The Software Engineer's Guidebook

Part I Developer Career Fundamentals

Chapter 1 Career Paths

1. Type of tech company
    - **Big tech**
        - Examples: Apple, Google, Microsoft, and Amazon.
        - Career paths: Beyond staff engineer level.
    - **Medium to large tech companies**
        - Examples: Atlassian, Dropbox, Shopify, Snap, and Uber.
        - Career paths: Beyond staff engineer level
    - **Scaleups**
        - Examples: Airtable, Klarna, and Notion
        - Venture-funded, later-stage companies with product-market fits.
    - **Startup**
        - Venture-funded companies which have raised smaller rounds of funding, and aiming for a product-market fit.
        - Startups are inherently risky.
            - They often lack meaningful revenue and are dependent on raising new rounds of funding to operate.
        - Upsides
            - Most freedom to software engineers
            - Offer employees equity are high-risk/high-reward places.
        - Downsides
            - Offer the least stability.
    - **Traditional, non-tech companies with tech divisions**
        - Examples: IKEA, JPMorgan Chase & Co, Pfizer, Toyota, and Walmart.
        - Upsides
            - More job stability
            - A better work-life balance
        - Downsides
            - Lower compensation than Big Tech and many scaleups.
        - Career paths: Beyond staff engineer level are rare.
    - **Traditional but technology-heavy companies**
        - Examples: Broadcom, Cisco, Intel, Nokia, Ericsson, Mercedes-Benz and Saab.
        - Upsides
            - Offer complex engineering challenges that can be very satisfying to work on as an engineer.
        - Downsides
            - Less desirable to work at than Big Tech or scaleups.
    - **Small, non-venture funded companies**
        - Examples: Bootstrapped companies, family businesses, and lifestyle businesses.
        - Upsides
            - Friendly, stable places to work

    - Downsides
        - Are not high-growth and are conservative in hiring.
- **Public sector**
    - Examples: Governments
    - Upsides
        - More job stability.
        - Compensation is usually clearly communicated and follows a formula.
    - Downsides
        - A slow, bureaucratic approach
        - Need to support legacy system which are hard to change.
        - Hard to move from a government job to the private sector.
- **Nonprofit**
- **Consultancies, outsourcing companies and developer agencies**
    - Rent software engineering expertise via an agency or outsourcing provider.
    - Examples: Accenture, Capgemini, EPAM, Infosys, Thoughtworks, and Wipro.f
    - Upsides
        - The easiest places to get hired.
        - Provide training for less experienced engineers.
    - Downsides
        - Don't offer paths to above staff engineer level.
        - The scopre of work is limited to work the customer sets.
        - Not much focus on good software engineering practices.
        - Hard to switch to product-focus companies later.
- **Academia and research labs**

2. Typical software engineering career paths

- **Single-track career path**

Level	Individual contributor	Manager
1	Software Engineer	
2	Senior Engineer	
3	Staff/Principal Engineer	Manager
4		Director
5		VP of Engineering
6		CTO

- **Dual-track career path**

Level	Individual contributor	Manager
1	Software Engineer	
2	Senior Engineer	
3	Staff Engineer	Manager
4	Senior Staff Engineer	Director
5	Principal Engineer	Senior Director
6	Distinguished Engineer	VP of engineering

7	Fellow	Senior VP of Engineering
8		CTO

- **Different approaches on career path**
    - Individual contributor
        - Always stick to IC path.
    - Engineering manager
        - Switch to engineering manager from senior or staff engineer.
    - Switch between IC and manager track
        - Go back to being an engineer after a manager role, and perhaps repeat in future.

3. Compensation and “tiers” of companies

- **The combination for total compensation**
    - Base salary
    - Cash bonus
    - Equity
        - Public company, this is liquid.
        - Private company, this is illiquid.
- **Tiers**
    - <u>Tier 1: Local market</u>
        - Concept
            - Companies benchmark against the local market.
        - Group
            - Local startups
            - Small, non-venture funded companies
            - Traditional non-tech companies with tech divisions
            - Public sector
            - Nonprofits
            - Consultancies, outsourcing companies, and developer agencies
            - Academia and research labs
    - <u>Tier 2: Top of the local market</u>
        - Concept
            - Companies aim to pay at the top of the local market to attract and retain standout local talent.
        - Group
            - Some medium-sized tech companies (optimize compensation for the local market)
            - Some scaleups
            - Some startups (healthy funding and a regional focus)
            - Some traditional companies with tech divisions (double down on tech investment)
    - <u>Tier 3: Top of the regional/international market</u>
        - Concepts
            - Companies aim to pay the best across the region, compete with peer tier 3 companies, not local rivals.
        - Group
            - Big tech
            - Most medium-sized tech companies
            - Well-funded scaleups (compete for talent with big tech and midsize tech companies)

