LytQuest

THE FLAGSHIP BLUEPRINT

THE SOFTWARE ENGINEER BLUEPRINT SERIES

# The Software Engineer Blueprint.

How to build, evolve, and reinvent a career that lasts. For the engineer just starting, the one quietly stuck, and the one who has already arrived.

WRITTEN BY AKINTOMIWA (JBOSS) · LYTQUEST.COM

SECTION 00

# Contents

A field guide in six parts, written for the engineer just starting, the one quietly stuck, and the one who has already arrived.


00 **Three engineers, one question**
Why direction beats effort, and who this is written for

01 **Part One: For those just starting**
The road to Edinburgh, and why it does not start with a language

02 **Part Two: Learn to read your market**
Dreaming in company names, stacks, products over projects, geography

03 **Part Three: Depth over collection**
What interviews actually test, and why depth keeps you in the room

04 **Part Four: The most dangerous stage**
Comfort, blueprint paralysis, and the story I do not tell quickly

05 **Part Five: Every blueprint has an expiry date**
Nokia, 2go and the discipline of renewal

06 **Part Six: Bringing it all together**
Three questions to answer before you close this page

07 **Why LytQuest exists**
The ecosystem built around the Blueprint Principle

08 **About the author**
Akintomiwa (JBoss) Abolade, and an invitation to LytQuesters

Have you ever worked really hard at this craft, learning, watching tutorials, reading articles, buying courses, writing code almost every day, and still felt, quietly, that you were not making the progress you should be making? If that is you, I want you to know something before you read another line. You are not alone, and you are not broken.

One of the biggest lies we tell ourselves is that everyone else has figured this out except us. They have not. I have interviewed engineers with ten years of experience who had no idea where their career was going. I have mentored graduates who became senior engineers faster than people who had been writing code for years. So experience alone is not the difference. Time alone is not the difference. Even working hard, on its own, is not the difference.

The difference is direction.

That is what this piece is about. Not Java. Not Spring Boot. Not AWS. Not Kubernetes. Not AI. One idea, examined properly: the Blueprint Principle. How you build a blueprint, how you evolve it, and, hardest of all, how you know when to burn it down so you can build something better.

> **Busy people work hard. People with blueprints make progress.**

## Three engineers, one question

I have come to believe that almost everyone reading something like this is one of three people.

The first is trying to break into software engineering. You have heard that technology changes lives. You have seen friends get jobs, watched people relocate, watched people move from struggling financially to building meaningful careers. Every so often you tell yourself, I think I can do this. Then you open YouTube and there are thousands of videos. One person says start with Java. Another says Java is dead. Someone says learn Python. Someone else says AI has replaced programmers. Another insists you need Kubernetes before anyone will hire you. After ten videos you have learned nothing.

The second is already working as an engineer. You are earning. You are contributing. People call you an engineer. But something does not feel right. Every few months another framework appears, another AI tool, another trend, and you keep wondering whether you are learning the right thing. You are busy. You are not sure you are moving.

The third is senior. You have built systems, survived production incidents, mentored juniors, watched technologies come and go. You know what works. But the industry feels like it is changing faster than ever. Cloud. AI. Platform engineering. Large language models. Agentic systems. And sometimes, without realising it, we start defending what made us successful yesterday instead of preparing for tomorrow.

Three different people. Three different fears. One identical question underneath all of it: what is next?

For those meeting me for the first time, my name is Akintomiwa. Most people simply call me JBoss. I have spent the last sixteen years building software across different industries and different countries: startups and enterprises, government systems, retail, banking, financial services, logistics, education, agriculture. I have built in Nigeria, in the United States, and here in the United Kingdom. Today I work as a Principal Software Engineer.

None of that is the part of my story that matters most, because titles do not teach lessons. Experiences do. I have an instinct that has shaped my whole career: I chase problems rather than avoid them. Every unglamorous, badly documented, politically awkward problem I have taken on in one industry has quietly paid for the judgement I brought to the next one. That range is the only reason I trust what I am about to tell you. Everything here I have either lived through, or watched someone close to me live through.

I started LytQuest because of something I noticed over the years. There are brilliant people everywhere. Nigeria has them. Ghana has them. Kenya has them. India has them. The UK has them. Talent is not the problem. Direction is. I have met genuinely intelligent people who stayed in the same position for five years, not because they were incapable, but because nobody ever showed them how to think about a career, and nobody explained that software engineering is far bigger than learning programming languages.

