about scope, about being the person decisions ultimately rest with, translated directly into how I approached MB

Systems, even in the areas where that translation was incomplete and I still had real, painful lessons left to learn on my own.

CHAPTER EIGHT

Owning ExtendedForms.io

“401,000 users don't know or care that I own this product. They only notice when the page loads in 1.2 seconds instead of 3.2.”

At ExpressTech, I am the sole technical owner of ExtendedForms.io, a platform that extends Google Forms with analytics and AI capabilities inside the Google Workspace ecosystem. I want to spend a full chapter on this product specifically, because "technical owner" is a phrase that can sound like a job description when what it actually is, day to day, is a completely different relationship to your work than being one contributor among many.

When I say I own this product technically, I mean that decisions about architecture, performance, and where engineering effort goes next do not get handed to me from someone else's roadmap. They start with me. That is a heavier kind of responsibility than it sounds like from the outside, because it removes the comfortable excuse available to almost every developer at some point in a difficult project: "I just built what I was told to build." When something goes wrong with ExtendedForms.io, there is no ambiguity about whose judgment is being questioned. I have found, somewhat unexpectedly, that removing that excuse entirely makes the work more satisfying rather than more stressful,

most of the time. There is a clarity to full ownership that shared responsibility never quite offers. You are not waiting to find out whose call a decision was. You already know.

The product today serves more than 401,000 users, processing 579,000 forms and 8.7 million respondents, and has driven 3x revenue growth during my ownership of it. Those numbers took years of compounding decisions to reach, not one clever launch, and I want to resist the temptation to make this chapter sound like a single triumphant redesign moment, because that is not how it happened. It happened in the same way the HQ Infosystem years built real expertise: repeated, careful, sometimes tedious decisions, most of them invisible individually, that only add up to something impressive when you step back far enough to see the whole shape of the accumulation.

I rebuilt performance-critical paths, taking page load time from 3.2 seconds down to 1.2 seconds, and later going further, cutting the load time on our highest-traffic pages from a genuinely painful 30 to 35 seconds down to just 5 to 6 seconds using a combination of smarter caching and TanStack Query for data fetching. I want to be specific about that particular fix, because it is a good example of what real ownership looks like in practice: nobody filed a ticket demanding a 30-second page get fixed by a specific date. I noticed it was a problem, understood it was costing us users at the exact moment they were trying to trust the product

with their most demanding use cases, and treated it as urgent because I was the one who would have to live with the consequences of ignoring it, not because someone above me assigned it a priority label. I remember the specific frustration of watching that page load slowly in a test environment, knowing that a real user somewhere was experiencing the exact same thirty-second wait, probably assuming the product was simply broken rather than merely slow, and deciding, on my own initiative, that this was more important than whatever else was next on an informal list of priorities.

Growing the product's organic traffic by 150 percent came from a similar instinct: performance and product quality are themselves a growth strategy, not a separate department's job. When a page is fast and a feature works reliably, users tell other users, and search engines notice the same signals that human users notice. I have never separated the "engineering" work on this product from the "growth" of this product in my own head, even when other people's org charts would draw that as two different boxes. I think this is one of the more valuable habits ownership has taught me, and one I would recommend to any engineer who eventually finds themselves with real product responsibility: stop thinking of performance work as a purely technical concern that happens to have business side effects, and start thinking of it as directly, inseparably, a growth lever in its

own right, deserving the same strategic attention a marketing campaign would receive.

The feature I am probably most proud of building into ExtendedForms.io is our AI Watchers capability with face detection, built to protect the integrity of exams and assessments run through the platform. It sits at the exact intersection I find most interesting in my work: a real technical challenge, wrapped around a real, human stake, in this case the fairness of an exam that matters to the student taking it. Building that feature well required care that goes beyond "does the model work." It required thinking hard about false positives, about the experience of a student being incorrectly flagged, about the difference between a feature that is technically accurate and a feature that is fair to the person on the other end of it. That is the kind of thinking that only really shows up once you own the outcome, not just the ticket. I spent a genuinely uncomfortable amount of time considering what happens to a real student, in a real moment of real academic stress, if this system gets something wrong about them, and I do not think that consideration would have received the same weight if the feature belonged to someone else's roadmap and I was simply executing against a specification handed down to me.

Owning a product changes your relationship to your own work permanently, in a way that is hard to explain to someone who has only ever built things other people were

ultimately responsible for. You stop asking "is this done" and start asking "is this right," and those two questions, despite sounding similar, lead to very different amounts of care in the final ten percent of any piece of work. "Is this done" is satisfied by a checklist. "Is this right" is only satisfied by genuinely imagining the real person on the other end of the feature, and asking whether you would be comfortable if that person could see exactly how the decision was made. I did not always ask myself that second question earlier in my career. Ownership taught me to ask it by default, on almost everything, and I do not think I could stop asking it now even if I wanted to.

I also want to note, honestly, that ownership at this scale is not without its costs. There is no clean boundary, most weeks, between "ExtendedForms.io's problem" and "my problem," and that blurring is part of what makes the role meaningful and part of what makes it genuinely demanding. I do not think every engineer should seek this specific kind of ownership, and I do not think it is automatically better than being an excellent, focused contributor inside someone else's clearly scoped roadmap. I think it is simply a different relationship to the work, with different rewards and different weight, and I have found, for myself specifically, that the weight is worth carrying.

