How do I get more technical?

Tips for leveling up your technical chops and being more effective at work

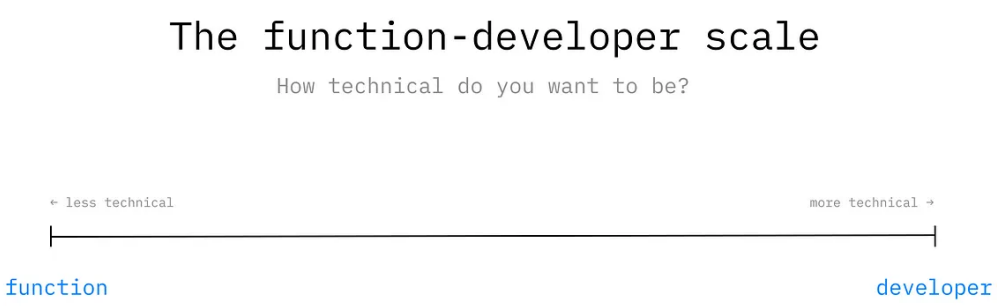
Justin
Jul 12, 2022

One of the defining problems of our time, if I may say so myself, is technical literacy: technology is eating up more and more of what we do, and we seem to be getting *worse* at understanding it. Maybe you work at a tech company and want to understand what your engineers are talking about; or you could be employed at a hedge fund, trying to figure out what the newest infrastructure company going public actually does. No matter who you are, being a non-engineer in an engineer's world can be very frustrating. And it's hard to find a way out.

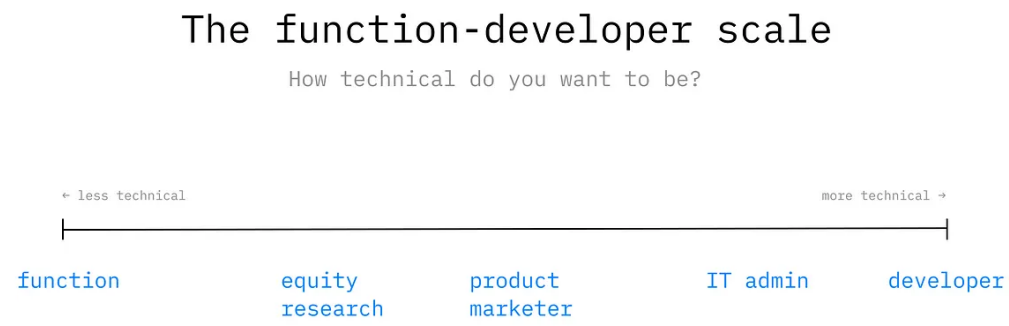
So how do you get more technical? What should you even be learning about? This post is going to walk through what it means to really get more technical, what worked for me, and a few suggestions as to what could work for you.

What does technical literacy mean? The function-developer scale

The first area where people stumble in getting more technical is defining *what technical actually means*. Technical knowledge exists on what I like to call the **function-developer scale** [1]:



I call it the *function-developer scale* because it goes from being completely non-technical – the *functional employee* like the average salesperson, marketer, recruiter, etc. – to being so technical, you will argue about it on the internet – the *developer*. Now unless you want to be in a function or you want to be a developer, where you want to end up is going to be somewhere in the middle. And indeed, if you look at job roles that relate to technical stuff, you'll find they land in various places on the scale:



So the first and most important part of getting more technical is asking yourself: *how technical do I want to be?* If you want to be working with code on a day to day basis, you're going to need to learn how to program; but if your goal is to be able to speak semi-intelligently with developers, you may only need to move up 20-30% on the scale.

To make this a little bit more practical, here's a table of where, at least in my experience, different roles need to be on the *function-developer scale*. Keep in mind that every job, company, and person is different, and I am simply too old to tolerate comments critiquing this for missing some niche you came up with because you had already resolved to get angry today.

Role	Goals	Scale Location
Product Marketing	• Understand the product • Communicate with technical audiences	5.5
Financial Analyst	• Understand business models • Analyze what technical companies do	4
Recruiter	• Speak to and recruit developers • Understand the product	3
Consultant	• Understand business models • Identify where technology can help	4
Security Analyst	• Implement threat detection and prevention • Understand device and account security	7

Each of these roles requires a different level of technical literacy to excel. There are also **multipliers** (is this a game?) that increase the level of technical literacy you'll need, no matter what role you're in:

- **Does your company sell to developers?** If so, apply a large multiplier. Every job is going to interface with technical material, and the more you understand, the better you'll be.
- **Is your company early stage?** If so, apply a medium multiplier. With smaller teams and fast growth, you'll be more effective if you can stretch your expertise and be more self-sufficient, even with something as small as fixing a typo on the website.
- **Do you want to grow into a more technical role?** If so, apply a medium multiplier. Joining companies to grow into more technical roles is a common path, but it's not going to happen without putting in the work.