# 01

PART ONE · THE BEGINNER

## You are not short of effort. You are short of a destination.

# Part One: For those just starting

I want to speak to beginners first, and I say the word with enormous respect. Every senior engineer you admire was once a beginner. Every Principal Engineer. Every Distinguished Engineer. Every CTO. Every founder. Every person earning well today once opened an IDE for the very first time and had no idea what they were looking at. Nobody skips that stage. So never be ashamed of being new. Beginners have one thing experienced people often lose: curiosity.

Here is a question worth answering honestly, even if only to yourself. How many programming courses have you started? How many YouTube playlists, how many Udemy courses, how many PDFs, how many bookmarked repositories? Now the harder question. How many did you actually finish?

If that answer makes you uncomfortable, good. Now we can talk honestly. Most people diagnose this as a discipline problem. I just need more motivation. I need more consistency. I need one more course. Almost always, that is the wrong diagnosis. The real problem is direction.

## The road to Edinburgh

Imagine it is Saturday morning and you are driving from London to Edinburgh. You have fuelled the car, checked the tyres, packed the bags, charged your phone. Everything is ready. You start driving. An hour in, you feel good. Two hours in, still driving. Three hours in, still driving. Then somebody in the car asks where exactly you are going, and you realise you never entered a destination into the GPS.

You are still moving. You are still burning fuel and spending time and feeling productive. But every mile could be taking you further from where you actually wanted to be.

That is exactly how a great many engineers build careers. Busy. Learning. Coding. Watching videos. Building random projects. Collecting certificates. Always moving, never once asking whether any of it is taking them towards the career they want. A blueprint is the GPS. It does not remove traffic. It does not stop roadblocks or delays. But when life takes you off course, it recalculates, because it always knows where you are heading.

## It does not start with a language

So what does an engineering blueprint actually look like? People assume it begins with choosing a programming language. It does not. It begins earlier than that, with choosing the kind of problems you want to spend your career solving.

Software engineering is not one industry. It is hundreds of industries. You could spend a lifetime building software for hospitals, banks, airlines, schools, gaming, agriculture, cybersecurity, artificial intelligence, space technology, retail. Every one of them needs engineers. So the first question is not what language should I learn. It is: what problems excite me enough that I would not mind spending years solving them? If someone paid you exactly the same salary in every

sector, which one would you choose? Healthcare? Education? Finance? Sport? Travel? That is where a blueprint begins. Not with syntax. With purpose.

## Part Two: Learn to read your market

Now I want to add something I have never seen taught properly, and it is the single biggest reason beginners stall. Most beginners do not struggle because they lack talent. They struggle because nobody ever taught them to read their market.

Think about how children are taught to dream. Nobody says, I want to work in a building. They say, I want to be a pilot. I want to be a doctor. They dream in specifics, and the specificity is what makes the dream actionable, because a pilot knows there is a licence, and hours, and a particular training school.

Engineers should dream with exactly that precision. Not I want to become a software engineer, but I want to work at Interswitch. I want to work at Uber. I want to work at Paystack. I want to work at Moniepoint. I want to work at Netflix. Naming an actual destination is what turns a vague hope into a plan. Pick a handful, and I would encourage a deliberate mix: some African companies, some global ones, so your ambition is not hostage to a single economy.

Once you have named your targets, the very next step, before you write a single line of code, is to find out what those companies actually run on. This is not guesswork, and it is not something you need permission to discover. Read their engineering blogs, because most serious engineering organisations publish one and most of them are startlingly candid. Watch conference talks where a Head of Engineering or a technical lead has stood on a stage and described their stack, their migrations, their regrets. Study their public job postings, which frequently list the exact technologies required in the exact order of importance.

Do that for a week and patterns emerge. Netflix has long been associated with Java at serious scale. A very large share of UK fintech and banking runs on Java, which is not fashionable to say but is true in the payroll data. Monzo is associated with Go. In Nigeria, Moniepoint, Interswitch and Remita are associated with Java, while Paystack is associated with JavaScript and Node. UK betting and gambling platforms often lean towards Go. I offer these as an engineer's observations rather than as gospel, and you should verify them yourself, but the exercise of verifying them is the skill I am actually trying to hand you.

