Git Cute

A SOFTWARE ENGINEER'S GUIDE TO SENIORITY

Jocelyn Harper

Git Cute: A Software Engineer's Guide to Seniority

By: Jocelyn Harper

Published by Jocelyn Harper

First edition: 2021

Visit the author's website: jocelyn-harper.com

eBook ISBN: 978-1-63760-361-1

Dedicated to Brian, Bobby, Carina, Erika, Kelsey, Nuru, and Sidney who have pushed me forward, professionally and personally, when I wasn't convinced I could finish.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Foreword

Over the past year, I have enjoyed watching Jocelyn grow her following and share her invaluable knowledge of the tech industry, and A Software Engineer's Guide To Seniority is no exception.
When I first learned of Jocelyn's plan to publish a book about seniority, I jumped at the chance to pre-order it. Between Jocelyn's podcast, Git Cute, which I've long admired, and her authenticity on social media I had every confidence that her book would be a valuable resource.

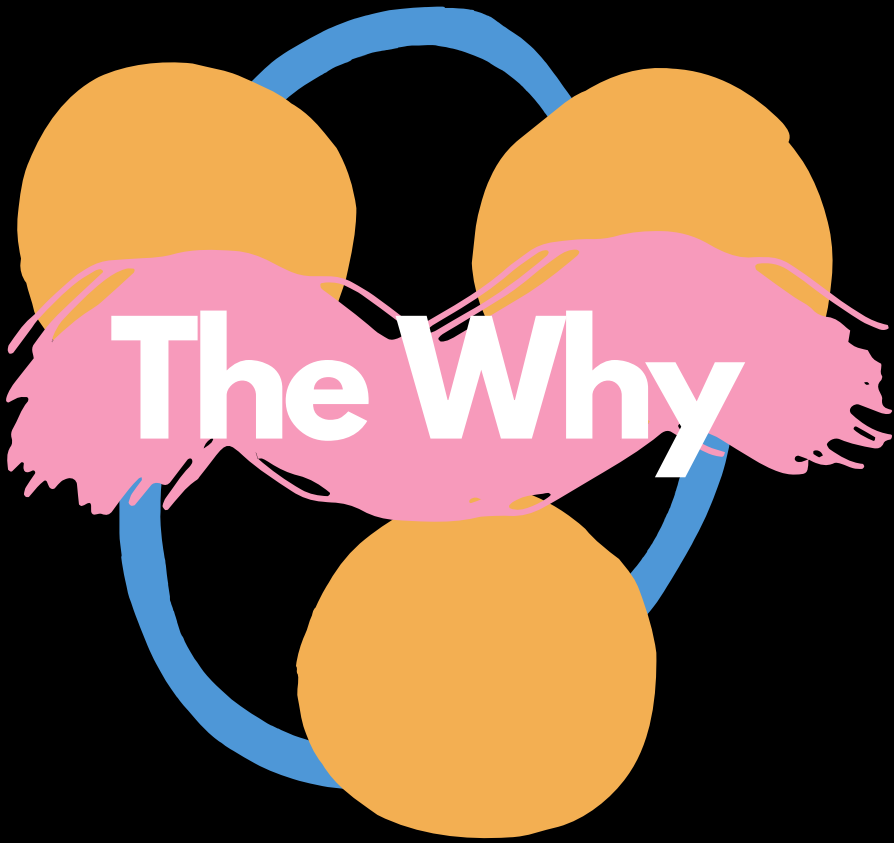
As an engineer who has never been promoted, A Software Engineer's Guide To Seniority is the resource I've been missing. I've worked in tech professionally for nearly six years, at three separate companies, and have never been able to take the next step to becoming a senior engineer. While I am not ashamed of this, I was always lacking the resources necessary to advance my career. Needless to say, I am highly anticipating this book as it's the long-lost resource I've always needed.

Foreword

A Software Engineer's Guide To Seniority reads less like an instruction manual and more like advice from a trusted friend. In it, Jocelyn shares her knowledge about advancing her development career, the lessons she's learned along the way, and tips you may not have thought of, such as using social media as a networking tool.

I am humbled and thrilled to be writing the foreword to this book and I hope it helps you as much as it will help me. Thank you, Jocelyn, for gracing the tech industry with your knowledge and expertise.

Software Engineer, Spotify


The Why

I first understood my love for tech when my mother bought me a Gateway computer in the summer of 1998. It was a gift for passing the 4th grade with all A's and honors. We were not a rich family — in fact, poverty is appropriate to describe the majority of my child and adulthood — but my mother would put herself in poor financial decisions if she knew it would benefit me.

I went with her to the rent-and-buy store to pick it up, and once we got home, we set it up on the floor of my older sister's room. I wouldn't get a desk until next month, but I was perfectly content sitting on the floor for hours. I would sit there and explore the software — Encyclopedia Britannia, Solitaire; I was absolutely hooked.

Then one day my father brought in the mail after work. Situated between a couple of flyers and bills was an AOL CD-ROM. I already knew what the Internet was; we

had it in school and were learning how to use AskJeeves.com — a prehistoric predecessor for the search engine — for research. I quickly escaped to my sister's room with it and began to figure out what I needed in order to get connected.

The Internet opened up an entire world to me that I didn't know existed. It was through browsing unsupervised on AOL that I found a group of teenage girls that put their creativity into making their own layouts and websites on Express and AngelCities. It enthralled me. How did they get everything so aesthetically pleasing? The scrollbars matched the complimentary colors in their layout that they had spent hours carefully crafting in Jasc's Paint Shop Pro. There were sections for custom designed HTML tables, an area for custom pixel GIFs that were an entire feat on their own. I needed to know how these girls did all of it, and I needed to know immediately.

The resources for learning how to build your own website was finite, especially material that 9 year old would find interesting and also understand. Lucky, I was able to find Lissa Explains, a darling nugget of a 90s nostalgia that that taught me the basics of HTML, CSS, and even a little bit of JavaScript with a neon color palette mixed in.

It was on that website that I cultivated and nurtured my love for creating. I loved knowing that I was building something from the ground up. At one point I even upgraded to self hosting and was teaching myself PHP when my love for the custom website world phased out with the introduction of MySpace. I learned HTML, CSS, and remedial JavaScript at the age of 9 and sat on that knowledge until I was 26 and at my wit's end at a dead-end office receptionist job for a prominent home builder in Delaware. I was bored out of my mind and severely underpaid.

My best friend Kelsey - who is still my best friend (sup, Kels?) — had just started a coding boot camp within our state and persuaded me to at least apply. So, I did, and the rest is a 5 year history of technical mishaps and wins, interviews, panels, keynotes, podcast episodes, and now, a book.

Why am I writing this book?

It is the natural next step with everything else that I already have offered as far as truthful content about my experience as an employed software engineer this far. Also, because no one has written it yet.

We are fortunate to have a lot of great books on how to knock a technical interview out of the park and how to increase your productivity as an engineer. There's even a book where it details skills and tricks that are not available to the wider public because no one has sat down to put them together into a comprehensive guide, and I figured,

why not me? I got my first senior position just 3 years shy of my anniversary of being a software engineer. I quickly got another great with a Fortune 500 company. The majority of my career I worked for Fortune 500 companies, and although according to people on r/cscareerquestions, these companies and the technology they produce are valid because they server hundreds of millions with billions of transaction each day.

No one told me how to become a senior software engineer. I just knew that it was the next milestone in my career and that I needed to figure out how to become one somehow. There were a lot of bumps and bruises along the way of figuring it out, and honestly, I fell into my first senior engineering job with a tiny bit of skill and a whole lot of luck.

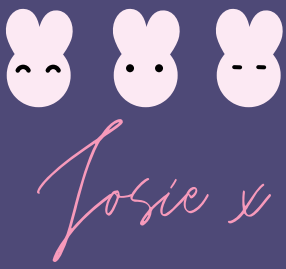
People are not being honest with you if they do not acknowledge how being in the right place at the right time has affored them better projects, jobs, and opportunities. I call it luck and others may refer to it as

This book is the comprehensive set of my experiences, from networking to finding the perfect senior position to interviewing candidates for your team to mentoring and more

I am not claiming to know everything about obtaining a position and exceling within every company. As you read the book, you will come to understand that this is highly subjective to each company, recruiter, interviewer, team, etc. There are too many variables to form a perfect algorithm, but there is enough data to cover a wide breadth of companies and situations. What you do with the information is up to you.

So, go forth and read! I encourage you to actively read meaning to highlight information that you find valuable, circle words or acronyms that you are not sure of, and take notes in the margins. I wanted this book to grow with you no matter how many times you come back to it. There is a glossary at the back of the book for many of the acronyms and words that I use throughout the book that I thought would need some more in depth information and exploring.

Your first step to your next career goal is just beyond the next page. Good luck!




The
Content
Creator

Social media has ruined us in the best way.

It has become the driving factor behind how we consume entertainment and marketing. Since this book is written during The Great COVID-19 Pandemic of 2021, it has also become our lifeline to the outside world. I don't mean to sound old, but I am, and I also need to drive the point home about the importance of social media; It is not something to ignore.

With that being said, do you need social media to help your career as a software engineer? The answer isn't simple. In fact, at most, it's a strong maybe.

I will lay out the benefits of social media and how it has helped me progress in my career. The positives do outweigh the cons for me, but that's only because I have to force myself to take a step back when it starts to become overwhelming, and it *will* become overwhelming.

Social Media Management

Limit Your Social Media Accounts

Seriously, do it.

I force myself to focus on creating content on one social media profile, and that's on Twitter. You can choose whatever platform you use the most or have the largest following on and focus on that. I don't suggest using Facebook as your main one because the interactions between you and your followers is slow and archaic compared to Instagram and Twitter.

In order for me to focus on my Twitter posts and interactions, I limit myself to two Twitters: my personal and my podcast's. I do not have a secret alternative account where I post the things that I wouldn't dare say in public. I am at a place where I do not want to work for a company that heavily monitors or filters my social media profiles. I highly suggest that you adopt the same way of thinking when it comes to a distinct separation between yourself and your work.

Most, if not every, company will include a clause about social media in your hiring contract.

While the verbiage may vary, It will come down to these:

- Don't interact with any customers of your company without first identifying that you work for said company
- Don't say or depict anything that's illegal

That's it! Straightforward, no?