CHAPTER NINE

Building Quzo.ai From Nothing

"I gave a machine the job of writing exam questions. It took thirty seconds. It used to take thirty minutes, and none of those minutes were mine to spare."

Quzo.ai is the product I am proudest of in a different way than ExtendedForms.io, because I built it entirely from scratch, on my own, outside of any employer's roadmap or resources. It is an AI-first exam and quiz platform with proctoring, and it now handles more than 400,000 student exams. There was no point in its history where someone else's budget or someone else's decision was the reason it existed. It exists because I decided it should, and then I built it. I want to sit with that sentence for a moment, because there is a real difference, psychologically, between building something inside an organization that will exist whether or not your specific project succeeds, and building something where the entire existence of the thing rests on a decision you made alone, on your own time, with your own judgment about whether it was worth the hours.

I started Quzo.ai during what I think of as an early, formative stage of practical AI tooling, and I leaned into that timing deliberately rather than cautiously. The landscape of AI-assisted development was moving quickly enough during this period that waiting for it to stabilize before starting

would have meant waiting indefinitely, because it has not really stabilized since, it has simply kept moving. I decided early that the correct response to a fast-moving landscape was not caution but fluency: get comfortable enough with the tools as they existed at that moment that I could keep adapting as they changed, rather than waiting for a settled version of the technology that was never actually going to arrive on any predictable schedule.

I built the product using a combination of OpenAI, Claude from Anthropic, and Gemini, giving the platform a credit management and optimization system that lets the underlying AI model be selected based on the task and cost tradeoffs, rather than locking the whole product to a single provider's pricing and capability curve. This turned out to be one of the more important architectural decisions in the entire product, though I did not fully appreciate how important at the time I made it. Locking a product to a single AI provider feels simpler in the short term and becomes a genuine liability the moment that provider's pricing changes, or a competitor's model becomes meaningfully better at a specific task, or an outage takes the provider offline entirely. Building in flexibility from the start cost more effort upfront and has paid for that effort many times over since.

I built a retrieval-augmented generation pipeline using pgvector on top of PostgreSQL to power AI-generated question banks, which took a task that used to require an

educator's hours, writing and organizing exam questions by hand, and cut it down dramatically: what used to take thirty minutes now takes about thirty seconds. I remember the first time the pipeline produced a genuinely good, well-structured set of questions from source material I fed it, and feeling a version of the same satisfaction I had felt writing my very first working script as a student, just scaled up enormously in both technical complexity and real-world usefulness. The underlying feeling had not changed at all across all those years. I had simply gotten much better at generating it on purpose, at a scale that now actually helps thousands of educators and hundreds of thousands of students.

I was also an early adopter of the current generation of AI-native development tools, building parts of Quzo.ai using v0.dev, Windsurf, Cursor, Antigravity, and Claude Code, and I set up custom agent rules, memory, and workflows so that specific agents could handle specific functions inside the product, including support-facing and engineering-facing responsibilities. Working this way required a real mental adjustment. For most of my career, I had been the one writing every line personally and reviewing every decision by hand. Learning to direct AI-native tooling effectively meant learning a new kind of leadership, not of people this time, but of process: deciding what a tool should be trusted with, what still needs a human check, and how to write instructions precise enough that an agent produces

something usable on the first or second pass rather than the tenth. This adjustment rhymed, more than I expected it to, with the adjustment I had already made becoming a technical lead at ExpressTech: in both cases, the actual skill being developed was not doing the work myself faster, it was getting a capable but not fully autonomous entity, whether a junior engineer or an AI agent, to produce reliably good outcomes with the right combination of clear instruction and appropriate oversight.

Beyond the core exam and question generation experience, I built AI-powered photo-based conversation features, AI video animation generation, and AI video conversation capabilities into the platform, extending it well past a simple quiz tool into something closer to a full AI-assisted learning companion. Each of these features started as a smaller experiment, something I was curious whether the current generation of AI tooling could actually support well, rather than a feature planned months in advance on a formal roadmap. That is one of the real advantages of building something entirely on your own terms: the roadmap can bend toward genuine curiosity rather than needing to be justified in advance to a committee that has not yet seen the technology prove itself.

None of that was built in isolation from the business side of the product. I ran organic keyword and SEO strategy for Quzo.ai in close collaboration with our marketing and

support teams, because I have never believed that a strong product markets itself automatically. Somebody has to actually do the unglamorous, ongoing work of making sure the right people can find it, and I have never been comfortable delegating that entirely to someone who does not also understand the product at a technical level. I think this cross-functional comfort, moving fluidly between architecture decisions and growth strategy conversations in the same week, sometimes the same day, is directly traceable back to growing up around Gujarat's business culture, where nobody around me ever treated "building the thing" and "selling the thing" as separate people's jobs by default.

Building Quzo.ai from nothing taught me something ExtendedForms.io, as much as I own its technical direction, could not fully teach me, because ExtendedForms.io existed inside a company with resources and infrastructure already in place. Quzo.ai taught me what it actually costs, in hours and attention, to take a product from a blank folder to four hundred thousand exams handled. There is a specific kind of confidence that only comes from having done that once, entirely on your own terms, and I carry that confidence into every technical and business decision I make now, including the harder ones described in the chapters ahead, particularly the founder chapter that follows a little later, where that same confidence was tested by an outcome that did not go the way Quzo.ai did.

CHAPTER TEN

Weekends That Mattered — HelpDesk AI and the Side-Project Habit

"Nobody assigned me a support ticket problem. I just couldn't stand watching a customer wait two days for something a weekend could fix."