REFERENCE: COMPANY TO STACK

Observed associations to verify yourself, not gospel.

**Netflix**
Java at scale
Streaming, global distributed systems

**UK fintech & banking**
Java
The quiet majority of the payroll data

**Monzo**
Go
Cloud native banking, microservices

**Moniepoint**
Java
Payments and agency banking, Nigeria

**Interswitch**
Java
Payment switching and infrastructure

**Remita**
Java
Collections and payment platforms

**Paystack**
JavaScript / Node
Developer first payments API

**UK betting platforms**
Go
High throughput, low latency systems

The conclusion is simple. Once you know your target company and its stack, that stack becomes the first technology you commit to. Not because it is trending on social media. Because it is the specific key to the specific door you have chosen to walk through.

> **Learn the stack that opens the door you have chosen, not the stack that is loudest on your timeline.**

## Build products, not projects

There is a second idea that belongs immediately beside the first. Once you know your market, stop building projects and start building products.

A project is generic and disconnected from any real business context. Another to-do app. Another calculator. Another CRUD demo that looks identical to ten thousand other bootcamp portfolios, because it is identical. A product feature is small, focused and modelled on something a real company in your chosen industry actually needs.

If Netflix is your target, do not try to rebuild Netflix. Build a recommendation engine, small and honest. If Uber is your target, build a dynamic pricing or fare calculation feature. The size of the feature matters far less than the depth of the execution. Design it properly. Write it. Test it. Add authentication so only the right people can reach it. Decide what happens when it fails, and make it fail well. Deploy it somewhere real, where it has an address and can embarrass you. Then be able to explain every decision you made along the way.

| A PROJECT | A PRODUCT FEATURE |
| --- | --- |
| **Generic. Disconnected.** | **Small. Real. Owned.** |
| • Another to-do app or calculator | • Netflix style recommendation engine |
| • CRUD demo identical to ten thousand others | • Uber style dynamic pricing or fare flow |
| • No industry context, no failure design | • Designed, tested, authenticated |
| • Runs only on your laptop | • Deployed somewhere with a real address |
| • Nothing to explain in an interview | • Every decision explainable end to end |

VS

That kind of end to end depth on one well chosen feature is what a hiring manager actually respects, and the reason is not sentimental. Authentication, failure handling, deployment, testing, reasoning about trade offs: those skills transfer completely to any other domain once they have been genuinely earned in one.

The same principle applies further up the ladder. When a mid-level engineer tells me they are choosing between Kafka, RabbitMQ and Amazon Kinesis for an event driven system, my first question is never which one is better. It is what does the requirement actually demand. If several independent consumers must each receive and process the same event in full, and must be able to replay history when one of them is rewritten, that points naturally towards Kafka's model. If the need is straightforward work distribution with per message acknowledgement, that is a different shape of problem. Choose infrastructure for reasons, not for fashion. The reasons are what you will be asked to defend in the room.

### Geography is part of the market

One last piece, and it is the one people learn most painfully: technology demand is not universal. It is regional. Reading your market means reading geography too. An engineer moving from Nigeria to the United Kingdom will find strong and sustained demand for Java, and comparatively thinner demand for Node in several segments of that same market. Knowing what to learn is only half the work. Knowing where that skill is valued is the other half.

A brilliant umbrella, beautifully made, will not sell in a desert. Nothing is wrong with the umbrella. Everything is wrong with the street corner. Position yourself where what you have built is already needed.

## Part Three: Depth over collection

Assume you have chosen your industry and your first stack. Here is where the next mistake arrives. People decide they now need to learn everything. Java. Spring Boot. Docker. Kubernetes. AWS. Kafka. Redis. Terraform. React. TypeScript. GraphQL. They build a checklist of forty technologies and six months later they are exhausted. Not because engineering is difficult, but because they are trying to swallow an entire industry in one sitting.

Suppose you wanted to become the best heart surgeon in the world. Would you begin by studying every medical specialty at once? Dentistry, neurology, psychiatry, dermatology, orthopaedics? Of course not. You would master one area deeply, and then build outward from a position of genuine authority. Engineering works exactly the same way. If you have chosen fintech, do not try to build Monzo or Revolut or Wise. Build one feature. Money transfer. Authentication. Exchange rate calculation. Transaction history. Then go deep: understand why it exists, who uses it, what happens when it fails, how it scales, how it is monitored, how it is secured. Something interesting happens when you go that deep. You stop writing code and you start thinking like an engineer.