If you follow me on Twitter, you can attest to the fact that I cuss a lot in my Tweets and post things that people would consider rude and lewd among some technical and funny gems.

You do not need to lose your personality in order to gain a following. In fact, personalities are how real people follow your account and your content creation career.

How to Grow Your Following

This is the annoying part of social media where I have to tell you that you need to do more work outside of social media in order to get people to follow you.

When I first started Git Cute Podcast, I had shy of just 1,000 followers. To me, that was a lot! I had been on Twitter for 9 years at that point and used it primarily to Tweet nonsense and to keep up with pop culture and other entertainment.

It wasn't until I started Tweeting heavily about programming that the followers slowly started to trickle in. I also started following these people back, and suddenly I had accrued another 500 or so followers that were all in tech in some way or another. Soon after, I started giving talks at more tech conferences and was invited to speak on panels once people saw how much I enjoyed presenting (It helped that I was also great at it.) Subsequently, the follower account ascended.

I put my Twitter account information *everywhere*. I make sure that I say it in the ending credits of every Git Cute episode that I make. If I write an article on Medium or another platform, Twitter is my main way of communication after my email. If you make your social media account the only way for people to reach you, then you are bound to have more people follow you.

I mentioned that I put my Twitter account in a lot of places, and that brings us to the fact that you want to think about how you want to grow your presence outside of social media and the people on your team.

Have you come home from working a full shift at your day job, pull open VSCode and start fixing bugs that are on an opensource application that you started for fun one weekend, but now it's turned into a source of responsibility because it is generating income for you?

Side gigs are running rampant within the tech community - or at least on Tech Twitter - and are skewing the perspective of what it takes in order to be considered an influential person within the tech space.

The truth is that running anything outside of your daily tasks - work, errands, things to keep you alive - is time consuming and can be potentially very expensive if you don't position and market yourself early.

This is all completely optional, but I have met the best engineers, designers, technical writers, developer relation engineers, DevOps engineers, etc. through social media.

I am not saying that you have to do anything with social media to become a senior software engineer. However, what I am saying, is that much like networking in the traditional sense, social media can quickly give you connections to people at companies that some day you would like to apply to and grow with. I found out about my current job through social media because I had wrote a Tweet about how studying for a technical interview was difficult when you are actively recovering from PTSD.

Remember that social media is second only to your content when it comes to creating content.

Content Tips

I cannot stress enough that you do not have to create content in order to become a senior software engineer. Most of the people that are seniors exist outside of the microcosm that is Tech Twitter, and those people are doing beyond fine, however, I have found great success with networking and honing my skills with content creation, and I want to share those tips with you.

Write About What You Like

You will find that it's easier to write about topics that you are passionate about aside from picking something because it's popular at the time. For instance, the first thing that I wrote about for a conference talk were application monitoring systems because I was deep into researching and implementing this for a client at that current job. I knew a lot about it, and all of it fascinated me, so it was easy for me to write a talk surrounding the topic and to also sound knowledgeable.

Honesty

Honesty and vulnerability are traits that people often tell me that they like about my Twitter and my content in general. Most people know when they are being lied to or when something seems fake or staged. While content like that may increase your numbers, it won't give you a loyal fanbase that will come back for the next episode, article, or book.

There are plenty of people that have YouTube channels, blogs, and Twitter accounts that copy and paste the same content from each other in order to get tens of thousands of followers. I am sure you have seen the "How to Become a Developer in 30 Days" threads and other variations like it. That is not honest or genuine content creation. That is simply clickbait, and unfortunately, it works. Creating great content and being honest is the longer path to creating an audience, but it will be the audience that sticks with you. At the end of the day, content creation is marketing, and the first thing you must market is yourself.

What makes you different?

Schedule

I am bringing up schedules a lot in this book, but there is a reason for it! Schedules can help you avoid burnout. The chances are that you are already doing something full-time and adding content creation is a part-time - if not another full-time - job.

I like to decompress after work with a few hours of gaming, so I leave all of my content creation tasks for the weekend. If I have no social plans, all day Saturday will involve me working on the transcript for a "Git Cute" episode or fixing bugs in an open source project. Sunday is reserved for recording the episodes or any other activities that I happened to pick up during the week.

If you do not set a schedule for yourself, you will experience burnout.

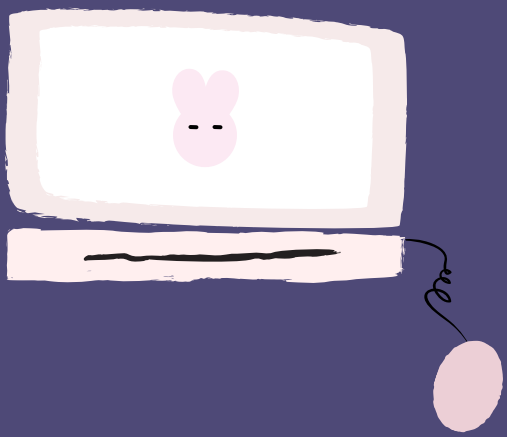
The following websites and forms of media are a great way to get started in expressing yourself and to segue into

- Medium
- Dev.to
- Blog
- Newsletter
- Podcast

Medium

Medium is a great self-publishing article platform that will allow your technical writing to reach out to a wider audience. It is also easy to turn articles that you post to Medium into extra income thanks to their Medium Partner Program. The Medium Partner Program allows you to put your articles behind a paywall, and users that pay for Medium's subscription services can view your articles, giving you a percentage that adds up based on how many people view your article.

The downfall to this is that people that do not have Medium's subscription service cannot view your article therefore drastically decreasing your audience. From experience, putting the one article I have on Medium behind a paywall helped cut harassing comments, so it is up to you to weigh the costs depending on the content you want to share. I suggest your articles be free at first, and then when you garner a following, put it behind a paywall to secure a stream of passive income.



Dev.to

Dev.to is a popular site to post to among the tech community do to it being marketed to developers by developers. With a community of over 500,000 developers, it is a great and direct way to put your tech articles in front of people just as passionate about technology as you are.

They also have a feed on the main page that shows articles that have been recently published. It ensures that there is a way to get your article to as many eyes as possible along with its tagging system.

Blogging

Both Medium and Dev.to are streamline, hosted ways of blogging. If you don't want to go to either platform, you can host your own blog on your portfolio website or a different blogging platform altogether. My advice for writing is to write about things that you are excited about sharing or something that you found out the hard way that other people may need to know about a specific technology.

Podcasting

The hard part isn't starting a podcast; it's keeping it going.

Podcasting is a combination of managing your social media following and also producing content that you and your audience are interested in without growing stale.

I would strongly suggest that you start a podcast with at least one other person so it is easy to bounce ideas off of each other and also to feed off of each other's energy. I am the sole mind for 'Git Cute' and while I love having sole creative control, it can be challenging to think of ideas that excitement, writing the scripts on my own, and also all of the post editing.

You do not need expensive equipment to start out podcasting. I started with a $40 Blue Snowball microphone and a small notebook of ideas. You could even record directly from your phone and do the editing with apps like Podia.

Why are you doing all of this yourself? Can't you hire someone?

Great question!

This answer to this comes down to basic business economics. Once you start your adventure into podcasting, you will quickly come to realize that you are now running a small business, and that every thing that you would pay for to save yourself time and money will put you into the red very quickly for business finances.

Where I would love to be able to outsource a lot of the administrative tasks of podcasting, it is not financially realistic for my business at this time. I am operating at the lowest cost as possible, and that involves me putting in the time and energy for the podcast. While writing this book, my podcast went on a month hiatus. It's because I could not balance giving consistent, quality content on 'Git Cute', and then put that same amount of energy into it as well. I quickly found the balance between my job as a software engineer and this culmination of side gigs.

I compare creating a podcast to producing a radio show. The components of both are very similar! The only difference is that you don't need to go into a studio in order to create it. I create my podcast from the comfort of my couch every single week.

Podcasting Tips

- Always write a transcript for your episodes.
- Be consistent.
- Create a tagline you can use in every episode.
- Make sure that your intro and outro music is fun
- CITE YOUR SOURCES!
- Your first episode is not going to be your best. Just start recording!

Podcasting Resources

Hosting:

- Wordpress.com
- Squarespace.com
- Podia.com

Recording and Editing Software:

- Audacity

Equipment:

- Shure SM7B Microphone
- Blue Snowball Microphone
- Cloudlifter Mic Activator
- NVIDIA RTX Voice

Streaming

Streaming has quickly become not just for us gamers anymore! There are a whole group of software engineers that have Twitch channels that are dedicated to creating coding content and interacting with other engineers or aspiring engineers.

The truth is that as long as you have a computer that can handle running Streamlabs OBS or another streaming software while you have your IDE of choice open, you can also become a coding streamer.

I happen to stream myself, but I stream strictly video games. If I added my job into the one thing that I truly enjoy the most, I will experience burnout, so be careful of your own personal limits.

Popular topics that are streamed are coding open source projects, working on hardware, building a PC, or any other topic that is related to computers.


The Resume

Josie Harper

EMAIL | | GITHUB

Skills

Experience

Education

A properly formatted and detailed resume is the fastest way for you to skip the first barrier toward your senior software engineering job. During the time period that I was focusing on consulting full-time, I would fix resumes for people that were having issues just to get to prescreening interview.

I gave you a mockup of what I prefer to see for a resume. As far as the 1 page for a resume goes, I strictly follow the 1-2 page rule. If you have been in your current industry for a number of years, it is important to go through your resume and cull jobs and experience that is no longer relevant to the position that you desire to apply for. For example, if you have been a software engineer for 15+ years and are looking to apply for Director of Engineer positions, your first job as an intern for IBM is not going to be relevant in terms of showing your managerial, technical, architectural, and leadership skills.

It is common for us to just add onto our resume and ship it off toward a company's automated response system or a recruiter and hope that it is good enough. I guarantee that if you do that you will find yourself with an inbox full of automatic rejection emails or no emails at all. Why?

These systems and recruiters are going to glance at your resume looking for keywords that.

It is important that your resume is easily *scannable* within a few short seconds. I would like to say that people read the full resume before putting people into the hiring pipeline, but I would be lying. Whoever looks at your resume first is scanning for key elements that have identified they need for a specific role and will either approve or deny based on that.

It's why I included a resume temple at the beginning of this chapter that has gotten me, and my resume clients, the most success in obtaining interviews and offers.