So before you ask *how* you can get more technical, you need to ask yourself *why* you want to get more technical. Some reflection on the *why* will put you in a much better position to answer the *how*.

Breaking down what you need to learn: "being" technical

After diligently reading the first section and some existential introspection, you now have a good idea of how technical you want to be. But what do you actually need to learn? What does it mean to *be* technical for your own goals? Like any area of knowledge, there's the theoretical and then there's the practical.

The theoretical and the hypothetical

Part one of being technical is having a theoretical understanding of how the technology you work with actually, well, works. How many of the following questions can you answer confidently?

1. *How does the internet work?*
2. *What's a server?*
3. *What is the cloud?*
4. *How do relational databases work?*
5. *What's a frontend and what's a backend?*

The answers to any of these individual questions might not surface in your day to day job, but without them you're powerless to understand the larger picture. As a financial analyst, you simply cannot understand what Datadog does – and why or why not it's worth investing in – without understanding servers and the cloud. And as a product marketer at a company like AWS, you're going to struggle to write compelling copy for developers without grasping the world of databases.

What makes this doubly difficult – which is true of all knowledge – is that these theoretical foundations are **layered**. They build on top of one another. It's hard to answer the question "*what is the cloud?*" without understanding what a server is or how the internet works. And that nested, entangled knowledge graph makes learning this from scratch all the more daunting. If you're found yourself reading some technical blog and not understanding every 3rd term, you know what I'm talking about.

A strong grasp of the theoretical will allow you to:

- Speak intelligently with developers about topics relevant to your job
- Understand the role of specific technologies in your product and workflow
- Place companies and competitors relative to where you work
- Cut through fluff and noise like AI, Blockchain, trading stocks, etc.

Theoretical knowledge is, of course, only theoretical. Once you get some practical experience you'll really be cooking with gas.

The practical and the tactical

Part two of being technical is being able to use that theoretical knowledge to do some real , bona-fide things. Some of these will involve – alert – writing basic code, but not all do. My favorite illustrative example:

> *Imagine you're a marketer at a fast growing startup. You're running through the homepage as a quick check before a launch tomorrow, and you see an egregious typo. What do you do?*

Most marketers will be completely powerless, since they don't know how to code. You'll have to file a ticket with the engineering team and hope they get to it when they're free; or hope there's some sort of CMS you can use. But what if you could just fix the typo yourself? That would be very useful. And the great thing about this example is that you don't *really* need to know how to code to fix this – you just need to know how git works.

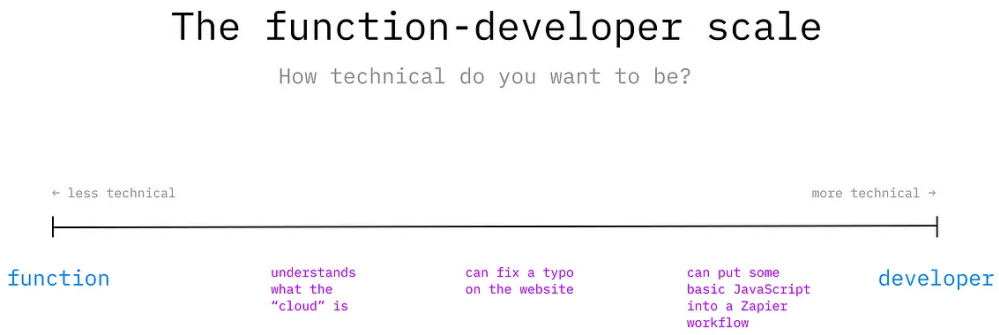
There are a lot of these situations that arise, especially at smaller companies. You might be gathering metrics every week by hand, and wish you could automate it to save yourself time. Maybe your team wants to send any tweets that mention your company into your Slack, but nobody feels comfortable using Zapier. There are just so many cases in which being practically technical – even if you're not a full on developer – is useful.

Just like being a developer is about a lot more than just "knowing how to code," the same is true of being practically technical. Having an understanding of **how your team's toolset works** will allow you to do things with that toolset even if you can't code. If your blog runs on a CMS like Contentful, you might be able to change copy on the site without writing code, but you may need to learn how to

deploy the site from Netlify. The specific tools don't matter; what matters is identifying this "gray area" of technical expertise that most people shy away from.

This is doubly relevant as more and more so-called no code and low code tools are entering the market. This discussion is much more complex than a single paragraph, but suffice it to say that the "dangerously technical" persona today has many, many more tools at their disposal. Airtable can act as a basic database, Retool can get you internal frontends with minimal code, and Integromat can automate basic workflows. Practical technical "plumbing" knowledge may not be hot, but it's useful.

Back to the *function-developer scale*, you can start to swap out job roles for specific levels of technical expertise:



The more technical you get, the more capable you are; but the more time it takes to get there.