One of the biggest mindset shifts of my career came when I realised that companies do not pay us because we know Java. Nobody wakes up and says, we need someone who knows Java. What they are actually saying is: we have a business problem. Our customers cannot make payments. Our website crashes on Black Friday. Our transactions are too slow. Our fraud detection is not working. Our users are leaving. The language is the tool. The value is the problem solved.

That is why I tell people not to introduce themselves by their programming language. There is an enormous difference between I am a Java developer and I build resilient payment systems capable of processing thousands of transactions per second. Complete this sentence for yourself, and notice how much harder it is than the usual one: I want to become known for. Not I want to learn. I want to become known for. Whatever follows is the beginning of your professional identity.

I learned this far later than I wish I had. Early on I thought success meant collecting technologies. Every new framework, I wanted it. Every new library, I wanted to try it. Then I noticed that the engineers everyone respected were not respected for knowing everything. They were respected because they understood something deeply. When they spoke, people listened. When production went down, people called them. When difficult architectural decisions had to be made, they were in the room.

> **Breadth gets you into the conversation. Depth keeps you in the room.**

Early in a career, exploration is healthy. Try things. Discover what excites you. But there comes a point where you must stop exploring and start mastering, and that is where careers accelerate. Have you ever looked at someone on LinkedIn and thought that person knows everything? I used to. Then I started meeting them. They do not know everything, not even close. They know one area exceptionally well, and because of that, everything adjacent becomes cheap to learn. That is how expertise compounds.

Think of a career as a tree. Most people spend their energy growing branches. Another framework, another certification, another language, another cloud provider, another course. More branches, more branches, more branches. Nobody sees the roots, and the roots are where depth lives. The deeper the roots, the stronger the tree, and the more branches it can carry. If your roots are shallow, every new technology feels like a threat, every trend feels personal, and every interview feels impossible, because there is nothing solid underneath.

## What interviews actually test

I have sat on many interview panels, and I want to correct something. People think interviews are knowledge tests. They are not. They are thinking tests. I have interviewed people who could not remember half the methods in Spring Boot and they got the job, because they knew how to think. They asked good questions. They broke a complex problem into smaller ones. They communicated clearly. They reasoned out loud and let me watch. Engineering is that. The code is simply the final output of good thinking.

Candidates sometimes apologise to me: I have forgotten the syntax. My answer is always the same. I can teach syntax. I cannot teach thinking. Google will remind you of syntax in thirty seconds. Nothing on the internet will hand you judgement.

So stop measuring progress by tutorials completed. Ask instead: am I becoming better at solving problems? Am I asking better questions? Do I understand why systems are designed the way they are? Am I becoming someone people rely on? Companies do not hire walking documentation. They hire reliable problem solvers.

# 02

PART TWO · THE MID-LEVEL ENGINEER

## Comfort is the most expensive thing an engineer can buy.

# Part Four: The most dangerous stage of your career

Let me be honest with mid-level engineers, because I think this stage is more dangerous than being a beginner. People assume beginners have it hardest. I disagree. Beginners know they are beginners; they know they must learn. The mid-level is where comfort quietly creeps in. You have a salary. You know your way around the codebase. People trust you. Git no longer frightens you. Production deployments no longer frighten you. Life is comfortable. And comfort is one of the great enemies of growth.

There was a season in my own career when I lived exactly this, and looking back, what I did about it changed my life.

## The story I do not usually tell quickly

There was a period when I was working as an IT instructor, and my responsibility was to train prospective software engineers. I loved it. I genuinely loved it. One thing I have been blessed with is the ability to teach: taking something that looks complicated, breaking it apart, finding the one relatable example, until somebody's face changes and they say, ah, I finally understand. That moment never gets old. If you have ever taught anyone anything, you know exactly what I mean.

Every day I was impacting lives. People came into my classes confused and left confident. People who had never written a line of Java were writing Java. People who had decided software engineering was not for them started believing in themselves. It was a period when many were running away from Java because everyone was chasing the next shiny thing, and every week another article announced that Java was dead. I kept telling people the same thing: trends come and go, fundamentals stay, and if you understand the fundamentals you will always be employable.

From the outside, everything looked right. Students were getting jobs. Families were changing. If you had looked at my career from a distance you would have said, this is exactly where he is supposed to be.