The Overview Section

Hot take: I hate overview sections; they're redundant. You should not have to summarize your experience and skills at the beginning of your resume because if your resume has the correct and pertinent information, I should be able to get an overview of your experience from that.

Leave this part off of your resume.

Please.

Skills

When I scan resumes, the skills section is what I focus on the most. I am looking to see if the person has the skills that I am looking for with the job requirements OR has similar experience in another language or framework. There are a few ways that you could format this to make it easy to read.

- Alphabetically
- From most knowledgeable to least

Experience

Your job experience is the most important part of your entire resume, even more important than your name. I have reviewed a lot of resumes, and the Experience section is where 99% of them fail before they even get the chance to interview.

This is where you need to make everything that you ever did technically sound like it was the most impactful piece of code that you have ever written.

The hardest part about describing what you have done is that you have already decided that what you accomplished was insignificant. The language that you need to use for your resume has to be specific and using technical keywords that a recruiter or hiring manager will be seeking for their specific role.

For example, say that at work you are currently a mid-level Java software engineer, and you are going to be applying for senior level Java positions. Java positions are typically backend orientated, so there's a high chance that you are going to be building RESTful APIs with Spring Boot or another similar framework. Luckily for you, you've already created an endpoint in Spring Boot for the application at your current company.

You should ask yourself the following questions when you are creating or revising your resume experience:

- Does this explain the level of commitment I put into this particular task?
- Does this explain what languages and/or frameworks I used for this particular task?
- Did I explain what this task's function was?
- Did I explain the business impact that my task had?
- Can I quantify that impact with numbers or percentages?

Numbers and percentages may seem odd to you, but when you start building and fixing more problems for businesses, the more you realize that there are analytics and metrics for nearly everything that you work on.

These things should be discussed between product, your manager, and the team frequently. If it is not, you can access the monitoring for your specific API and endpoint directly and see what performance increases have happened since your work had been pushed to your production environment.

It's possible that for a fix that you felt was minor had a 50% drop in 500 errors for that specific endpoint increasing completed transactions by 70%. That's why it's important to view your achievements in metrics when it comes to your resume because it will even help you realize the positive impact that you have on your application.

Everything Doesn't Belong

Now that we have gone over what you need to have on your resume, we will explore what you need to leave off of your resume at all costs.

Do not put anything on your resume that you do not want to be asked about or hired for that particular skillset. This is incredibly important because when you are in the mindset of getting a job, any job, you won't care about the stack you are working on so long as the check clears after you sign the dotted line.

Also, do not put every single job that you have ever had on your resume. Every job experience that you have on your senior software engineering resume needs to be a technical or engineering job. With that being said, the job that you had a IBM in the mid-90s is not relevant to what you will be applying for today.

Education

You should be proud of the college or university that you graduated from, however, you are no longer looking for internships. Your education should not be at the top of your resume. The tech industry is slowly trending toward not caring — or that's what they want us to believe — about degrees. When it comes to senior and above positions, where you went to school might get you a "Oh, hey, we're alumni!," but it is not going to guarantee you a interview. The concise explanation of your employment will.

Optional Sections

I give a lot of technical talks and also used to do a lot of mentoring to women that were indeed of guidance at the beginning of their journey in the tech industry. Since I find that this information is relevant and will further support what a valuable asset I am to the company I am applying for, I have two additional sections on my resume for this. If you have a volunteer section that is relevant, feel free to add that as well.

Aesthetics

There are amazing websites that will help you format your resume accordingly. I personally use resume.io for mine because there are so many templates that you can easily drag and drop your information into without having to worry about your color themes, fonts, or where the information needs to be placed. However, if using a resume template generator is off-putting, here are the general guidelines I have when creating resumes for myself and others.

- No more than two different font families for your resume.
- Use different font sizes and weights for emphasis rather than a new font.
- Double space your resume for easy reading.
- Use consistent spacing (IE: tabs vs spaces, enter, etc.)
- Spell check at least two times and then have someone else go over it for you.
- Use one tone for your entire resume (IE: past tense)


The
Interview

The Hard Truth

If you were hoping that I would walk you through algorithms, system design and API design questions, this is not the chapter for you. There are plenty of great sources that exist already that can walk you through the technical portion of the interview process (IE: Emma Bostian's Decoding the Interview Process and "Cracking the Coding Interview".) I want to focus on processes for solving complex problems, system design, and the forgotten soft skills that people often forget to study.

The chances are that if you are not a sociable person, that these are things that you will have to think about and study, and that is okay! While I am not cosigning that having to conform to these societal standards, the reality is that these are still markers that a lot of companies, FAANG included, look for when they are hiring. Like the rest of the advice that I have written in this book, feel free to add, adjust, or remove tips that do not fit your personality.

Recruiters

A good recruiter is there to help facilitate getting matching you with a compatible company, team, and salary. This means that you need to have open communication and. As much as you may hate getting the text messages, emails, and phone calls, they are trying to help you at the end of it.

Be Honest... About Your Skills

There is no benefit to lying to a recruiter if they have reached out to you with the opportunity to interview for a job. This is a lot of preparation, but I do love to be prepared. I have a list of things that I would leave my current job for that I like to call my perfect job. Sometimes it is a dream company but other times it is a list of criteria that the recruiter and company would need to meet in order for me to even considering leaving a company and team that I enjoy.

LeetCode

LeetCode is a platform where they specialize in algorithm questions ranked from 'Easy' to 'Hard' based on the complexity of the subject and solution. They also have a forum where people share what interview questions they encountered in phone screens and on-site interviews to share with the larger public and to help software engineers prepare. I have been fortunate that the only time that I encountered a LeetCode problem during a screening process was for Amazon. As tedious as studying for LeetCode problems is, I have compiled steps that made solving them easier and faster as I was grinding algorithms.

Set a Schedule

It seems odd to set a schedule for studying, but if you don't set a couple of hours each, or every other, day to devote to solving LeetCode problems, you will never do them. Trust me. When I was at the peak of studying for interviews, I was doing a couple of LeetCode problems a day ranging from easy for warmups and the majority of them being medium and trying to tackle a

a hard difficulty one just for one. I have never solved a hard LeetCode problem because I have never been forced to do so for a job, but it was fun for me to try them every now and then. Slowly build momentum. I highly suggest going through the series of problems where answering one snowballs into another and grows in complexity. It gives you a sense of confidence when it comes to tackling the rest of them.

Break the Problem Down

LeetCode problems are just really large - often poorly written - word problems. Like solving a word problem, look for key words within the problem that will give you clues as to what you will need to do in order to reach the correct answer.

Brute Force Solution in Plain Terms

LeetCode problems are just really large - often poorly written - word problems. Like solving a word problem, look for key words within the problem that will give you clues as to what you will need to do in order to reach the correct answer.

When I find a solution, I write it down as if I was writing notes about how I would solve the solution.

Pair Programming

Talk the problem over with a friend or find yourself someone to pair program the solution with you. This type of back and forth conversation is what we call "rubber ducking." When you are able to verbally explain to someone the issue that you are having solving the current problem, it usually jogs the correct answer. It is also great to have another person there to challenge you on how you are solving the problem. My friends will typically challenge me on a factor like return type because that may involve some data structure changing and possible casting.

Leave your ego at the door when it comes to solving these problems. Just like we are with work, learning things that may be outside of our scope at the time is best learned with someone else along the way with you. You could also take this as finding a study buddy that will help you with the LeetCode problems as well.

Write It

This is the fun part because usually the language that you are completing the challenge in will make it easier or harder for you. Be sure you choose a language to answer the challenges in that you are very comfortable with. It will make it easier for you if you don't have to go back and forth searching for basic syntax like how to create a variable or a specific data set.

```
class Solution {
    public int removeElement(int[ ] nums, int val) {
        int i = 0;
        for (int elem: nums){
            if (elem != val){
                nums[i] = elem;
                i++;
            }
        }
        return i;
    }
}
```

"No one writes great code on the first go."

Refactor for Efficiency

This is optional, but it is something that I like doing once I have already solved the problem.

All of the test cases are passing, so that means you can just submit it, right?

NO.

Now is the time that you can look at your algorithm and figure out where you could get some much need time or memory allocation back. Here are the questions I ask myself in an attempt to try to make my answers more efficient.

- Did I need to have another for loop there?
- Is there another way to iterate over this without having it?
- Do I need to iterate over this data at all?
- Do you really need to have another data structure there allocating more memory that isn't necessary?
- Is there another data structure that could handle this data more efficiently?
- What if the bounds of the challenge expanded?
- Would my answer be able to cover those additional edge cases?
- How could I refactor it if that's the case?

Why am I asking myself these questions?

Because often in whiteboarding interviews, these are the questions that the interviewer is going to be asking you to elaborate on. If you get into the mindset of asking yourself these questions after every LeetCode problem, if someone asks you this in an interview setting, it will be less shocking.

System Design

Now, there's a lot of discourse surrounding Cracking the Coding Interview. With that said, McDowell's brief but informative chapter on System Design and Scalability absolutely helped me become comfortable with System Design and helped me with how to tackle API design as well.

Why the do they want to test me on system design? Won't I be a senior software engineer?

Ah ha, silly friend!

The higher you get on the software engineer career ladder, you are not only expected to know how to code, but you will need to know how architect a system, to correctly identify system failures, and how to quickly and correctly alleviate said system failures if, and when, they occur.

I love system design interviews because it is chock full of components that I love to learn about.

- Application security.
- Databases.
- Server partitioning.
- Load balancing.
- THROUGHPUTS.
- LATENCY.

As it's mentioned in Cracking the Coding Interview, the most important thing for you to remember is that there is no right or wrong answer. The goal of these interviews is for the interviewer to have you drive the conversation about the problem, see, and hear your thought process.

With the back and forth that happens in system design interviews, it's important that you also know how to take guidance from the interviewer. There is no need to fear them; they're there as your resource and to help you. They want you to succeed.

From here, you would want to break your problem down into the smallest pieces possible. Do you see how this is starting to become a theme throughout the entirety of software engineering and generally complex problem solving in general?

From here, start asking your interviewer questions to help you solve your problem.

- What features would you like me to design?
- Are we sticking with the MVP which would be Sending Tweets and viewing Tweets on a timeline?

You don't want to further complicate the problem, so it's best to stick to the bare bones so you don't spend too long trying to design your system. These interviews are typically 45-60 minutes long. When you factor in the talking with your interviewer with drawing out and explaining your solution, the time goes by quickly.