Methods and resources for getting more technical

This last section is going to be the worst one. I started Technically two years ago because I couldn't find any great resources on the web for getting more technical, and things haven't changed very much. If you want to learn how to code, there's *a ton* out there; but if you want to end up anywhere lower on the *function-developer scale* than developer, it's slim pickings.

To get more technical, you need to **do more technical things**. You've learned new subjects before, so you already know that theoretical knowledge without practical import is boring, frustrating to learn, and fades quickly. If there's one "hack" or shortcut that I've learned as a developer, it's that you need to *do* to *learn*.

1) Learn how to code

Call me contrarian (please don't), but I think basically everyone should understand how to write basic code. It's silly to do a PhD in French literature without speaking French, and it's silly to learn about the universe of software without speaking software's language. You don't need to spend 3 years and become a software engineer; a basic understanding of one popular language (I'd recommend JavaScript or Python) will go a long way.

There are tons of resources for getting started with basic code. I started with Codecademy, and eventually graduated to Lynda (now LinkedIn Learning) for more use case specific packages and frameworks. **The best tip** I ever got for learning to code: have a specific project in mind, like an app you've been wanting to build, or a process you want to automate. Otherwise it can feel like you're swimming in the overwhelming, endless, meaningless current of infinite possibilities.

2) Do things at your company

Being employed, gainfully or otherwise, at a bona-fide company is a luxury when it comes to getting more technical. There are entire teams just an email or Slack away who are spending their days doing technical things! Squint and you'll find plenty of things your team is doing inefficiently that could benefit from code, a better tool, or even just better communication with a technical team.

One thing I struggled with when getting more technical was **finding useful examples**. Reading about how React works was too theoretical. But once I started working at a company that actually *uses* React, I had more chances to understand what it really does, why we used it, and what it meant to work with. The same thing applied to understanding AWS – much easier when there's a

running account you can look at! – as well as basic networking concepts, Kubernetes, etc.

3) Find a developer friend

One of the most useful parts of being employed is having access to a **team of engineers**. They might be bothered by your obsequious servility at first, but like all great monarchs, they'll eventually get used to it. Being able to ask them questions can come in the clutch! To go a level deeper, you should find a friend who you trust – and is a developer – to act as a sort of guide for you. As I was getting more technical, I would often bounce questions off a few technical friends and see what came back (sometimes it was "fuck off").

Aimless conversations aren't as useful as structured ones in this arena. If you're trying to understand a specific topic, come armed with specific questions in advance, and focus on areas that you're weakest in. Do some footwork to come prepared. And of course, don't be a taker – if you find yourself getting a lot of value out of someone, buy them something nice :)

4) Sleuth out the best content

At one point or another, you're going to have to resort to searching on Google for a set of terms you don't understand. And if you're reading this, you've probably already tried that and gotten less than satisfactory results. The search rankings are dominated by sites like ZDNet and TechTarget, filled to the brim with ads, videos, and otherwise obstructing views. These explanations can be decent but generally rely on a lot of previous knowledge that you may not have. And man, why are there *so many* ads?

In my experience, the best content for explaining tech concepts sits in two places:

1. **YouTube** – a lot of shit, but can be a goldmine for visual explanations of tech concepts. The most useful videos are not always the flashiest or best produced ones.
2. **Company-specific blogs** – harder to find, but generally the best content for any individual concept comes from a company that makes money off said

concept. My favorite examples are this Duo Security post about SAML, and Cloudflare's guides for understanding their company.

Like anything, you'll need to refine your search skills and develop an intuition of when a piece of content is worth your time or not.

5) Read Technically (lol)

I started Technically two years ago to solve the exact problem that the past few pages have laid out. It's meant to be a broadly accessible resource for learning how to be more technical without learning how to code, focused on practitioners at tech companies, banks, etc. Technically covers topics as basic as *What's an API* and as advanced as the details on specific ETL tools.

I am admittedly biased here but I (and 30,000+ other people) really do think that Technically is the best resource on the web for getting more technical.

🙋 *What have you found most helpful in your journey towards becoming more technical? Chime in with a comment below.*

---

1 I made this up at the time of writing

---

12 Comments

Write a comment...

Molly Jul 12, 2022

This is super helpful. One piece of feedback: using the term "grandma" to mean non technical is pretty outdated (also gendered for no reason). It assumes a grandma is the opposite of a developer when someone can be a software engineer and a grandmother. Would love to see a better way to explain this spectrum because the content in this article is otherwise excellent!

♡ 10 Reply Gift a subscription Collapse •••

5 replies by Justin and others

Eberhard Moog Jul 20, 2022

Hey Justin, great article and I read some of your previous posts.

Looking at your content I wonder where to start best? Would it makes logical sense to start with your first article and work my way up? I am not sure about it when I look at the titles of you posts.

Thanks for your advice.

♡ 1 Reply Collapse •••

10 more comments...