        - Startups with strong fundings
- **Tradeoffs between tiers**

Area	**Tier 1 (local)**	**Tier 2 (top of local)**	**Tier 3 (top of regional)**
How hard to get a job	Easiest	More challenging	Very challenging
Performance expectations	Usually reasonable	Often demanding	Almost always demanding
Career paths as an individual contributor	Usually up to senior or staff	Sometime beyond staff	Almost always beyond staff
Work/life balance	Can be a focus	Usually less of a focus	Usually less of a focus

4. Other factors need to be considered how how satisfying your job is
    - People whom you work with and team dynamics
    - Your manager and your relationship with them
    - Your position within the team and the company
    - Company culture
    - The mission of the company, and what it contributes to society
    - Profressional growth opportunities
    - Your mental and physical health in the environment
    - Flexibility (Can you work remote, or from home?)
    - Oncall (How demanding and stressful is it?)
    - Life beyond work (How easy is it to “leave work at work?”)
    - Personal motivations

Chapter 2 Owning your career

1. Advices for career path
    - **Own it**
        - You are in charge of your career (no body cares about your career as much as you do).
        - Don’t hang around waiting for a manager cares about your professional development.

2. Advices for daily work
    - **Gets things done**
        - Finish the work you’re assigned, and deliver it with high-enough quality and at a decent pace.
        - Get impactful things done which make a difference to the business and your team.
        - Tell your manager and team when you get things done (also measure the business impact).
    - **Keep a work log**
        - Record key work each week, including
            - Important code changes
            - Code reviews,

        - Design documents
        - Discussions and planning
        - Helping out others
        - Postmortems
        - Anything else which takes time and has an impact
    - Benefits
        - Easier to know the top priorities
        - Feel good about stopping work at the end of the day
        - Easier to say no when some new tasks come to you but there is too much on your to-do list.
        - Performance reviews, promotions, and quantifying impact
- **Ask and give feedback**
    - <u>Ask for feedback</u>
        - Get feedback in normal ways (code review, design doc review, peer performance review, etc)
        - Be proactive in seeking feedback from teammates, manager and others with whom you work
            - Avoid asking for personal feedback.
            - Ask for feedback about something specific you did or work on.
    - <u>Give feedback to others</u>
        - Give positive feedback
            - Call out good work!
            - Be specific
            - Only give positive feedback when you mean it
        - Give critical feedback
            - Focus your observation on a situation and its impact
                - Describe the situation, that you observed and the impact it had.
                - Ask for their opinion for ways to reduce it possibility of it happening again.
            - Avoid saying “you should do this”
                - Avoid to give your teammate instructions unless you are engineering manager.
                - Help them come up with a solutions.
            - Give negative/constructive feedback in-person
                - People may misunderstand you if you provide feedback over email or chat.
                - Talk with them in person or via video, so you can see their reaction.
            - Make it clear from the start you’re on their side
                - Ask them to hear you out first
            - Make it clear it’s just your observation, which they can ignore if they wish
                - They are free to ignore if they think your feedback is invalid.
            - End the discussion positively
                - End the discussion with positive feedback or thanking them for being so open in listening to you.
    - <u>Receive poorly delivered feedback</u>
        - Insist on specific examples of what the feedback applies to.
        - Clarify the impact.
        - Ask for suggestions
        - If you disagree: explain why this is the case.

3. Advice for working with manager
    - **Have regular 1:1 time with your manager**
        - Share the work you've been doing
        - Share the work log of wins and challenges
        - Discuss your professional goals
        - Ask about their challenges, and how you could help them and the team.
    - **Tell them, don't assume they know**
        - Tell the manager the work you've done, don't assume they know
    - **Understand your manager's goals**
        - Ask about the biggest upcoming challenges of the next month and half year, there might be opportunities to help.
    - **Deliver what you agree on, and give updates when you cannot**
        - Finish a task by the agreed date and let your manager know when it is done.
        - Notify your manager upfront if it will be late.
    - **Establish mutual trust with your manager**
        - Aim to be open, honest, and transparent with your manager.
    - **Get your work recognized**

4. Pace yourself
    - **Model**
        - Stretching
            - Get out of your comfort zone, face new challenges and learn new things
            - If you do too much stretch work, you will have physical and mental exhaustion.
        - Executing
            - The normal way of working: Use your skills and experience to get things done well.
            - Working in comfort zone.
        - Coasting
            - Doing less and lower quality work than you're capable of.
            - Coasting can be a temporary, short-term breather after a tough project, taking a mental break.
    - **Advice**
        - Mix up stretching, executing and the occasional period of coasting to optimize for long-term professional growth and avoid burnout.