Asking questions about the specifics of the system will help you understand how many potential APIs you will be working with and to further identify possible bottlenecks that your system might endure.

You know that you are going to have users accessing this system.

- Is there some sort of firewall that you want to put between your users and the direct access of your servers that are storing the API?
- How many servers are going to have your API loaded on them?
- How many data stores will these servers with the API have to access?
- Are there any caching points within your system to make sure your system doesn't bottleneck?
- What sort of database will work the best with this system and what are the tradeoffs between a SQL or NoSQL one?
- Will your database be heavily read-only or will it be making writes as well?

This all might seem complex, and it is, but it is important that you think about these things when going through an interview. For people that already have work experience in an enterprise or any other setting, these things will come second nature. You can easily tie these concepts back to the application that you've had to work on.

When thinking about current or past systems that you have worked on, what would the answer to these questions? Was that system what you would call efficient? If you had the ability to change whatever you wanted on the system, what would you do to optimize it and why?

We have all had those fleeting thoughts where we were fixing something that broke *again* on an application, and you though to yourself, "Well, if we only did this it would stop breaking!"

That is system design.

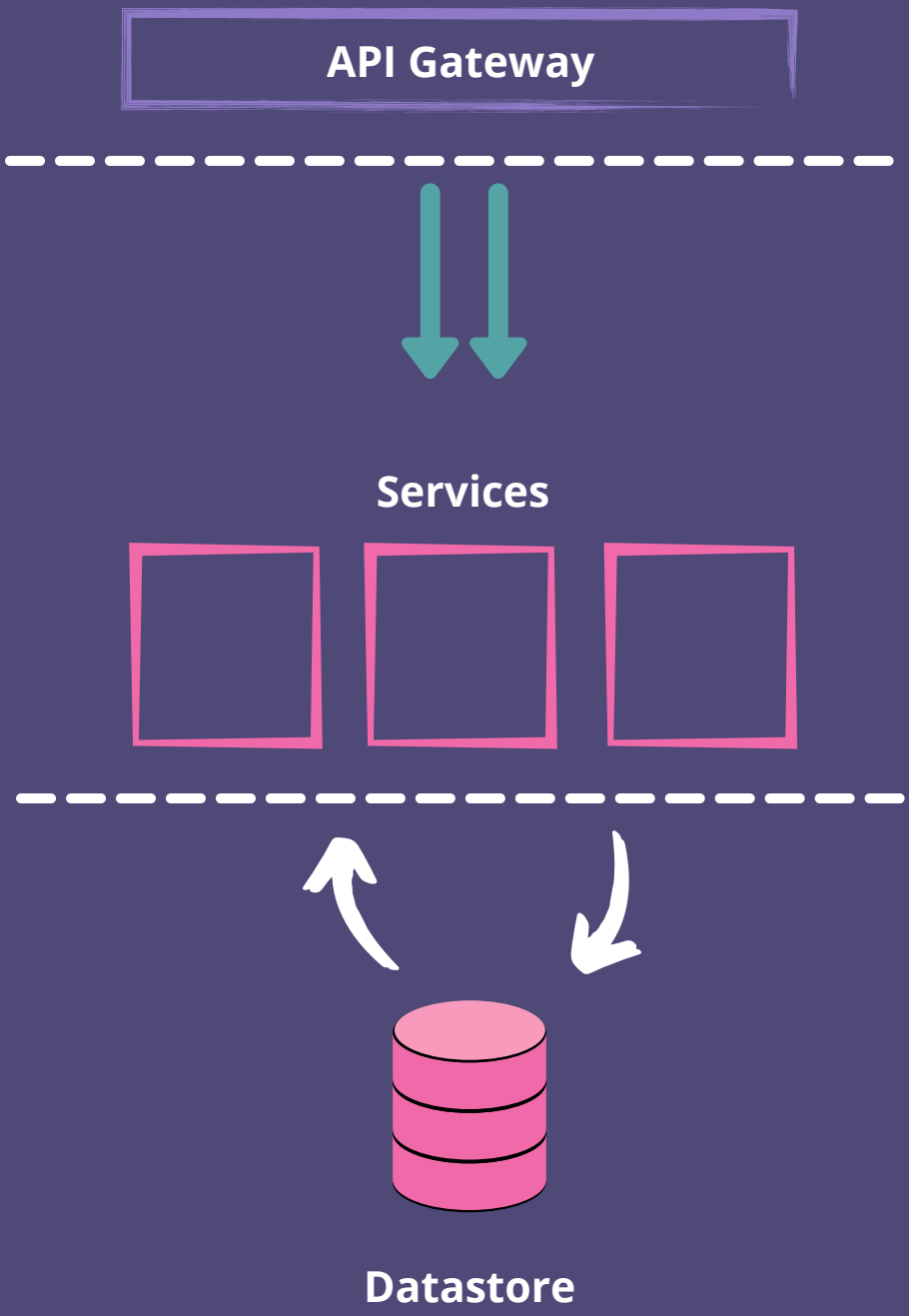
API Design

Typically when you and the interviewer are done going over the system design aspect, they will tell you to deep dive into one of the APIs that you specified within your system that you will need for the system to work.

Sticking with the example of designing Twitter, let's say that we have simplified the process by saying that we are just going to focus on the feature of being able to see your friends' tweets in a timeline.

Remember to still stay relatively high level when talking about the specs of your API. The important things to focus on in this interview are:

- What objects is this API going to need and what object is this going to put out?
- How many endpoints will I need to solve this problem?
- What parameters your endpoints are going to take?
- How many endpoints will you need to complete what the interview is asking for?


API Gateway
Services
Datastore

At this time you are going to look back to your system design and see what other APIs you said you were going to need and if you are going to need the information from them as well for this specific feature. Don't be afraid to ask about possibly editing your system design if you find that you might need more information that you didn't specify in the initial design.

Also don't be afraid to ask for clarification about the specs for this API. Again, this is interview is designed to be vague and they expect you to not only ask questions but to also drive the conversation. The interviewer is there as a resource, but they are not going to hold your hand through the process. A part of being a senior role is being able to identify complex situations and talk your team through possible solutions.

When it comes to defining endpoints and what those endpoints are going to do, the interviewer will probably start asking you questions that will lead you down the path toward your goal. Remember that there is no right or wrong answer. They should not expect perfection because it does not exist and only make sure that you understand the business use case and how to scope the problem successfully.

It helps for me to write out the endpoints in modified code in order to make it clearer to me. If I know that I am going to need specific objects to make this call successfully return a 200, I will write out those objects with the attributes that need to be passed. I will also draw a line between objects so I can visually see where the data is being passed.

Do whatever you need to do — within reason — to make it easier for you to build your answer.

When designing an API endpoint you want to design under single responsibility principle; you don't want your endpoint to do more than one thing. This is where the my highly favored Java naming convention comes in: name your endpoint exactly what behavior it is supposed to be performing. This will help you figure out what parameters you are going to need to get the result that you want.

Say you've decided that your endpoint is

```
retrieveTimelineTweets();
```

What parameters do you think you need to pass in order to get the list of Tweets successfully? Perhaps a User object? Maybe a collection of friends that are tied to the User? But how would they be tied? What sort of schema are we encountering in our database? Wait, speaking of databases, what kind of database would we be using?
These are the type of questions that you need to ask yourself and the interviewer to get to a successful end API design.

Soft Skills

Soft skills are not talked about enough when it comes to preparing for your interview with a company for a software engineering position. Oddly enough, a lot of companies are known for heavily weighing passing your soft skills interview almost as heavily as passing your technical skills interview. For example, Amazon is known for it's difficult technical screenings, but are also known for an equally difficult soft skills interview regarding

Listen

You never want to take over the conversation with your interviewer. While you do want to make sure that your answers and opinions are heard, you don't want to ramble on either. It is beneficial to keep your answers to questions as short and succinct as possible. With that being said, don't be afraid to ask for clarification to questions.

Body Language

Soon companies will start making potential candidates go back on-site for their interviews, but this advice is relevant for remote interviews as well. First, let's start with a story.

When I was still mentoring people at my coding boot camp, I would hold technical interviews with any person that needed the practice. There was one particular person - who is a friend now - that had over-the-top body language. Relaxed posture. Wild arm movements. (Think Wacky Wavy Inflatable Arm Flailing Tube Man - seriously.) While this was not something he was conscious of at first, I know that this body language impacted the outcome of my mock interview and potentially others.

Maintain eye contact with everyone in the room. Even if there is that one person that is in the interview but is very clearly still coding something on their laptop, make the effort to look at them when giving your answers.

The Wait

The next few days to weeks after an interview can be anxiety inducing because of not knowing. I can sympathize; I get anxiety basically over everything. What can you do while you wait? When is the best time to follow up if you don't hear back from your recruiter or interviewing contact?

Go Over Any Information They Give You

It is best to start prepping yourself for the possibility of getting the job rather than focusing on the negative possibility. If your recruiter gave you any information about benefits (IE: health insurance, 401k, RSU vesting, employee stock purchasing plans, life insurance, etc.) you can go over these details with a fine tooth comb and create a list for clarification when the recruiter reaches back out. Here's the list I keep for when I am interviewing.

Team

- [ ] Confirm your manager, team, and organization
- [ ] Confirm your job role and official title
- [ ] Confirm your tech stack

Benefits

- [ ] Confirm when the health insurance starts
- [ ] Confirm vestment schedule for RSUs (if any)
- [ ] Confirm pay schedule
- [ ] Ask for benefits website and/or paperwork
- [ ] Prepare paperwork for ACH direct deposit

Paperwork

- [ ] Sign the offer letter
- [ ] Finish background paperwork
- [ ] Finish tax paperwork

The Contact After

I am an impatient person when it comes to communication, and this has benefited me multiple times when it has come to securing a job, tech and otherwise.

Keep in mind that the recruiter is there to help you and that they also want you to join the company.

If a recruiter said that they are supposed to reach out to me by end of day or perhaps before the weekend, and Wednesday rolls around with no answer, I will draft an email like the following template.

Hi {insert recruiter name here},

It was great interviewing with {insert company here} ,and I appreciate you facilitating this entire process! I wanted to follow up with you to be sure that I hadn't missed any correspondence from you regarding to the position.

I look forward to hearing from you.