Not every product in this book pays my bills directly, and I think that is worth being upfront about, because HelpDesk AI, one of the projects I am most satisfied with, was built entirely on weekends, alongside my full-time role at ExpressTech, for no reason other than that I wanted to improve my own skills and solve a problem I found genuinely interesting.

HelpDesk AI is a multi-tenant AI support agent platform with threshold-based automatic replies and ticket triage. The core idea is simple to state and was not simple to build well: most support tickets follow recognizable patterns, and a well-designed system can recognize those patterns and either resolve the ticket automatically or route it to the right person immediately, instead of making a customer wait while a human works through a queue one item at a time. In practice, this cuts support engineer time by roughly ninety percent for the tickets it can handle, and takes what used to be a one- or two-day resolution process down to a matter of

hours. I want to dwell for a moment on that specific gap, one to two days versus a few hours, because it is a good illustration of a broader pattern I have noticed across almost every efficiency project I have ever worked on: the gap between "eventually gets done" and "gets done promptly" is rarely a gap in effort. It is almost always a gap in queuing, in the invisible waiting time between when a task arrives and when the right attention finally reaches it, and that waiting time is usually the single biggest thing worth attacking first.

I built the platform with a multi-tenant architecture from the start, giving each client their own subdomain with strict data isolation enforced through JWT-based authentication and RFC-compliant security standards, along with AES-256 encrypted mailbox credentials and a dedicated singleton component I call the WatcherManager, responsible for reliably watching each connected mailbox over IMAP without the kind of resource leaks or missed messages that plague less careful implementations of the same idea. None of that complexity was required by any client contract or business deadline. I built it that way because building it any other way would have meant building something I was not proud to call finished, and because the year and a half I had already spent thinking like an attacker on Kali Linux made it almost impossible for me to design credential handling any other way. There is a specific satisfaction in solving a hard, unglamorous infrastructure problem, like reliably watching

dozens of mailboxes simultaneously without ever silently dropping a message, purely because you know it is the right way to build it, with nobody checking whether you cut a corner or not. I did not cut the corner. I do not think I would have known how to live with having cut it.

I want to be honest about why I keep doing this, spending weekends on projects nobody is paying me to build, when I already carry a demanding full-time role. Part of the answer is straightforward: it keeps my skills sharp on my own terms, without waiting for my employer's roadmap to happen to intersect with whatever I am currently curious about. But the deeper answer is that weekend projects are one of the only places in my professional life where I get to be completely, unambiguously in charge, from the very first architectural decision to the smallest naming choice in the code. There is a particular kind of clarity that comes from that, a feedback loop with no committee in the middle of it, and I think that clarity is part of why the technical decisions I make on these projects tend to be some of the cleanest ones I make anywhere.

There is also a quieter, more personal reason, which I have not always said out loud as plainly as I am about to say it here. A weekend spent building something real, from an idea to a working system, is a specific kind of rest for me, even though it does not look like rest from the outside. It is not the same category of activity as the demanding,

high-stakes decisions I make during the week at ExpressTech, even though the underlying skill involved overlaps significantly. There is something restorative about a project where the only person I am answerable to is myself, and I think everyone who has sustained a demanding career for a long stretch eventually needs some version of that outlet, whether it looks like a weekend coding project or something else entirely.

I do not think everyone needs to build a full product on their weekends to have a strong career. I have colleagues and friends I respect enormously who protect their weekends fiercely and are excellent engineers regardless, and I want to be clear that this chapter is not an argument that my particular relationship to free time is the correct one for everybody. What I would say, for anyone reading this who is curious whether this kind of side-project habit is worth the tradeoff, is that the value is not primarily in what the project becomes, though HelpDesk AI became something genuinely useful. The value is in the muscle you build by repeatedly making an idea real without anyone else's permission or resources standing between the idea and the outcome. That muscle transfers directly into every larger, higher-stakes decision you will eventually be trusted with, including the ones described in the next chapter, where I finally put that same instinct behind a company of its own, with much higher stakes and a much less forgiving outcome.

Looking back, I can trace a fairly direct line from the small, unsupervised scripts I wrote as a student in Surat, through fifteen-plus client projects at HQ Infosystem, through a year and a half of Kali Linux nights, through the architecture decisions I now make as a technical lead at ExpressTech, and into a weekend project like HelpDesk AI, built for no reason except that building it was worth doing. It is, in a real sense, the same underlying habit expressed at increasing scale and increasing stakes, and I do not think that habit would have grown as strong as it has if I had ever fully stopped exercising it in low-stakes, self-directed settings like this one, purely because a full-time job was already demanding enough on its own.

CHAPTER ELEVEN

MB Systems — Founding, Failing, and What It Taught Me

"A startup doesn't fail in one dramatic moment. It fails the same quiet way it almost succeeds: one ordinary week at a time."

From 2023 to 2024, for about a year, I founded and ran a startup called MB Systems in Surat, alongside my full-time role at ExpressTech. I want to state plainly, before anything else in this chapter: MB Systems did not survive. I am not writing this chapter to disguise that outcome as a hidden success, and I am not writing it to bury the good parts out of embarrassment about the ending. Both things are true at once, and I think a book that only tells one of those truths is not worth the reader's time. I have read enough founder memoirs that quietly reframe a shutdown as a strategic pivot to know how tempting that reframing is, and how little it actually helps anyone reading it for honest guidance. I am not going to do that here.