Chapter 3 Performance Review

Chapter 4 Promotions

Chapter 5 Thriving in Different Environments

1. Types of teams
    - **Product team**
        - Role

        - Build a product for external customers.
        - Have good product skills on top of engineering skillset.
    - Characters
        - Proactive with product ideas and opinions
        - Interest in the business, user behavior, and relevant data
        - Curiosity and a keen interest in “why”
        - Strong communicators with great relationships with non-engineers
        - Offering product/engineering tradeoffs upfront
        - Pragmatic handling of edge cases
        - Quick product validation cycles
        - End-to-end product feature ownership
        - Strong product instincts through repeated cycles of learning
    - Tips
        - Understand how and why your company is successful.
        - Build a strong relationship with your product manager.
        - Engage in user research, customer support, and related activities.
        - Bring feasible product suggestions to the table.
        - Offer product/engineering tradeoffs for projects.
        - Seek regular feedback from the product manager.
- **Platform team**
    - Role
        - Build blocks that product teams use to ship business-facing functionality.
    - Characters
        - Focus on a technical mission
        - Customers are usually internal
        - Used by multiple teams
    - Upside
        - Gain more technical knowledge.
        - Wide impact.
        - More engineering freedom (no product manager interaction).
        - Less day-to-day pressure to ship new features or pducts.
        - Attract more senior-and-above engineers.
    - Downside
        - Business impact is harder to define.
        - Frequently seen as a cost center.
        - More distant from customers.
    - Tips
        - Build empathy for external customers.
        - Talk to engineers using your platform.
        - Work side by side with platform customers.
        - Do a rotation working on a product team.
        - Aim for some urgency and focus.

Chapter 6 Switching Jobs

1. Search new job
2. Interview
3. Getting Offer
4. Onboarding to a new job
    - **General guidance**
        - Take ownership of your onboarding, and not just hope to be wilcomed with a fabulous onboarding process.
    - **Suggestion**
        - *At all companies and levels*
            - If there's long notice period after signing a new contract, consider catching up with your future manager before you start. Ask for suggestions on how to prepare to hit the ground running.
            - Read the book "The First 90 Days" By Michael D Watkins.
            - Keep a work log/brag document from the start.
            - Keep a weekly journal. Write down three things you learned, and three things you don't understand during the first few months.
            - Clarify your goals with your manager for the first month, the next 3 months, and the first 6 months.
        - *Onboarding to a smaller company*
            - Clarify the work you'll do before you join.
            - Connect with the most senior person, early on. They'll often have long tenure.
            - Don't assume how things work; ask.
            - Aim to ship something in week one. There's no excuse not to at a small company.
        - *Onboarding to a larger company*
            - Get an "onboarding buddy", even if not assigned one.
            - Keep a "cheat sheet" document in which you record everything that's new.
                - Acronyms and their meanings
                - Build commands
                - Bookmarks to key resources
                - More
            - Familiarize yourself with the company's tech stack (new language and new framework).
            - Accept that many things don't make sense at first, especially if you arrive from a smaller company.
        - *Onboarding to a senior or above role*
            - Connect with a senior-or-above peer. Understand how they organize and measure their work.
            - Prioritize be productive on the team's code base, and becoming hand-on with it.
            - Clarify 3, 6, and 12-month expectations with your manager.
            - Understand current and future projects and the priorities. What is the team working on, why doesn it matter, how doesn all this fit into the company's strategy and priorities?
            - Find and browse outage documents like postmortems with are relevant to your team and are, in order to understand problem areas, and not be caught off guard when familiar issues reoccur.

        - Start a list of engineering teams worth knowing. When you hear a new team mentioned, add its name to this list. Then, introduce yourself to a member of that team and learn what they do, and how your teams are connected.
    - *Onboarding to a staff-or-above role*
        - Meet your team early and make a friendly impression.
        - Meet peer staff+ engineers early (You will rely heavily on them to understand how things work).
        - Figure out what's expected and what's not in your role.
        - Clarify 3, 6, and 12-month expectations with your manager, and also with your skip.
            - What are the team's top priorities and the biggest problems to address?
        - Spend time with product folks to understand their current concerns and plans.