Regards,
{insert your name here}

It's after this correspondence that I back off and wait for them to reach out to me. There are many factors that could have a recruiter delayed in reaching out to you.

- Waiting for higher level approvals on offer
- More deliberation is happening over your interview packet
- They are negotiating on your behalf for your offer

Do not panic!

You have made it through the technical screening an onslaught of technical interviews. If you have interviewed at any other companies, this is a great time to follow-up with those recruiter contacts as well.

The Ghosting

Ghosting is not common when it comes to companies, but occasionally, you will have contact with a recruiter or hiring manager that will simply not follow up after reviewing your resume or even after you interview. I consider ghosting to have occurred when you have not had any communication with your assigned recruiter or company for 2 full weeks.

You have every right to be upset when you have been waiting for weeks to hear for an answer about a job, but, they do not owe you that communication.

Kindness is not something that every person or company practices. It is wildly unprofessional and be sure to add that company to your avoid list for future communication and job prospects. If they ghost you once, they will ghost you again.

With your feelings aside, how are you supposed to handle a recruiter or company ghosting you? Well, with a carefully crafted email of course!

Hi {insert recruiter name here},

While it was great interviewing with {insert company here} , I have not had any communication with you since the interview on {insert date here}. I know that we were in the decision phase with my interview packet and wanted to make sure that I was still be considered for the position as I am continuing to interview at this time.

I look forward to your communication.

Regards,
{insert your name here}

The Offer

You did it! You have received an offer for a senior software engineer role, and that is not a feat to take lightly. Now, the numbers that you are going to be viewing will more than likely be an astronomical sum of money, and when seeing numbers like that, you will want to just sign on the dotted line.

I have a lawyer on retainer, so I will pass any contracts that I receive - jobs, apartment leases, etc. - through to him so he can review them and make sure that there are no hidden clauses that could be seen as abnormal when it comes to employment contracts.

If you do not have a lawyer, take the time to carefully read the contract and take notes on things that you do not understand or if they seem strange to you. You can pass those notes along to your recruiter for verification, and after you are satisfied, you can sign on the dotted line!

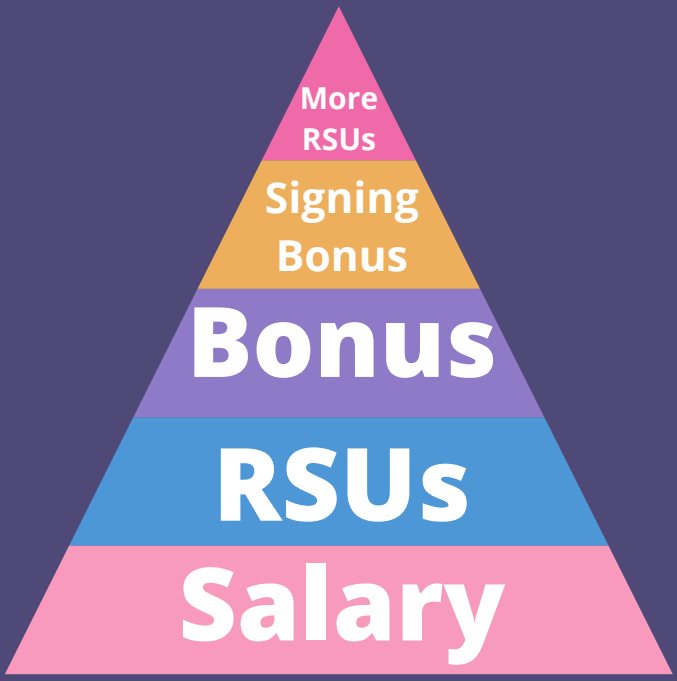
....Or you negotiate.


The Salary

People have jobs because they need to make money in order to support themselves by paying bills for housing, food, utilities, medical needs, and maybe to have some money to buy some fun things when everything is paid for. I went to the coding boot camp to become a software engineer because I knew that I would no longer be living in poverty and would immediately be elevated to middle class upon completing the course and passing the interview. Money is why people work.

Now that we have that out of the way, as software engineers, companies in the United States are throwing copious amounts of money at us to get us and retain us. Unfortunately, I do not have any experience for salary numbers in other countries, but the negotiating tips that I give you can be used for any situation outside of cultural society norms that I am not privy of when discussing monies. All discussion of monies is in USD.

Firstly, I need to break down how salary within tech is broken down with FAANG companies and other companies that are based in Silicon Valley or base their recruiting practices off of these companies. If you go to look for any information regarding software engineering positions through your search engine, you will see that people refer to "TC" quite often. TC is total compensation. What is total compensation you ask? Look below.



Total Compensation Pyramid

Total Compensation encompasses all of these elements: Base salary, Restricted Stock Units, Bonus, Signing Bonus, More Restricted Stock Units.

Salary

If you have been a non salary employee before, then this is exactly the same. It is the amount of money you make hourly based on a 40 hour work week for 32 weeks. When you are looking at offers, you will see that your base salary will generally look low. Do not let that scare you; the rest of the money comes with the rest of the pyramid.

Restricted Stock Units (RSUs)

This is where software engineers make the majority of their money. Restricted Stock Units are stocks of the hiring company that they offer a potential employee, typically presented in a large lump sum with a vesting period attached to them.

When we get into stocks, it's easy to start thinking of r/wallstreetbets and investing, holding, etc. Just know, that at some point you will be able to do that with those RSUs, but we are focusing on how they are given first.

For example, say that I am going to work for a Silicon Valley company. The recruiter returns to me and says that they have great news, and they verbally give me an offer letter run down for the potential compensation. I happen to fixate on them saying that they are going to give me $200,000 in RSUs. When you hear $200,000, you think, "Holy shit!" because that is a lot of money. However, there is a question that you need to ask:

What's the vesting period for the RSUs?

Vesting Period

The trick with RSUs is that you do not get the entire lump sum of money that the company is offering you right away. They are given to you over a period of time such as 25% of the RSUs over 4 years. This span of time is what is known as the vesting period. The other trick about RSUs is that you need to stay at the company for the entirety of the vesting period in order to get the money.

The really cool thing about the vesting period though is that if your company issues based on stock units vs stock price, the amount of your stock units will ebb and flow with the market. So, it's possible that the 25% of your RSUs on your vesting date will be worth more than what it was when you originally were given them. The opposite does apply, but, the great part about that is that you don't have to immediately cash out your stocks if they are low. You can hold onto them for as long or as little as you want. When to hold and sell is up to you, and this is *definitely* not financial advice, but I recommend researching about the stock market and other ways to reinvest on your own and to make your own decisions.

Bonus

We are used to bonuses as a culture. Perhaps you get one in the winter at the holidays because companies want you to have extra money for gifts and food around that time. From the job offers that I have received in tech, bonuses are talked about strictly as percentages and always given as what you *could* earn. What does that mean? It means that your bonus is not guaranteed, and for that reason, if I am making large purchases, I do not use my bonus in my total compensation.

Percentages? Why?

Bonuses are often tied to 2 things: the company's and your performance. I view the company portion of the bonus as a multiplier. If your company has a really great year and makes a lot of money, if they are worth a lick of salt, they will pass some of that money down to you. Now, with my performance, it is based on how I perform throughout the year. Did I accomplish the goals that I set up with my manager at the beginning of the year? Did I also take advice in my 1-1s and mid-year review and apply it to myself? What projects did I work on? There are a lot of other questions that can be asked based on what your company's guidelines are for reviews, but a combination of all those elements will usually get you your max percentage.

Back to our scenario. The recruiter answers that the vesting period is 25% over 4 years which is a little bit of a long time but typical.

They also tell you that you have the chance to earn up to 20% of your base salary. For the sake of keeping the math simple, say your proposed base salary is $200,00. 20% of $200,000 is $20,000. An additional possible $20,000 tacked onto your base salary. Holy moly.

Just so we are clear on where we are now in terms of calculations:


$200,000
+
$200,000
4
+
$20,000

We are already sitting at an annual TC of $270,000. I went ahead and added the bonus so you could have a better picture of the amount of money that you are looking at receiving.

Signing Bonus

A signing bonus is given for you to sign the contract. Contrary to popular belief, it is not given to you when you actually sign. It is bundled in with either your first or second paycheck depending on the company and payroll processor. When I had my first job as a software engineer, I received a signing bonus of $5,000. I was gobsmacked; I had never had that much money given to me at one time *in my life*, and I definitely did not receive a signing bonus at any of my previous jobs. Now, when I am interviewing for positions, I ask for one. The recruiter will typically give me a spiel about them not normally giving a signing bonus and having to seek additional approval, but I've never been denied one.

The recruiter says there's more; they are offering you a $25,000 signing bonus.


$200,000
+
$200,000
4
+
$20,000
+
$25,000

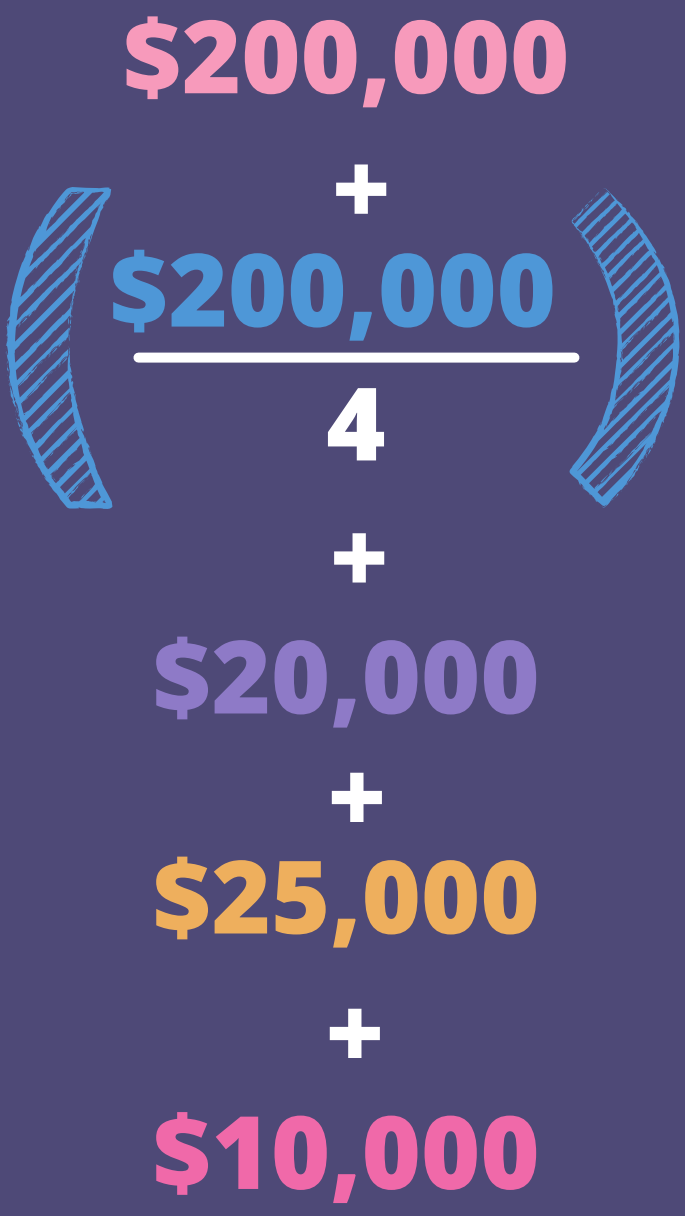
$295,000 per year.