During that year, MB Systems delivered client projects and Shopify e-commerce builds for local businesses in Surat, and we built the BPS crypto trading platform, which went on to reach more than ten thousand users and over two million dollars in trading volume, with more than a thousand concurrent WebSocket connections handling live order

placement and streaming across multiple crypto pairs. We trained six to seven interns during that year, and every one of them is now well-settled in their own careers, which is one of the outcomes from this chapter I feel the least conflicted about. Whatever else happened to the studio itself, that mentorship was real and it mattered to real people, independent of whether the business around it ultimately survived. I think about those interns more often than I think about the specific financial details of the studio's closure, and I suspect that says something true about which parts of a venture like this actually matter most in the long run.

What founding MB Systems actually taught me is different from what any of the individual projects taught me, and it is worth separating those lessons out clearly, because I think they generalize to almost anyone considering their own founder leap.

I learned how markets actually move, in a way that reading about markets never taught me. As an employee, even a technical lead, you are shielded from a significant amount of market reality by the structure of the company around you. As a founder, there is no structure between you and the market's actual response to what you built. Demand does not arrive because you worked hard. It arrives, or it doesn't, based on factors that are often only partially within your control, and learning to read those signals honestly, rather than through the lens of how much effort you

personally invested, was a genuinely humbling adjustment. I had, before this year, an implicit belief that good work reliably finds its reward if you simply keep producing it. That belief did not survive this year intact, and I think it needed to break, because it was never quite true, even though it had served me reasonably well up to that point.

I learned to manage developers directly, for the first time without an existing organizational structure absorbing the hardest parts of that responsibility for me. Hiring, delegating clearly, giving technical direction that someone else could actually execute without me hovering over every decision, all of that required skills that are adjacent to but distinct from being a strong individual engineer or even a strong technical lead inside an established company. At ExpressTech, HR processes, established review cycles, and years of accumulated organizational practice absorb a huge amount of the difficulty of managing people well. As a founder, I had none of that scaffolding, and I had to build my own version of it, quickly, while also trying to keep the underlying business afloat.

I picked up, out of necessity rather than by choice, a project manager's habits: juggling scope, timelines, client expectations, and my own limited hours across far more roles at once than any single job description would normally ask of one person. There is no clean separation, as a founder, between "engineering work" and "everything else." It is all

just the work, and you do whichever part of it is most urgent on a given day, regardless of what your formal title would suggest you should be spending your time on. I remember weeks where I moved between writing production code in the morning, handling a client's frustrated call at midday, and reviewing an intern's pull request in the evening, all before touching my actual responsibilities at ExpressTech the next day. It was, without exaggeration, one of the most demanding stretches of my working life, and I do not think I fully appreciated how demanding it was until it ended and the sudden absence of that pace made the previous pace visible by contrast.

I learned to handle clients end to end, including the harder conversations: scope changes, missed timelines, the moments where expectations and reality diverge and someone has to be the one to say so honestly rather than let the gap grow quietly. That is a different skill than technical communication, and it does not come naturally to most engineers, myself included at the start of that year. I had delivered client work before, extensively, at HQ Infosystem, but always with a layer of account management or project leadership between me and the hardest parts of that relationship. As a founder, that layer did not exist. I was the layer, and I had to learn quickly, sometimes through conversations that did not go well the first time, how to be direct without being cold, and honest without being

defeatist, when a project was clearly not going to land on the original timeline.

And I took on data and security decisions personally, as the person ultimately accountable if something went wrong, rather than as one engineer contributing to a shared responsibility spread across a larger team. That single shift, from shared accountability to personal accountability, changed how carefully I thought about every technical decision during that year, in a way that has stayed with me since, even back inside a larger organizational structure. I still, today, think about certain technical decisions at ExpressTech with a version of the same personal weight I carried during that founder year, even though the formal accountability structure around me is different now. I do not think that carryover was an accident. I think it is one of the more permanent, useful residues the entire experience left behind.

I want to spend a moment on the ending itself, because I think the honest, specific version of a failure is more useful to a reader than a vague acknowledgment that "it didn't work out." The studio closed because the combination of cash runway, the time I could realistically devote to it alongside a demanding full-time role, and market timing did not line up in our favor, and no single dramatic mistake explains the outcome better than that unglamorous combination does. There was no single bad decision I can point to and say,

clearly, that fixing it alone would have saved the business. It was a slower erosion of margin for error, across several fronts at once, until there was no margin left. I think that is actually the more common way startups fail, more common than the dramatic collapse stories that make for better anecdotes, and I would rather tell the less dramatic, more accurate version here.

The honest ending of this chapter is that MB Systems closed after about a year, and I came away from it with a much clearer read on why startups fail in general: cash runway and timing matter as much as the product itself, sometimes more, and no amount of technical excellence fully insulates a young company from either of those pressures. I do not regret the year. I would be lying if I said the ending did not sting. Both of those statements are true, and I think that is exactly what an honest account of a failed venture is supposed to sound like. If I ever start something like this again, and I have not ruled that out, it will be with everything this chapter cost me already paid for, in advance, rather than learned again the hard way a second time.

CHAPTER TWELVE

The Craft — Charts, Maps, and Chasing New Tools

“A chart that lags is the only kind of slow database a user can actually see with their own eyes.”