Part II The Competent Software Developer

Chapter 7 Getting Things Done

1. Getting things done
    - **Focus on the most important piece of work**
        - Identify the most important piece of work work
            - Rule: The most important project on your plate
            - When you identify it, ensure you deliver that piece of work within the agreed timeframe.
        - Learn to say "no"
            - Use the template to reply: "Yes, I'd like to help, BUT…"
    - **Unblock yourself**
        - Identify when you're blocked
            - Rule: If you go more than 30 mins ~ 1 hrs without meaningful progress
        - Try different unblocking approaches
            - Do "rubber-ducking"
                - Explain your problem and the approaches already tried to a "rubber duck."
                - Verbalizing a problem sometimes triggers a train of thought, leading to a solution.
            - Draw it on paper
                - Visualizing a problem might prompt ideas for what else to try.
            - Reading documentation and references
            - Use an AI tool
            - Search online for a similar problem
            - Post a question on a programming Q&A site
            - Take a break
            - Start from scratch or undo all changes
        - Get support to unblock yourself
            - Ask internal team support: "Hey, I'm stuck on this problem. Can anyone lend a hand, perhaps pair with me on it?"
            - Sometimes, you'll be blocked by waiting on somebody.
        - Escalate without harming the personal relationship

        - Escalation: Ask your manager or somebody higher in the chain of command to communicate your message using the authority of their position.
        - The approach to escalating:
            - Explain: Explain why help is needed, giving context so they understand the importance
            - Ask: If nothing happens, ask why.
            - Warn: If still nothing happens, I raise the prospect of escalation.
            - Escalate: And if still nothing happens, I'd escalate by involving my manager, the other person's manager, or both.

2. Planning

- **Break down the work**
    - Think about stories, tasks, and subtasks
        - Break down the work into small parts, and then challenge yourself to make them even smaller.
    - Prioritize works that gets you closer to working functionalities
        - The sooner you have something you can test end-to-end, the sooner you can get feedback.
    - Don't be afraid to add, remove and change tasks
        - Remember the goal is to build software that solves customers' problems, not to close tasks.
- **Estimate the duration of your work**
    - Common categories of work and how to estimate them precisely:
        - *Similar work to that of a colleague*
            - A developer who's coded something similar can estimate how long it should take.
            - You will take more time to complete the work, as you need to learn some of the context.
            - You should consult them and break down their approach.
        - *Refactoring*
            - If you've done refactoring before, you can use that experience for a baseline time estimate.
            - Ask teammate if they've done similar refactoring work.
            - Timebox the work (Timeboxing means assigning a period of time to a task and only working on it for that long, and no longer).
        - *Greenfield work using a technology you know well*
            - Use the technology or framework you know well, so the only biggest risk is that business requirement are unclear.
        - *Integrating with a system you don't know well, using a well-known technology*
            - Consider prototyping first, and making an estimate only when you've built a simple proof-of-concept that tests the other system or API.
            - If prototyping is not possible, provide best-case estimation and worst-case estimation.
                - If there's pressure to give an estimate, provide a worst-case estimation
        - *Building something simple with a mature technology you don't know well*
            - Ask an engineer who has used the technology. They can give pointers on where to start and help you estimate how long it should take.
            - Pair with a developer who knows this technology well.
            - If pairing is not possible, always use a timeboxed estimate to get up to speed with the technology. Also make this timebox big enough.

        - *Building something simple with a new technology you don't know well*
            - Build a proof-of-concept with the new tech, and don't make an estimate until you're confident using it.
        - *Building something complex and integrating with a system you don't know well, using an unfamiliar new technology*
            - Prototype. You can make an estimate with learnings from the prototype.
            - Break up the work more. You can efficiently divide and conquer the unknowns.