Is that a lot of money? *Absolutely.*

Is that a real number that a senior software engineer can make at a company? Yes, and it's not even the highest salary that I've seen. And these numbers are made up!

More RSUs

There are more RSUs and these are known as refreshers. Because software engineering is such a competitive field, especially in Silicon Valley, companies offer employees extra RSUs throughout the year as refreshers essentially to keep you with the company longer. These refresher stocks typically have a vesting period of one year, and I haven't found with any of my jobs that they followed a base percentage, but that could be the case at other companies.


$200,000
+
$200,000
4
+
$20,000
+
$25,000
+
$10,000


305,000
USD

$305,000 annually.

It is astronomical and a real number that is possible for a company to offer you or for you to negotiate up to if you want. While I already discussed that money is generally a motivating factor for people to be a software engineer, I should also say that you do not have to negotiate when you receive your offer. I know a lot of people are judging me writing this, but you don't! I can understand the pressure of having to weigh your worth with monetary value, and it can be daunting. However, I will let you know why I negotiate on every offer that I receive.

I was asked in an interview with a local publication once if I thought that the amount of money that software engineers made was obscene. The quick answer is yes, I do, and I am aware of that. In fact, so vocal about it that I made it a whole chapter in this book.

Despite the factors that I am going to go into about *my* personal experience in this industry, I believe that everyone should accept the amount of money that they believe they are worth. That is not a number that is finite or one that I can dictate for you.

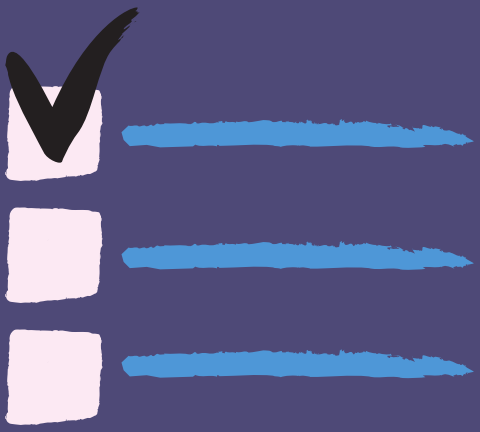
As a Black woman that is a software engineer, I find added pressure in making sure that I am getting the maximum amount of what I am worth. Because I know that if I take the bare minimum offer from a company that the next Black woman is going to be expected to take that amount as well, and that is not an idea that I want to perpetuate by my actions. There is nothing worse than realizing that someone else on your team or in your location is making significantly more than you.


Senior
Responsibilities

As with every other job, when you add 'senior' in front of your title, not only does the money go up but so does your responsibilities. I do have to preface this chapter with the fact that you have the option to not do anything that I tell you to do. If you do not want to mentor a junior or mid-level engineer at your job, by all means do not do that! However, successful senior software engineers that will eventually be promoted to staff and principal engineers follow the same formula, and a lot of that is centered around human interactions and great technical understanding.

Your Onboarding

As a senior software engineer for any company, your scope of extends way past just coding and system design. You are now seen as someone that has some ranking authority on your team, someone to look up to - a mentor even! The following are things that I like to keep in mind for new jobs to make sure that I am staying on task so I can become productive faster in any new senior role.



Your Introduction

I like letting my team members now who I am and exactly where I stand as a person as soon as I join the team. Aggressive? Sure, but I am an aggressively bubbly person, so this suits me quite well. This usually ends up being a blurb being passed around in various email chains, but I want my coworkers to have a great first impression of me outside of interviewing.

I also make sure to send out a message or email to any interns or junior developers on the team. I want to let them know that I will be there to answer any questions that they have during the onboarding process and beyond. I have a good understanding that being a senior software engineer

means that I will need to be available as a resource and a mentor for these specific people and also the rest of my team! But because I empathize with how stressful being new on the team can be when you are at your first job, I make myself available to them as a resource and confidant.

You won't know a lot of specific answers about the code base because you will be learning that together with them, but you can give them your expertise in setting up their machine, dealing with general Node or JVM problems, or maybe just answering questions about how to approach solving problems when they ask. Opening that channel of communication early is very important so you can let them know that you are readily available as a resource. No one wants the senior software engineer that thinks their time is too precious to answer a junior engineer's question or anyone's question for that matter. Collaborative environment and all of that.

Working Hours

With the Covid-19 pandemic in 2020, a lot of companies that would never hire engineers remotely are now changing their companies to fully remote or opening up more possibilities for being remote. Because of this and the addition of more time zones, it is important that you figure out what the best working hours for you will be.

There are a couple of factors that I take into account when I want to figure out my working hours: what hours am I the most productive? When will most of my team be available?
A lot of this will depend on how flexible your manager is with setting your hours. I am very lucky that I have an amazing manager at PayPal, and we have a open communication when it comes to my hours. I am on the East Coast where most of my team is on the West Coast in California, so my working hours differ so I am not working until 8PM at night.

Don't Miss Meetings

We all hate meetings. I get it, you get it, everyone gets it. We wish that a lot of them could just be a message in Slack or Teams or perhaps even an email, however, they are another necessary evil. Tech seems to have a lot of them, huh?

Again, I will be prioritizing remote work, because if you are going into an office, there's a large chance a coworker will tap you on the shoulder or drag you off toward the conference room for your meeting.

Every company and team could have a different idea of what meetings are considered important, but generally if you are working in a scrum environment, these are the ones you do NOT miss if you are able:

- Stand up
- Backlog grooming
- Sprint planning
- Knowledge transfers (KTs)

Stand Up

A stand up is a daily scrum meeting that runs 10-15 minutes for every member on your team to go through from the day before: their progress, their blockers, and what they plan to do today. It's a great way to get a temperature check of what everyone on the team is accomplishing and if you need to swarm and help someone with a task. It can quickly become terrible if they turn into 30-45 minute meetings.

Backlog Grooming

Backlog grooming is where your team has stories or tasks that they have already written but are inactive and haven't been assigned to anyone to complete. As a team, you go through each story to make sure that the acceptance criteria and ask for the engineer is clear and then assign story points - points of difficulty - to the tickets. That way the ticket and just be picked up and worked on immediately once it's moved to 'Ready' state.

Sprint Planning

Where backlog grooming focuses on getting tickets ready to be picked up, sprint planning is where they are picked up or assigned to engineers to be completed within a 2 week period - or a sprint. Depending on your team's velocity, you will usually be required to pick up *n* tickets to be productive for your sprint.

Knowledge Transfers (KTs)

Knowledge transfers are exactly how they sound; you are gathering information from either another team or coworker for further context about an application, process, or task that you and your team will complete. A problem that I had with KTs in the past is that I couldn't go back and reference them when I was in office, but because of everyone being remote, most KTs are recorded and archived for easy rewatching.

What projects does your manager want you to tackle once you are up-to-speed?

Being "up-to-speed" is going to be subjective to your company, team, and most importantly your manager. Most teams don't expect you to make any meaningful contribution - such as code pushes or releasing to production - until 3-6 months into the job. As someone that has experience miscommunication about the expectation of my contribution, it's important that you confirm this with your hiring manager to make sure you are both on the right path of understanding what your year goals are going to look like.

There is a possibility that between the time you interviewed and started that your team started a new project with technology that you may not know. I'd recommend brushing up on the technology that it's using so it makes reading the code base a little bit easier. Doing this a couple of days to a week beforehand can be helpful but obviously isn't necessary, however, knowing what the end goal of the application is and the full tech stack is very important for you to know well.

It is also best to keep in mind that you could be placed on a team that are in charge of a lot of applications or APIs that have varying tech stacks. You should be prepared for context switching, especially as a full stack engineer.

Hiring Practices

Your team should give you a standardized way of how senior members should conduct interviews with candidates. In the event that this doesn't exist, you should come in with your own staples that you can use and even incorporate them for your team.

Make sure that you are able to write down your thoughts of the candidate before you meet up with the rest of the committee to discuss the potential hire. The point of the committee is for everyone to give their experience of the interviewee without influence and then come to a general consensus about the person.

Reviewing Resumes

I already know what you're thinking.

"There's a specific way to look at resumes?"

Yes!

There has been a large push for diversity and inclusion within tech over the past 5 years - rightly so - and despite what others may think, this process begins before you even meet a potential candidate face-to-virtual face. It begins with their resume.

I am guilty of being bias toward people that look like me as a cis Black female because we are severely underrepresented in technology, especially as software engineers. While you may be able to gather context and clues from their resume about who they are and what they identify as.

It is key to be aware that everyone harbors unconscious bias toward people regarding characteristics of gender, race, identity,

religion, and creed. I cannot say that every time I review resumes that I have the ideal situation where I can review them in an anonymous manner, but I will outline my tips as I do try to apply them as much as I can.

- A great attitude
- Shows the skill and aptitude to thrive in a collaborative environment
- Willingness to learn
- Technical skills
- Knows our current tech stack

Do you see how "Technical skills" and "Knows our current tech stack" are at the bottom of my list? These skills are what software engineers tend to focus on the most, and rightly so, but I also see it as the easiest thing for an engineer to learn.

Technical skills are the easiest thing for someone to learn on the job, but I cannot teach someone to not be an asshole. Well, I can, but I am not a licensed therapist or psychiatrist, and that is firmly outside of my wheelhouse and yours.