But something started bothering me. Not suddenly. Gradually, and very quietly. The people I trained were leaving. They were joining engineering teams, building products, sitting in architecture meetings, solving production incidents, working with product managers, designing systems used by thousands and sometimes millions of people. And every evening, I was preparing tomorrow's lesson.

Do not misunderstand me. Teaching is one of the noblest professions there is. I still teach today; it is part of why this Blueprint exists. Teaching was never the problem. The problem was that my own blueprint had stopped evolving. I had become very good at one chapter of my career, and I was not writing the next one.

Then one day I asked myself a question that made me deeply uncomfortable, and I would like you to sit with the same one: if I keep doing exactly what I am doing today for another five years, who

will I become? Not what will I earn. Not what title will I hold. Who will I become. That question changed everything.

> **Sometimes the biggest danger in a career is not failure. It is succeeding inside a blueprint that no longer takes you where you want to go.**

Leaving was not easy. When you are good at something, people expect you to keep doing it. They celebrate you. They tell you that you are amazing at this, that you have found your calling, that you should not leave. But I knew there was another engineer I had not become yet. There were production systems I had not built, customers I had not served, incidents I had not handled, architectures I had not designed, mistakes I had not yet made and lessons I had not yet earned.

So I left. Not because I hated teaching, but because I wanted to become the kind of engineer who solves real problems every day. I wanted to know what happens when a banking system goes down at two in the morning. I wanted to know what it feels like when thousands of customers depend on something you built. I wanted software beyond tutorials, because production is where software tells you the truth.

That decision exposed me to problems I could not have imagined. It forced me to think differently. It humbled me, stretched me, made me deeply uncomfortable, and eventually made me better. I am telling you this because someone reading it is exactly where I was. You are doing well. Everyone thinks you are successful. And something inside you keeps whispering that there is another level. Do not ignore that voice. Sometimes that voice is your next blueprint calling.

I am not telling anyone to resign on Monday morning. This is not about changing jobs for the sake of it. Growth does not always require a new employer. Sometimes it requires a new project. Sometimes ownership. Sometimes mentorship. Sometimes learning distributed systems properly. Sometimes leading people instead of only writing code. Sometimes saying yes to the responsibility you have been quietly avoiding. The question is not should I leave. The question is am I still evolving. Those are very different questions.

## Blueprint paralysis

I have mentored hundreds of engineers, and one pattern stands out: the ones who grow fastest are not the smartest. They are the ones willing to become beginners again, deliberately, every few years, in situations where they do not have the answers.

The opposite of that is what I call blueprint paralysis, and it is one of the great killers of mid-level careers. It happens when you know too much about too many possible directions. Someone says Go is the future, so you start Go. Three months later someone says Rust is replacing everything,

so you switch. Then AI becomes unavoidable, so you are learning Python. Then you are told you need Kubernetes, then GenAI, then MCP, then agentic systems, then another framework, another certification, another course. Five years pass. You have been extraordinarily busy. And if someone asks what you are exceptionally good at, you cannot answer.

What I did instead was choose an anchor at every stage of my career. An anchor is the thing that does not move while everything else changes. For me, Java became an anchor. Backend engineering became an anchor. Distributed systems became an anchor. System design became an anchor. Everything else I learned, cloud, Kafka, Docker, Kubernetes, microservices, connected to that anchor. That is why the learning compounded instead of competing.

Depth compounds. Every new layer is easier because it rests on something solid. That is how senior engineers are actually made: not by chasing everything, but by connecting everything.

When someone walks into an interview with me, I am not trying to find out whether they can memorise APIs. I am trying to answer one question: can I trust this person with difficult problems? Difficult problems do not arrive with Stack Overflow answers attached. They require thinking, judgement, communication, trade offs, experience. That is what companies pay for. So if you are mid-level today, here is the challenge: stop asking what should I learn next, and start asking what should I understand better. Those two questions produce completely different careers.