There is a category of skill that rarely makes it into a job description in the way core languages and frameworks do, but that has quietly shaped a significant portion of my career: the craft of visualization and interaction, the part of the work where a product stops being merely functional and starts being genuinely pleasant, even a little bit delightful, to use. Most engineering education, and most engineering job descriptions, treat this category as a nice-to-have, secondary to the "real" work of correctness and performance. I have never fully agreed with that ranking, and this chapter is, in part, an argument for why.

I have spent years working with Highcharts, building bar and radial charts and heavily customized chart designs capable of rendering millions of data points without the interface grinding to a halt. Handling that kind of scale in a chart is a different problem than handling it in a database. A database can be slow in ways a user never sees, quietly optimized in the background while the user waits, blissfully unaware of how much work just happened on their behalf. A chart's performance is the user's direct, visible experience of

whether your engineering is any good, rendered in real time in front of them. There is nowhere to hide a slow chart the way there is sometimes room to hide a slow background job. I have spent real hours thinking about data downsampling strategies, about which points in a dataset are safe to aggregate without misleading the viewer, about the specific rendering tricks that let a chart feel instantaneous even when the underlying dataset genuinely is not small. This is a narrower, more specialized skill than general frontend development, and I think it is chronically undervalued relative to how much it actually affects whether a data-heavy product feels trustworthy to the people relying on it.

Mapbox has been another consistent thread, from geographic implementations and the Directions API to custom markers and, most interestingly to me, automatic geofenced and radial trigger systems: setups where crossing into or out of a defined area on a map automatically fires an action in the product, without a human needing to notice and respond manually. Building a reliable geofencing system sounds, on paper, like a straightforward exercise in comparing coordinates. In practice, it involves real, subtle engineering: accounting for GPS drift and inaccuracy near a boundary, deciding how to debounce a signal that flickers back and forth across an edge without firing the trigger repeatedly and uselessly, and designing the system so that a temporary loss of signal, extremely common in real-world

mobile use, does not silently break the whole feature. I built exactly this kind of system for a truck driver tracking application, a React Native driver app paired with a companion monitoring website, for a New York-based client, with a tracking experience modeled on the kind of live delivery tracking that platforms like Zomato and Blinkit have made familiar to users across India. That particular project was ultimately discontinued by the client before launch, and I include it here deliberately, alongside the projects that did ship, because the technical craft of building the geofencing and live tracking system was real and valuable regardless of the business outcome around it. Not every well-built piece of software gets to see daylight, and that does not retroactively make the building of it a waste. I learned real, durable lessons about geofencing reliability from that project that I still apply elsewhere, entirely independent of whether the specific client's business plan ever came to fruition.

Beyond charts and maps, I have spent real time with Three.js for embedding three-dimensional models directly into web experiences, controlling camera views and enabling model rotation and interaction, paired with Framer Motion for interface animation, Lenis for the specific, satisfying feel of smooth scrolling, and Animate.js for lighter-weight interaction polish. None of these tools are, strictly speaking, necessary to ship a working product. A form still submits without a smooth scroll effect. But there is a real difference

in how a product feels to use, and feel is not a decoration on top of function, it is very often the thing that determines whether a user trusts the product enough to keep using it past the first few minutes. I have watched, more than once, a technically identical feature receive a dramatically warmer reception once its interaction design was given real attention, and I do not think that warmer reception was irrational or superficial on the users' part. A product that feels considered in its small details signals, correctly, that it was probably considered carefully in its larger, less visible decisions too.

More recently, I have leaned deliberately into the current wave of AI-native development tools: v0.dev, Windsurf, Cursor, Antigravity, and Claude Code, treating fluency with these tools as seriously as I once treated fluency with a new JavaScript framework. I think this is one of the more important instincts in this entire book, more important in some ways than any single technology I have named: the willingness to treat "I don't know this tool yet" as a temporary and uninteresting fact about the present moment, rather than as a real barrier. Every framework, every API, every AI tool I now use fluently was, at some point, something I had never touched before. I remember feeling genuinely slow and clumsy the first week I tried directing an AI coding agent instead of writing every line myself, and I remember that feeling passing within a matter of weeks,

replaced by a new, faster default that I now cannot quite imagine working without. The specific tool changes. The pattern of becoming fluent quickly does not.

The skill that actually compounds over a career is not deep permanent expertise in any one specific tool, because tools change faster than any career lasts. I have watched entire ecosystems rise and fade during my own working years, tools that once seemed permanent fixtures of the industry now used by almost nobody, replaced by successors that will themselves eventually be replaced. The skill that compounds is the confidence and process for becoming fluent in whatever tool the next problem happens to require, quickly enough that the tool itself stops being the bottleneck. I try, deliberately, to notice when I am resisting a new tool out of genuine technical concern versus simple discomfort with unfamiliarity, and to be honest with myself about which of the two is actually operating, because only one of them is a good reason to hold back.

That, more than any specific chart library or animation framework, is the actual craft I have been building this whole time. The libraries are the visible evidence of the craft. The craft itself is quieter, and it is the thing I would most want a younger engineer reading this to actually take from this chapter: build the muscle of becoming fluent fast, and stop treating any specific tool as a permanent identity. None of

mine have been, and I do not expect the ones I am currently comfortable with to be permanent either.

CHAPTER THIRTEEN

Leading Without a Title — Mentorship Lessons

"The fastest way to help someone is to hand them the answer. The only way to actually help them is to resist that urge."

Across ExpressTech and MB Systems, I have now directly mentored more than a dozen engineers and interns, and I want to use this chapter to talk about what that has actually taught me, because I think mentorship is one of the most misunderstood parts of a technical career, treated too often as a soft, secondary responsibility rather than a genuine skill with its own learning curve, its own failure modes, and its own real craft to get right.