Mentoring

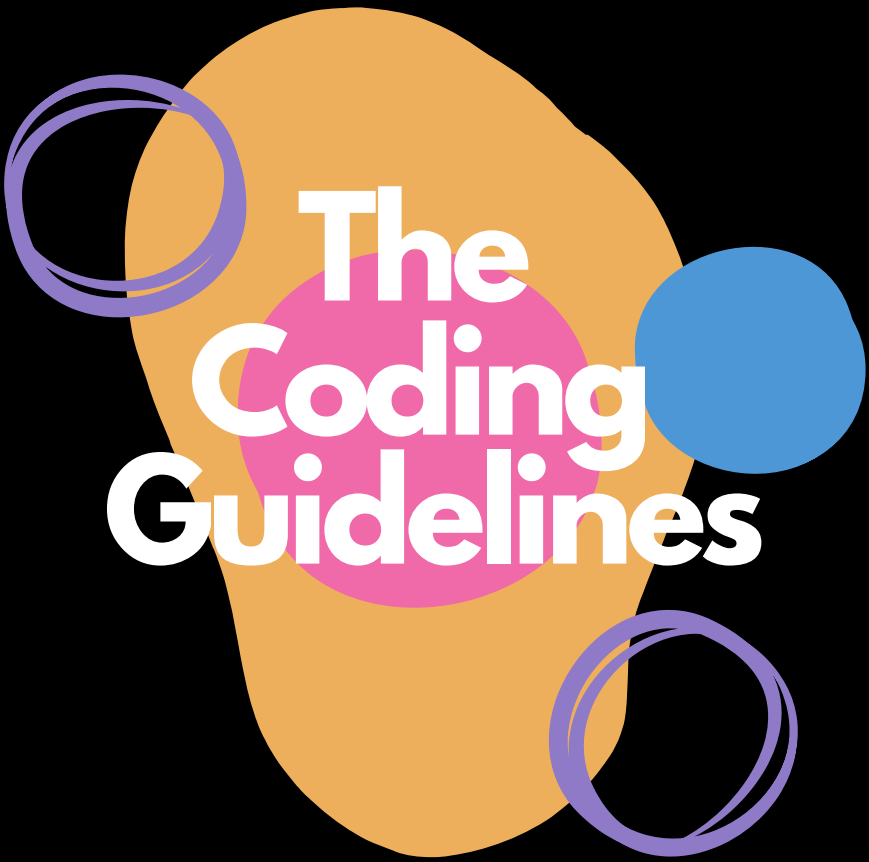
Take a moment to think about when you first entered tech, be it now or many years ago. Who is the first person that you think of that helped you navigate the first 6-12 months at your first, second, or third job? It could have been your direct manager, a skip-level, or perhaps another coworker. The relationship may not have been defined as a mentorship, but that is what helped you succeed and keep going.

Mentoring is an invaluable skill to have as a senior engineer. Depending on your particular work-style, it may be something that you are already doing and didn't realize that you were! Mentoring isn't just limited to junior or mid-level engineers. You could be helping another senior software engineer with something that they are having a difficult time grasping and vice versa. It truly comes down to a willingness to help others

If I am a mentor to someone, I will set up a bi-weekly 1-1 to gauge how my mentee is doing and if there is anything that they would like to talk about or perhaps something that I have seen that I want to commend them on.

While I do have the meeting scheduled, I have the tendency to give feedback immediately through direct messages to my mentee and let them know that they can reach out to me at any time, and that I will make time for them as soon as possible. Some may see this as being extra, but it is extremely important to me that as a person that is in a position that is seen as a form of leadership that I am the model mentor that I wish I had and that I make my mentees feel as comfortable to come to me with any wins or concerns.

Much like me starting Git Cute Podcast and writing this book, my main goal is to help people with the information that I have accumulated through years of work and half a decade of software engineering. I want them to grow into the best versions that they want of themselves in and outside of the job, and being compassionate and understanding are the pillars of the foundation for that.


The
Coding
Guidelines

Coding

I saved the best for last. You didn't think I would go this entire chapter without mentioning my coding and processes best practices did you? Do you even know me?!

When I just started writing Java, I was introduced by my teacher and mentor at the time to "Clean Code." Over the years, the author of the book and other books that I found useful in his series continues to spew vitriol and truly disturbing social takes, so I no longer refer people to him and his works. And, to be honest, his works are quite antiquated.

There are a couple of principles that I follow when I code, but do not go overboard with them because following them too strictly is a fast way to get into micro functions, and honestly, if you are using a great framework, there are guiding rails keeping your code written and organized in a certain way.

- Single responsibility
- Modularity and extensibility
- Formatting
- Extensive testing

Single Responsibility

Single responsibility is my favorite idea to implement and to teach. Essentially method that you write only does one thing. some people find single responsibility is too much micromanagement of code, and while I cannot agrueagainst that, but is my code readable? Could I hand off my code to someone that hasn't looked at it before and between my code and comments, does that person understand it? The answer is yes, so I don't care if it's micromanagement. It's good programming.

Modularity and Extensibility

At this point in your career, you should be thinking about how to make things reusable in your code. Not every coding task that you will complete will to be used elsewhere, but you don't have to rush to finish every task if your sprint is managed well. It could be worth it think about what is being asked of you and ask if this will need to be implemented in any other systems other than your own. You can do this within the confines of your programming language, or perhaps it could grow into a

project where you want to think about the overall architecture of your systems and think of a better way to manage components and make them reusable.

Formatting

Please user a linter. If your current code base does not have one installed, ask your managers and team if it would be terrible to implement one. You have heard of the great war between tabs and spaces, and a linter will keep that war at bay.

Extensive Testing

Extensive testing includes testing every function that you write, creating mocks and updating functional tests. It seems straightforward, but not enough software engineers adopt these properties, and it is worrisome. If your code is fully tested, that means that it is easier to pinpoint when something goes wrong because there are smaller components for you to dive into and investigate. The quickest way to a solution is to break it down into the smallest units and start bit-by-bit, so why would it be any different for testing your code? I am very strict when it comes to this. I agree that code must reach a mandated testing coverage before it can be merged into a repository.

The entire point of this guide is for you to become a *great* senior software engineer, and none of the great senior engineers that I know do not, at the bare minimum, write unit tests. With that being said, I also understand that a lot of people may not have the ideal team or work environment that puts focus on testing. Rather, they want features and products out as fast as possible, and testing takes time.

While they are wrong, at least for your own personal development, start implementing it in unit tests for the code that you write in the code that you touch. It will make you a better software engineer and who knows, perhaps your team, manager, or your company will see the positive improvements that extensive testing gives your application.

Integration Testing

Integration testing is making sure that the components or units of your code behave in a user flow that you have predefined with your product owner. For example, let's say that you are adding a new functional button on your front end that will take your user back to the previous state that they've visited. You've already thoroughly unit tested your button to make sure that the back functionality does what it is intended to do. Now, you are going to be writing a component test to make sure that the component that the button is part of does not

1. Break functionality of other parts of the component
2. Hinder any functionality or use further down the flow

Regression Testing

Regression testing is simply most of the time, that same set of integration tasks that you've already written have run more than likely, but before you go to prod, they are run again to double check and make sure that nothing's broken before we push.

The downside to regression testing is that means that your test suite is going to be massive, but I will take a massive test suite that will lengthen or release over breaking core functionality in production.

Pull Request (PR) Reviews

Does your repository have a Git hook with a format with what is required for your ticket to pass checks? If not, that would be a great thing to suggest and implement for your repository because of automation, but for the sake of brevity, I am going to assume that your repository has this installed and working.

Do not raise a PR when you know that the functional tests are not going to pass because you broke them or didn't update the functional test for the functionality that you are presenting. You having the PR open for an extended period of time while you are writing or fixing those tests is not better than just not having the PR raised at all.

When you are reviewing your PRs, be sure to be mindful about what you are writing on the code review and also direct. Reminder that these comments will be public and also that you are talking to an actual person that spent *n* amount of hours on that PR, and you can't just go through tearing apart like a jerk. Tear it apart like someone's caring friend instead.

Deployment

Deployment, for me, starts before you actually have an application that is prepared to be sent to production. It starts with how you organize your code and repositories.

Git Flow

I am a huge fan of Git Flow. Git Flow is the upstream process of how to take your code through various environment checks before it hits the live production application or site. It typically goes as follows depending on either your personal preference or your company:

A pitfall for this is that technically your QA, Staging and Production environments should be running the same to maintain quality for testing and bug catching, but hat's not the case. There are situations where the lower environments may not be working fully —usually Staging and lower environments— and are experiencing downtime. Another scenario is that there are server patches that are being implemented because of an SLA for application security.

You can't always rely on your environments to be up and up-to-date for manual testing, so a lot of that will rely on you. In enterprise engineering, the process becomes more complex, which I will cover, but I also want to say that if you are shipping a website, there are PaaS—platform as a service—that can help you maintain an appropriate Git Flow with intuitive user interfaces to keep it easy for you to manage.

One of those services that I used when I was at another job was Pantheon at Pantheon.io. Pantheon is great because you can spin up test environments from any branch to test your code before pushing it up the subsequent environments toward production.

Now, I mentioned organizing your code. This is important because I am of the believe all of your code should be branches made off of your master (or main) branch, named with whatever ticket number you are assigned for said task or feature, and a semi-detailed description of what the task or feature is.

Why?

Because if someone needs to roll back a specific update that you made, this is the easiest way for the other person to track down your specific commit and see exactly what you did. Please apply this idea to commit messages as well, and if you don't, be prepared to squash your commits into a more useful one when raising a PR.

For example, let's say that your code has already passed checks in your repository, and you are ready to push this into an environment pipeline.

It makes me believe that their deployment process is manual, but it's understandable that not every company has an automated pipeline. The process for implementing an automatic deployment is a long one and can be even longer if there is not a dedicated site reliability engineer team or remedial knowledge amongst your immediate team about it. In terms of resources, it can be a time sink in comparison to other feature work. The company doesn't have the resources to allocate to to the building and testing of the pipeline if they do not find automation valuable.

Continuous Integration & Continuous Deployment

If you are a Git Cute listener, you will know that I recorded an episode all about continuous integration and continuous employment in Season 1. I have also written multiple articles — coming soon— about the process and how companies should approach implementing and improving it. I know there are other applications that help with deployment such as CircleCI, but I thoroughly enjoy using Jenkins because that is what I have the most knowledge with working on. I have intermediate knowledge in setting up a pipeline from scratch including including plugins and scripting, etc.