# 03

PART THREE · THE SENIOR ENGINEER

## Every blueprint expires. Burn it down before it burns you.

# Part Five: Every blueprint has an expiry date

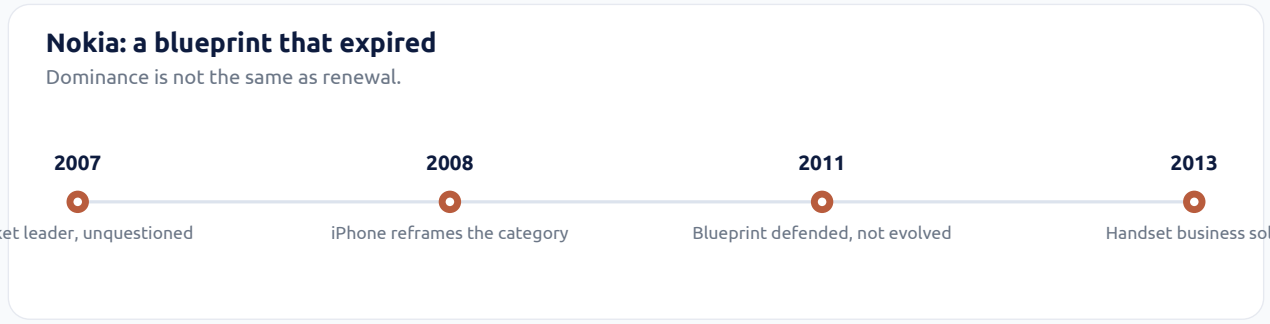
Now to the senior engineers. The architects. The technical leads. The engineering managers. The principal engineers. The people everyone calls when production goes down, whose opinions carry weight, who spent years earning the respect they have. I want to speak to you with the utmost respect, because everything that follows I say to myself first.

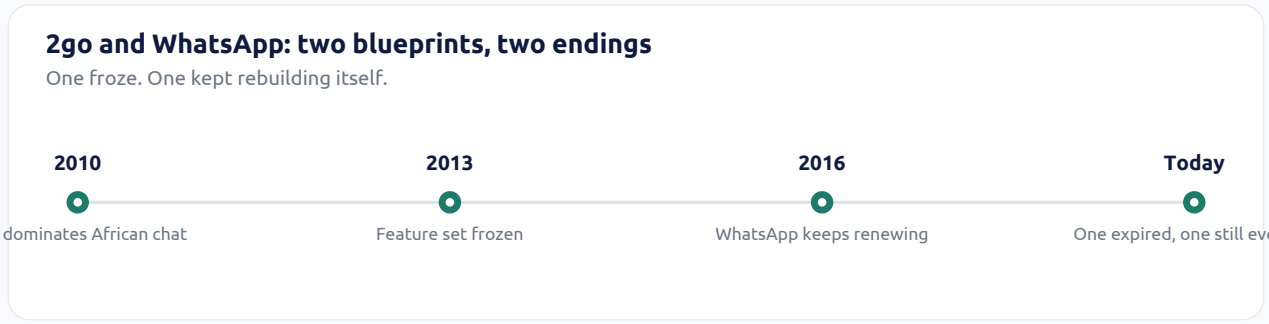
One of the hardest truths I have learned is that every blueprint has an expiry date. Not because it was wrong. Not because it failed. Because the world changed.

Five years ago, how many of us discussed AI assisted development daily? How many organisations had platform engineering teams? How many people were building with large language models? How many of us expected an AI to review our pull requests? Things change, and they change faster than we expect. The biggest mistake experienced engineers make is assuming that what made them successful will keep them successful. Sometimes it will. Often it will not.

## Two cautionary tales

The first is Nokia, and I tell it not as a story about failure but as a story about confidence, because confidence without evolution becomes dangerous. For years Nokia owned the mobile phone industry. If you grew up in Africa, and especially in Nigeria, you owned one or someone in your family did. They were everywhere. They hardly broke. The batteries lasted for days. You could throw one at a wall, pick it up, and carry on with your call. People trusted Nokia. Engineers respected Nokia. Investors believed in Nokia. Their blueprint worked.




Then Apple introduced the iPhone, and plenty of people inside Nokia dismissed it. It has no physical buttons. The battery will not last. The screen is too fragile. Some of those criticisms were factually correct, which is exactly what makes the story so uncomfortable. Nokia's former chief

executive is widely reported to have said, we did not do anything wrong, but somehow we lost.

Nobody became stupid overnight. Their engineers did not suddenly forget how to build. Their blueprint simply expired, and because they trusted yesterday's blueprint to solve tomorrow's problem, they slowly became irrelevant.