The first thing I learned is that good mentorship is not the transfer of your own solution to someone else's problem. It is far more tempting, and far less useful, than it sounds. When a junior engineer comes to me stuck on something, the fastest path is almost always to simply tell them the answer. I have had to train myself, repeatedly, to resist that fastest path, because the value of the moment is not in the engineer receiving my specific solution, it is in them building the muscle to arrive at solutions like it on their own the next time I am not in the room. Every time I hand someone the answer instead of the process for finding it, I have optimized

for my own convenience at the cost of their growth, and it took me longer than I would like to admit to notice I was doing that. There is a specific, almost physical discomfort in watching someone struggle toward an answer you could simply hand them in ten seconds, and learning to sit with that discomfort, rather than relieve it prematurely, has been one of the harder personal disciplines I have built.

The second thing I learned is that people learn to trust a codebase's standards, testing discipline, and review process much faster when they can see a leader hold themselves to the same standard, rather than simply enforce it on others. When I pushed for Jest coverage above ninety percent and brought in Cypress for end-to-end testing at ExpressTech, that request landed very differently once I had already applied the same discipline to my own code first. Standards imposed from above, without a visible personal example behind them, tend to be followed reluctantly and abandoned the moment nobody is watching. Standards demonstrated first tend to be adopted because people can see, concretely, what the payoff looks like, and because they are being asked to do something their lead has already visibly done, rather than something asked of them alone.

The third thing, and the one that surprised me most, is how much mentoring interns at MB Systems during a genuinely difficult, resource-constrained year taught me about patience under pressure. It is easy to be a generous,

patient mentor when the business around you is comfortable and stable. It is a much harder and more revealing test of your actual values when the business is struggling and every hour feels expensive. I am proud that the six or seven interns who came through MB Systems during that year are now well-settled in their own careers, and I think that outcome says more about the kind of leader I actually am under pressure than any of the technical metrics from that same period. There were specific moments during that year when an intern's mistake cost real, scarce hours, at a time when the studio genuinely could not afford to lose hours, and the temptation to respond with visible frustration rather than patient correction was real. I do not think I handled every one of those moments perfectly. I think I handled most of them well enough, and I learned, from the ones I handled less well, exactly what patience under real pressure actually requires from a leader, which is different from and harder than patience during comfortable, low-stakes periods.

I also want to talk about a subtler part of mentorship that took me longer to notice: the way you frame a mistake shapes whether the person you are mentoring keeps taking risks or starts playing defensively small. An engineer who is corrected in a way that feels like judgment tends to become cautious, running every future decision past you before acting, which defeats the entire purpose of mentorship, since the goal was independence, not a permanent dependency on

your approval. An engineer corrected in a way that feels like collaborative problem-solving, where the mistake is treated as useful information rather than a personal failing, tends to keep experimenting, keep taking reasonable risks, and grows faster as a result. I did not always get this framing right, especially early on, and I have watched the difference in outcome closely enough over the years to trust that it is real rather than a comfortable story I am telling myself.

I do not think leadership requires a title, and I do not think the impulse to mentor should wait for a formal management position to arrive before it starts being exercised. The habits that make someone a good technical lead, patience, the willingness to explain reasoning rather than just conclusions, the discipline to model the standard rather than only demand it, are habits you can start building the moment you have any knowledge at all that someone else does not yet have. I started building them long before ExpressTech gave me a formal title that matched, and I think that early practice is a large part of why the title, when it eventually arrived, fit rather than felt like a costume I was still growing into. If you are early in your own career, wondering whether mentorship is something to wait for until you feel senior enough to offer it, my honest advice is to start now, informally, with whoever around you currently knows slightly less than you do about something. The title will follow the practice. It rarely arrives ahead of it.

CHAPTER FOURTEEN

A Field Guide to Cracking DSA and Technical Interviews

"You don't crack an interview in the room. You crack it in the two hundred problems nobody ever watched you solve."

This chapter and the next are written slightly differently from the rest of the book. Where the earlier chapters told the story of what happened to me, these two are written directly to you, whoever you are reading this while preparing for your own interviews or your own first job search. I have tried to write down the exact things I wish someone had told me plainly, without the usual vague encouragement that fills most advice on this topic.

Start with the shapes, not the problems. There are a genuinely small number of underlying patterns behind the enormous volume of practice problems available online: two-pointer techniques, sliding windows, breadth-first and depth-first traversal, dynamic programming built on recognizing overlapping subproblems, backtracking for exploring a decision tree of possibilities, and a handful of others. Your early practice should be organized explicitly around learning to recognize these shapes underneath unfamiliar problem descriptions, not around grinding an undifferentiated pile of random problems in whatever order

a website happens to present them. Pick a pattern, work through a cluster of problems built on that same pattern until the recognition becomes automatic, and only then move to the next pattern. Random, unstructured practice feels productive because you are always solving something, but structured, pattern-first practice builds transferable recognition far faster.

Talk out loud while you practice, every time, starting well before your first real interview. Silent problem-solving and narrated problem-solving are genuinely different skills, and an interview only ever tests the second one. Practicing only the first and expecting it to transfer cleanly is one of the most common and avoidable mistakes candidates make. I would go further: record yourself narrating a solution occasionally and listen back to it. Almost everyone is surprised, the first time, by how much less clear their spoken reasoning is compared to how clear it felt inside their own head while they were thinking it. That gap is exactly what practice closes.