For brevity's sake, let's say that your application is a Java application. The artifact has been created and shipped to, let's say, an S3 bucket on an AWS server and it is waiting to be picked up and launched.

Jenkins has the ability to trigger its own job as soon as a new commit is pushed to a specific repository branch and thusly starting your deployment process.

There will be a lot of upfront work that your team will have to do in order to get to full scale automation. There's certain attributes and configurations for you to add in order to be compliant with security standards. Also, more configurations to make sure that you are fetching your artifact and pushing it to the specified environment safely.

There are various checks that will go into this step and triggering your integration and component tests is one of them. Your credential check will verify that you have access to the authorized datastore and also confirm that your artifact is there to copy and deploy.

I insist that you learn how the application that you work on actually deploys.

Take time to look at your older unit and component tests within your application if you didn't write them yourself. Also, talk to your coworkers on your DevOps team to see if there are certain plugins, code, scripts you need to have access to in order to start configurations on your specific job.

The most important thing out of all of this is to know that it is alright to ask questions. No one worth a lick of salt will make you feel bad for inquiring how the pipeline for your application works.

No one.


The
Promotion

Levels.fyi has shown us that leveling across FAANG and other technology companies is in no way linear; it's not even close. So, it is understandable that people are confused around what it takes to takes to advance in their careers when the levels are not clearly defined.

Some companies do not explicitly tell you the levels that they have for software engineers until you start asking your co-workers, or you start getting job requirements for roles and you hear the internal lingo of, "We are going to hire this person at E4."

What the hell is an E5? How is that differently than the role of software engineer if it's different at all? How do I get to that next level? What will the pay bump be? What will my responsibilities be?

These are all valid questions, and the encompassing answer to them all is, it really just depends on the company.

'E' levels are internal leveling at Facebook.
'L' levels are used at Google and Airbnb.
Apple is "ICTn."

As you can see, it looks as though names for leveling are purposefully obscure to those internally and externally. In fact, a lot of this won't matter until you start the promotion process and start negations for salary.

Why?

There are salary bands associated with each level at aforementioned companies. A salary band is a cap on the amount of money that you can earn while you are an employee with that level. The actual salary caps are not known externally and all information that you find about them are either educated guesses or misinformation. What we do know about salary bands are that they exist, and you need to keep them in mind when you are thinking of what compensation you are looking for.

Talking to Your Manager

Talking to your manager is where the promotion process begins. It is as simple as setting up a 1-1 meeting with your manager to discuss the promotion or to wait for you're already scheduled 1-1.

A great manager will be excited that you are looking to:

1. Stay with the company
2. Take on more responsibility

Your manager should go over your current performance to give you a realistic view of how you are already handling the workload that you have now. If there are things you can improve on, they will detail those to you, and you will need to start working on these areas.

If you are already in a great situation performance wise, ask your manager directly what steps you need to take in order to be considered for the next round of promotions.

Promotion cycles differ per company, so I will not get into the details of when is the best time to ask your manager about a promotion. The general answer is that as soon as you decide that you want to work toward the promotion, let your manager know.

For example, your manager might tell you that you need to be involved in activities outside of your team more. When you are going for more senior roles, your visibility outside of your team is important. Your manager wants to see what positive impact that you can have outside of your team and how you can handle that responsibility.

A great way to do this is to give internal talks about something that you built and how it has had positive impact on your team and the business.

Promotion cycles differ per company, so I can't get into the details of when is the best time to ask your manager about a promotion. The general answer is that as soon as you decide that you want to work toward the promotion, let your manager know.

For example, your manager might tell you that you need to be involved in activities outside of your team more. When you are going for more senior roles, your visibility outside of your team is important. Your manager wants to see what positive impact that you can have outside of your team and how you can handle that responsibility.

A great way to do this is to give internal talks about something that you built and how it has had positive impact on your team and the business.

Volunteering

Every company I have worked for has had initiatives for employees to volunteer, and this was an aspect that I really enjoyed since a lot of my free time is spent donating my time to helping people in tech already. Volunteering is a great way to not only give back to people that are in need or looking for resources, but to also show your manager and team that you are a multifaceted person. It shows that you care about events outside of work and are willing to dedicate the time to further it on your own hours.

This may or may not be explicitly said in official paper, but at other companies that I have worked for, it was an open secret that the more that you volunteered, the more that management took that into consideration for promotions, bonuses, and raises.

Is that fair? Hell no. I dislike things being open secrets, but that is why I am telling you after all.

If the thought of volunteering seems daunting for you, start out by volunteering your time to something small. Companies may plan hackathons - offer your time organize, set up, or even judge entries. I find that doing that is not that much of a time sink or an emotional investment.

Most companies will have an entire tool and website to track your volunteering hours, but also keep track of it personally in case for some reason it doesn't. The real question now is how many hours should you volunteer? That depends on you. You can volunteer 10 hours. 20. 40. The point is for you to *want* to volunteer your time to a great cause that you care about, not just to do it for the promotion. If you loathe the thought of volunteering or taking away any of your personal time, don't do it.


The
Productivity
Guide

For listeners of Git Cute Podcast, you should not be surprised that I have included a chapter about productivity. I rant and rave about productivity to anyone that is willing — and sometimes not willing — to listen to me. I am not coming at you from a place of you need to make a certain amount of content or produce a certain amount in a day at your job. I am someone that has ADHD, and organization is key for me to keep myself on track for the week and sprint.

When you become a senior software engineer, productivity and keeping track of all of your moving parts is essential. Maybe you are thinking, "Oh, I can just remember to do this and that," and if you can, I do envy you. However, when you are in the middle of completing a task and get pulled into a sync up or other meeting, you are bound to miss something.

So, how do I keep track of my tasks for the week so that if anything pulls me away that I can still manage to get what I need done for the week?

The Pomodoro Method of course!

The Pomodoro Method

Well, honestly it's the Pomodoro Method and a lot of cute Hello Kitty planners and notebooks from Erin Condren. The Pomodoro Method has helped me be able to focus in short bursts and still manage to juggle my tendency to switch between tasks if I get bored along with still completing them. When I found out about this method by ordering a productivity journal off of Amazon on a whim 3 years ago, it was a huge relief off of my shoulders. I had found a way to manage myself between the distractions and the brain fog attributed to my autoimmune disease. I had cracked the code!

This is something that has worked stupendously for me. Neurodivergent individuals are not a monolith, so it is possible that this may not work as well for you as it has for me. However, if you can take anything away from this chapter that will help you even an iota, then the information was worth sharing.

1. Write down the tasks that you would like to complete for the day at the beginning of your work hours.
2. Set a timer for 15 minutes to start and eventually work up to 25 minutes.
3. After 25 minutes, track that you completed one Pomodoro and what you were able to complete.
4. After 25 minutes of focused work, take a 10 minute break and eventually work down to 5 minute breaks at your own pace.
5. Repeat the cycle until you are through all of your tasks.

Why the modifications?

I found it hard to stick to the original Pomodoro method constraints. The act of working for 25 minutes straight without looking at my phone, getting up to clean something that I found bothering me, or any other distraction was challenging. Much like how you don't go straight into the gym and attempt to lift the heaviest weight, I found that starting at 15 minutes was more achievable, and the more that I did practice the Pomodoro method, the easier it was to lengthen the amount of time of focus.

I also modified the break time because after forcing myself to focus for that long, I found that it was mentally draining and needed to do something that wasn't related to that task in any way for longer.

You will not get through all of your tasks that you have written out on the first try. You might not even get through all of them on your fifth try either, but finishing them all is not the point. The point is making it easier for you to keep track of the extra work that you will have. It is also nice to have a written log of all of the tasks that you have so it is easier to make that clear in meetings with your manager about the amount of work that you have been doing because let's be real: not everything that you do is tracked in Jira or your task management system.

This is not a practice that should happen at all, but a few things slipping through the cracks is inevitable. It is great to be able to reflect back on the week and say, "wait a minute, I did a lot!" even when you think you haven't accomplished much at all. We are our own harshest critics.

I decided to design Pomodoro pages that I end up modifying in my own Productivity Journal that I bought off of Amazon. It gives you the breakdown of the number of tasks for the day. I chose five, but it can be as few or as many as you would like to track. The circles off to the side of the tasks are your Pomodoros that you want to keep track of. The idea of keeping it to five is that if you believe a task will take more than 5 x 25 minute bursts, then the task needs to be broken down into a smaller piece to make it easier for you to accomplish.

I added a section at the bottom for you to write that went well for the day - not anything that you did not accomplish. We want to be kind to ourselves even if we don't complete everything that we wanted in the way that we wanted to.

I hope this will give you an idea of what to look for in your own productivity journal, or you can print these out and give it a try without spending the money! The best thing about methodologies is that you do not need to strictly subscribe to them and can let them go if it is no longer serving its purpose.

Pomodoro Page Date _____

1. ○ ○ ○ ○ ○

2. ○ ○ ○ ○ ○

3. ○ ○ ○ ○ ○

4. ○ ○ ○ ○ ○

5. 

Celebrate! What went well today?


Thank
You

Before I was a software engineer and before I had started thinking about going to school for fashion, I loved writing. I would write anything that my heart desired at the time: sonnets, poetry, short stories. My first true dream career was to become a journalist after going to Emerson College to receive a Bachelor's in Journalism, but life doesn't always go to plan.

Society dictates what we view as success and at a young age. I expected to be well into my career, married, having children by the age of 25. My life truly did not begin until the age of 27, and I am now 32 and thankful because being able to write a *book* about a topic I am truly passionate about and to help lovely people such as yourself is much more fulfilling.

Success is ultimately how you define and what you make of it. This book is guidance on what steps I took and how I handle being a senior software engineer now.

So, dear reader, thank you.

Thank you for believing that I would bring value to you while you are navigating the early stages of your career or perhaps you are a senior engineer now and want to know how to succeed even more. Remember that everything that are in these pages are guidelines. You are the deciding factor in how your path.

Good luck and have fun.

Jocelyn Harper is a coding boot camp graduate and the founder, creator, producer, and host of Git Cute: a software engineer podcast that she started when she was doubting her technical prowess. She is currently a Senior Software Engineer for PayPal and resides in Wilmington, Delaware with her cat Luna. You can follow her on Twitter @javavvitch.