3. Seeking mentor
    - **Type of mentors**
        - The dedicated mentor
            - People you have interacted with before, or someone whose work you admire and who falls in line with your goals.
        - The ad hoc mentor
            - People with whom you may not work closely, but whose work intersects with your path.
        - The internet mentor
            - People who share their career journey and learnings on their blogs, books, podcasts, etc.
4. Keeping your "Goodwill balance" topped up
    - **Goodwill balance**
        - When you help others, your goodwill balance increases.
        - When you ask for help, your goodwill balance decrease.
    - **Advices when decreasing goodwill balance**
        - Avoid using your goodwill balance too quickly.
        - Go prepared when asking a colleague for help. Explain the problem clearly, summarize that you've tried so far, and what your next step would be if they cannot help.
        - If they're busy, respect their time as they probably have other priorities. Ask them for a quick pointer.
        - When someone helps you, and it solves your problem, thank them.
    - **Advices when increasing goodwill balance**
        - Make yourself available to others. Sit with them and help solve their problems.
        - Share your expertise to make others' work easier.
5. Taking the initiative
    - **Concepts**
        - Engineers pick up smaller or larger pieces of work which were not asked of them.
    - **Approaches**
        - Document unclear things
        - Volunteer for investigations.
        - Investigate interesting tools or frameworks your team could use.
            - Share learnings with the team.
            - Investigate tools or frameworks available at your company, or ones which other teams use.
        - Talk with your manager about upcoming project.
            - Express interest
            - Get better sense of priorities
            - Learn what to investigate in advance

Chapter 8 Coding

1. Practice coding

- **Advice for practice coding**
  - Write code regularly
    - Code daily and aim to work on meaningful tasks and problems every day.
    - If you don't have the opportunity to code regularly
      - Try to find ways to do like picking up an extra project at work
      - Move to a team which does a lot of coding
      - Work on a side project
  - Ask for code reviews
    - Aim to get feedback on all the code you write, even if code reviews aren't required.
    - If more experienced people are around, you can ask them for extra feedback.
    - Ask a colleague to pair for code review.
    - AI tools are also useful sources of coding feedback.
    - Make notes on the code review feedback.
    - Talk to the reviewer directly if you don't understand the reasoning.
  - Read as much code as you write
    - Get involved early in code reviews across the team or the codebase.
    - Go through each change colleagues make, understand what it does, and make notes on the approaches taken.
    - Follow code changes and code reviews by teams whose codebases you depend on.
    - Read actively developed open-source code which works with a similar language as you use.
  - Code some more
    - Build a side project.
    - Complete a tutorial/training with coding exercises
    - Do coding challenges
    - Do regular code katas

2. Readable code

- **The most important principles for pragmatic engineers**
  - Readable code
  - Well-tested code
- **2 main goals when writing code**
  - The code should be correct, meaning it produces the expected result when executed.
  - Other developers should find it easy to read and understand.
- **Advice for writing readable code**
  - *Naming*
    - Use self-explanatory, concise names. Stick to consistent naming approaches in line with the naming used on the codebase.
  - *Well-structured*
    - The codebase is easy to navigate as functions, classes, and modules follow a logical structure.
  - *Keep the code simple (KISS = Keep it simple, stupid)*
    - The simpler code is, the easier it is to read and understand
  - *Single responsibility*

        - Aim for functions to do just one thing, and for classes to have a main responsibility.
    - *DRY principle (DRY = Don't repeat yourself)*
        - Avoid copy-pasting code
        - If you find the need to reuse, consider refactoring the code so it follows the single responsibility principle.
    - *Comments*
        - Explain the "why" on the code, not the "how" in the comments.
    - *Continuously refactor to maintain readability*
        - Readable codebases stay readable thanks to continuous refactoring.

3. Quality code

- **Advice for writing quality code**
    - *Use the right level of abstractions*
        - The abstractions hide implementation details from other parts of the code.
            - The interface reflects a simpler, more abstract view of the module's functionality and hides the details. Reduce the cognitive load on developers who use the module.
            - Information hiding makes it easier to evolve the system.
    - *Handle errors, and them well*
        - Have a consistent error-handling strategy
        - When in doubt, use defensive programming
            - Validate inputs, especially from users.
            - Expect invalid responses.
            - Expect malicious inputs
            - Expect exceptions and errors
    - *Be wary of "unknown" states*
        - Allowlist
            - Create a list of successful responses and assume everything else is a failure.
        - Blocklists
            - Create a list of failure responses and assume everything else is a success.

Chapter 9 Software Development

Chapter 10 Tools of the Productive Software Engineer

Part III The Well-Rounded Senior Engineer

Chapter 11 Getting Things Done

Chapter 12 Collaboration and Teamwork

Chapter 13 Software Engineering

Chapter 14 Testing

Chapter 15 Software Architecture

Part IV The Pragmatic Tech Lead

Chapter 16 Project Management

Chapter 17 Shipping to Production

Chapter 18 Stakeholder Management

Chapter 19 Team Structure

Chapter 20 Team Dynamics