Get comfortable with the specific discomfort of being wrong in front of someone. You will take wrong turns in real interviews. Everyone does. What separates a strong outcome from a weak one is almost never the absence of a wrong turn, it is whether you can notice it, say so calmly, and correct course without your composure visibly collapsing. Practice this deliberately, not just the problems themselves. One

useful exercise is to deliberately pick a wrong initial approach during practice, notice it is wrong partway through, and practice the specific language of correcting course out loud: "I am noticing this approach won't handle this edge case cleanly, let me reconsider," said calmly, is a skill in itself, separate from the problem-solving underneath it.

Understand time and space complexity well enough to discuss tradeoffs, not just well enough to label a solution with a formula. An interviewer asking about your approach's complexity is very rarely testing whether you can name it correctly. They are testing whether you understand why it is what it is, and whether you can identify a different approach with a different tradeoff if the constraints of the problem changed. Practice this by taking solutions you have already written and asking yourself, unprompted, what would happen to the complexity if the input were ten times larger, or if memory were far more constrained than time, or vice versa.

Do not treat mock interviews as optional. The specific pressure of being watched while thinking does not disappear just because you understand data structures well. It has to be practiced directly, the same way public speaking has to be practiced directly even by someone who already knows their material cold. Find a peer, a mentor, an online mock interview partner, anyone willing to sit across from you

while you work through a problem in real time. The discomfort of doing this a handful of times in low-stakes practice is dramatically cheaper than experiencing that same discomfort for the first time in an interview that actually matters.

Treat every rejection as data, not verdict. I did not pass every interview I ever sat for. Nobody who has interviewed extensively has. The engineers who improve fastest are the ones who can extract a specific, actionable lesson from a rejection, rather than absorbing it as a broad, discouraging judgment about their overall ability. After a rejection, if you can, ask specifically what area felt weakest to the interviewer. Even a vague answer narrows your next round of preparation. And if no specific feedback is available, review your own recording or memory of the interview honestly, looking for the exact moment things went sideways, rather than accepting a vague, discouraging feeling that "it just didn't go well."

A brief word on system design and behavioral rounds, since most first-time candidates focus almost entirely on DSA and are caught slightly off guard by these. System design rounds, at a first-job level, are rarely testing whether you can architect a system at the scale of a major global platform. They are testing whether you can reason clearly about tradeoffs at a smaller, honest scale: how would you structure a basic service, where would data live, what

happens under moderate load, and can you explain your reasoning rather than simply naming impressive-sounding technologies you have read about but not actually used. Behavioral rounds are testing something equally simple underneath the surface: can you describe a real situation, honestly, with a clear account of what you did and what you learned, without either inflating your role or deflecting all credit away from yourself. Prepare two or three genuine stories from your own experience, even if that experience is limited to college projects, and practice telling them clearly and honestly rather than searching for a story that sounds more impressive than the truth.

And finally, remember what the interview is actually simulating. It is a compressed, artificial version of a real skill you will use constantly once you have the job: holding a problem in your head, breaking it into correct smaller pieces, and communicating your reasoning clearly to people who need to trust your judgment. Prepare for the artificial version as seriously as you can, but do not lose sight of the fact that the real skill underneath it is the one that will actually carry your career forward, long after any specific interview is a distant memory.

CHAPTER FIFTEEN

A Field Guide to Landing Your First Job

"Your first offer isn't proof that you're already excellent. It's only proof that someone was willing to bet you would be."

Landing a first job is a genuinely different problem than landing a fifth job, and most career advice does not distinguish clearly enough between the two. With no professional track record to point to, you are asking someone to make a bet on potential rather than evidence, and the entire strategy for a first job search should be built around making that bet feel as safe and well-informed as possible for the person taking it.

Build proof before you need it. A portfolio of real, working projects, even small ones, is worth more than almost anything else you can put in front of a hiring manager who has never met you. It does not need to be commercially successful. It needs to demonstrate that you can take an idea from nothing to something that actually runs, because that is precisely the uncertainty a first-time hiring manager is trying to resolve about you. A single small project, actually finished and actually working, with a short honest description of the decisions you made along the way, tells a

hiring manager more than a long list of course names ever will.

Treat your academic performance and your placement season, if your path includes one, as a compounding asset rather than a single event. The distinction I earned in college and the highest package offered in my batch did not happen because of anything dramatic in my final semester. They happened because of a long, unglamorous accumulation of consistent effort, and that same accumulation is available to anyone willing to start it early rather than waiting for the pressure of a deadline to force it.

Do not wait for the market to be favorable before you start preparing seriously. I have already written a full chapter about the period widely remembered as one of the worst hiring environments in recent memory, and about how I used that exact period, rather than waiting it out, to build the depth that later separated me from people who had simply paused. A difficult market is a reason to prepare more deliberately, never a reason to prepare less.

Learn to talk about your projects the way you would talk about a real job's responsibilities, in terms of the problem, the decision you made, and the outcome, rather than simply listing the technologies involved. A hiring manager evaluating a first-time candidate is trying to imagine you inside their team's actual problems. Naming a tech stack

does very little to help them do that. Describing a decision you made and why does a great deal. Instead of "I built a project using React and Node," try "I noticed this specific problem, chose this approach because of this specific tradeoff, and here is what I would do differently now that I understand it better." That last part, the honest reflection on what you would improve, is disproportionately persuasive, because it demonstrates the exact kind of self-awareness a growing engineer needs and a memorized answer cannot fake.