The second story is closer to home for many of us. Before WhatsApp captured the continent, 2go was how young Nigeria talked to itself. It worked on the phones people actually owned, it cost almost nothing to run on a thin data plan, and it built entire social worlds inside chat rooms. If you were a teenager in Lagos or Ibadan in that era, your friendships lived there. By any reasonable measure, 2go had won its market.

Then the ground moved. Smartphones spread, data got cheaper, and WhatsApp arrived with something simpler and more reliable that worked the same way on every device your contacts owned. 2go had the users, the brand and the head start. What it did not do, quickly enough, was evolve its blueprint for the world that was arriving rather than the world that had made it famous. Within a few years the rooms emptied. WhatsApp, meanwhile, has spent every year since rebuilding itself release after release, refusing to treat early dominance as a finished product.

Nokia globally, 2go regionally. The same lesson twice. In neither case was the danger a shortage of talent or a shortage of success. The danger was the failure to burn down a blueprint that had expired, at the exact moment courage was required to do it.

> **The danger is never that you are bad at what you do. It is becoming so good at yesterday's blueprint that you stop building tomorrow's.**

Sometimes the bravest thing an engineer can do is not learning a new framework. It is letting go of an old identity. Experience teaches us patterns, which is good, but the same patterns can stop us from seeing new possibilities. Experience becomes a filter. Instead of saying tell me more, we start saying we have tried that before. I tell engineering leaders this often: your experience should be a foundation, never a prison. A foundation lets you build higher. A prison keeps you where you are.

You have heard the sentences. We do not need cloud. Microservices are hype. AI will never write production code. We have always done it this way. Our monolith has worked for twenty years. Some of those may even be true in context. But that is not the real question. The real question is whether we are still learning, because the day we stop learning is the day the blueprint starts expiring.

I remember when cloud became mainstream. Many experienced engineers resisted, and not because they were unintelligent. Some of the sharpest people I have known questioned it, and their concerns were genuine: security, latency, cost, control. Those were legitimate engineering discussions. But the engineers who stayed relevant were not the ones who defended the past forever. They were the ones who became curious. Instead of this is wrong, they asked what can I learn from this. Completely different mindset, completely different decade.

The same thing is happening now with AI. Some are afraid. Some are dismissive. Some are giddy. My advice is simple. Do not panic. Do not worship it. Do not ignore it. Understand it, because understanding always beats opinion. If AI genuinely changes software engineering, I want to understand it. If it does not, I will understand why. Either way I win.

Even after sixteen years, I still deliberately put myself in rooms where I am the least experienced person present. I still buy courses. I still attend conferences. I still ask junior engineers questions and mean them. I still build things I have never built before. I do it because I never want my title to become bigger than my curiosity. The greatest compliment I can receive is not that I am the smartest engineer in the room. It is: you are still learning. That tells me my blueprint is still alive.

Does evolving mean throwing everything away? Absolutely not. When I say burn down your blueprint, I do not mean burn down your experience. Keep the experience. What you burn are the assumptions, the habits, the limitations, the beliefs that no longer serve you. You do not demolish the foundation. You renovate the house. You add a floor. You knock through one wall because the family has grown. You redesign the kitchen because your life is different now. Same house, better version of it. That is what career evolution actually looks like.

One question for every senior engineer reading this: when did you last change your mind about something important, not because somebody forced you, but because you learned something new? If you cannot remember, it may be time to revisit the blueprint. The engineers still respected at fifty, sixty, seventy are not the ones who knew the most technologies. They are the ones who never stopped being students. That is what seniority actually is. Not knowledge. Learning.

## Part Six: Bringing it all together

Beginners, mid-level engineers, senior engineers. At first glance they are facing different problems. They are not. They are all asking the same question. What is next? The beginner asks where do I start. The mid-level engineer asks how do I grow. The senior engineer asks how do I stay relevant. Three questions, one answer: a blueprint.

In sixteen years, the engineers who have impressed me most were not the smartest or the fastest, and rarely the ones from the best universities. They had one habit in common. Every few years, they reinvented themselves. Not because they were forced to. Because they chose to.

Imagine your career as a book. Every chapter is a season. The beginner chapter. The junior chapter. The mid-level chapter. Senior. Leadership. Architecture. Mentoring. In a good book, every chapter differs; the story moves and the character grows. Now imagine buying a novel where chapter one is identical to chapter two, and chapter three, and every chapter to the end. You would stop reading, because nothing changed. So here is the uncomfortable question. Has your career become that book? Have you been living the same chapter for five years with a slightly different salary?