Apply broadly, but prepare narrowly and deeply for the roles you actually care about. There is a difference between casting a wide net for opportunities and treating every application with equally shallow effort. The candidates who stand out are usually the ones who did visibly deeper homework on the small number of roles that genuinely mattered to them, not the ones who submitted the largest volume of generic applications. A short, specific note about why a particular company or team's work genuinely interests you, attached to an application, costs you a few extra minutes and is read by almost nobody else applying to the same role.

Do not underestimate the value of a genuine network, even a small one. This does not mean cold messaging strangers with a generic request. It means staying in real touch with classmates, seniors, professors, and anyone from

a project or internship who saw your actual work firsthand. A first job, more often than most people admit, arrives through some version of a real relationship rather than a cold application into an anonymous queue, and building that relationship honestly, well before you need anything from it, is worth far more effort than most students give it credit for.

Prepare for the possibility that your first offer will not be your dream role, and think in advance about what you actually need from a first job versus what you would simply prefer. A first role's most important job, in most cases, is not prestige, it is giving you real, structured exposure to how software actually gets built and shipped inside a functioning team, under real constraints. That exposure compounds into everything that comes after it, the way HQ Infosystem's four years quietly became the foundation for everything I built later, well beyond what any of us could have predicted at the time the offer letter first arrived.

And once the offer comes, understand that it is the beginning of the real learning, not the end of the hard part. Nothing in my own first offer, at HQ Infosystem, taught me as much as the years of actual work that followed it. The interview process, however stressful it feels while you are in it, is a small and temporary gate compared to the much longer and more interesting road on the other side of it. Prepare for the gate seriously. Do not mistake the gate for the destination.

CHAPTER SIXTEEN

What's Next

“I don't know what I'm building next. I know exactly how I'll start: badly, immediately, and without waiting to feel ready.”

I am writing the closing pages of this book from the middle of my career rather than the end of it, which is a strange and slightly uncomfortable place to try to draw conclusions from. There is no tidy resolution to offer, because nothing here has actually resolved. ExtendedForms.io keeps growing, and I keep finding new performance ceilings to push against once the current one has been cleared. Quzo.ai keeps handling more student exams every term, and the AI landscape underneath it keeps changing quickly enough that decisions I make about it today may look outdated within a year. MB Systems is closed, and the lessons from that year are still, honestly, being processed a little further with each new decision I make that turns out to reflect them.

What I can offer instead of a tidy resolution is a fairly clear statement of what I am actually looking for next, because I think that is more useful to a reader than a false sense of closure would be. I want to keep exploring genuinely new opportunities rather than settling into comfort simply because comfort is available. I have never been the person

who treats stability as the primary goal of a career, and I do not expect that to change now, even though I understand, more than I did before MB Systems, exactly what the cost of that restlessness can be when a bet does not land.

I want to keep building the kind of judgment that comes from thinking like a business owner even when my formal role is purely technical, because I think that dual lens, engineer and strategist at once, is the actual source of most of the decisions in this book I am still glad I made. I want to keep treating research and development as a genuine strength rather than an occasional activity, staying close enough to new tools and new approaches that I am never the person left behind when the landscape shifts again, the way it shifted with the current generation of AI-native development tools, and the way it will shift again toward something none of us have quite seen yet.

I also want to keep the security-minded lens I built during the Kali Linux period alive in everything I do, even though it never became my formal career. It would be easy to let that instinct fade now that it is no longer an active daily practice, and I do not intend to let that happen, because I think it has made me a measurably better architect of every system I have touched since, in ways that are hard to quantify but easy to notice once you know to look for them.

And I want to keep mentoring, deliberately and consistently, regardless of whether I am ever given a formal management title larger than the one I currently hold. The chapter on leadership without a title was not written as a consolation for people who have not yet been promoted. It was written as an honest description of how I actually think this works: the title, when and if it arrives, is a recognition of a practice that was already underway, not a permission slip that starts the practice for the first time.

And I want to keep the habit that this entire book has really been about, more than any single product or job title: starting things before I am fully ready, on the theory that readiness is mostly built in the act of starting rather than achieved in advance of it. That habit got me through a placement season, a first technical interview, a pandemic that could have stalled everything, a year and a half spent chasing vulnerabilities for no reason except curiosity, a jump into technical leadership, two products I built and own, one startup that did not survive, and a dozen or more mentoring relationships along the way. I have no specific reason to believe it will stop working now, and every reason, based on everything in the chapters before this one, to keep trusting it.

I do not know, as I write this, exactly what the next version of this habit will produce. It might be a third product. It might be a return to founding something, better informed this time by everything MB Systems cost me to

learn. It might be something in a domain I have not touched yet at all, in the same way none of the AI-native tools described in an earlier chapter existed when I first sat down at a computer in Surat as a curious student with no plan beyond curiosity itself. I am comfortable with that uncertainty in a way I do not think I would have been comfortable with it a decade ago, and I think that comfort, more than any specific skill catalogued in this book, is the actual sign of everything these years have built in me.

If you have read this far, whether you know me personally or found this book looking for something more specific than generic career advice, I hope what you are taking away is not a template to copy but a permission to trust your own version of this same instinct: that consistent effort compounds even when nobody is watching, that failure is genuinely useful information rather than only a verdict, and that the right response to not knowing how to do something yet is almost always to start anyway, and learn the rest on the way.

That has been the whole story. Thank you for reading it.