I chose the title The Software Engineer Blueprint deliberately, because blueprints are not meant to be framed and admired. They are meant to be used, improved, updated and sometimes discarded. Ask any architect. The blueprint for a bungalow is not the blueprint for a skyscraper. Both are correct. They are correct for different destinations. The blueprint that got you your first engineering job cannot be the one that gets you to Staff Engineer. The blueprint that made you a Senior Engineer will not prepare you to be an Engineering Director. Growth demands redesign.

Your career does not change because time passes. It changes because decisions are made. Five years from now you will not accidentally become exceptional. It will not happen because you wished for it, or watched another video, or because LinkedIn told you which technology is hottest. It happens because one day you decide, I am going in this direction, and then you keep walking.

People ask me how I got here, and the honest answer cannot fit in a sentence, because where I am is the accumulation of many small decisions. Choosing to keep learning. Choosing difficult projects. Choosing uncomfortable conversations. Choosing to ask questions. Choosing to fail, and choosing to start again. That is all growth really is. Small decisions, repeated consistently.

## Three questions, before you close this page

Write them down properly, on something you will see again.

First: what industry do I want to become known for? Not where do I want to work, but what problems do I want to solve?

Second: what is one thing I need to stop doing? Perhaps chasing every trend. Perhaps procrastinating. Perhaps comparing yourself to everyone else. Perhaps staying too comfortable.

Third: what is one decision I will make before this week ends? Not next month. Not next year. This week, because information without action changes nothing.

One of the things I love about software is that it is brutally honest. It does not care about your intentions, how long you worked or how passionate you are. It either works or it does not. Life is surprisingly similar. Life responds to what we build, not to what we intended to build.

When I look back at my own journey, from Nigeria to working across different countries to becoming a Principal Software Engineer, there was never a magical moment. No lottery ticket. No miracle opportunity. No secret shortcut. There were only seasons where I chose to rebuild myself.

When I needed technical depth, I rebuilt. When I needed architecture, I rebuilt. When I needed cloud, I rebuilt. When I needed leadership, I rebuilt. Every version of me required letting go of an older version of me, and that is hard, because we become attached to what we are already good at. Growth has always demanded courage.

> **Never become so attached to the engineer you are today that you refuse to become the engineer tomorrow requires.**

Tomorrow's problems will require a different version of you. If this piece has done anything, I hope it has given you clarity. Not all the answers, because nobody has all the answers. Clarity. Clarity creates confidence, confidence creates consistency, consistency creates mastery, and mastery changes lives.

## Why LytQuest exists

LytQuest was not created because the world needed another coding bootcamp. There are already enough courses, enough videos, enough documentation, enough tutorials. LytQuest exists because I wanted to build the learning environment I wish I had earlier in my own career: a place where people do not only learn syntax, but learn how engineers think, how they communicate, how they solve problems, how they design systems, and how they grow.

Getting your first job is wonderful. Building a career is an entirely different journey, and almost nobody talks honestly about the second one.

If you have read this far and something in you is saying, I want someone to walk with me while I build my own blueprint, then I would genuinely like to work with you. I cannot promise shortcuts, because there are none. I can promise direction. I can help you avoid mistakes that took me years to learn. I can challenge your thinking, mentor you, and help you build the kind of work that reflects how software is actually built in the real world. And if now is not the right time, that is completely fine. My aim was never only to invite you to a programme. It was to make sure you think differently about your career than you did before you started reading.

> **The future does not belong to the engineer who knows the most. It belongs to the engineer who never stops learning.**

Build your blueprint. Evolve your blueprint. And when the time comes, have the courage to burn it down so you can build an even better one.

ABOUT THE AUTHOR

## Akintomiwa (JBoss) Abolade

Principal Software Engineer, NatWest Group. Sixteen years building software across education, logistics, banking, fintech and agriculture, with a career spanning Nigeria, the United States and the United Kingdom.
Founder of LytQuest, an engineering career ecosystem.

---

## Welcome to LytQuest.

Join the LytQuesters community and build your blueprint alongside engineers who are doing the same work you are.

**lytquest.com**

*Build your Blueprint. Evolve your Blueprint. Build Better.*