Software Engineering - The Soft Parts

JUNE 14, 2022

Addy Osmani
@addyosmani

Today I'll share some of the software engineering "soft skills" I've learned from my first 10 years on Google Chrome, where I am a Senior Staff Engineering Manager. On my 10th anniversary, I wanted to reflect on some of lessons that have stayed with me. I hope these prove useful to you during your career.

Becoming a good engineer is about collecting experience. Each project, even small ones, is a chance to add new techniques and tools to your toolbox. Where this delivers even more value is when you can solve problems by pairing techniques learned on one project with tools learned working on another. It all adds up.

I'll preface this by saying I am unimportant, unwise and unoriginal. YMMV :)

Skip to lessons most relevant for leaders/managers

▶ Table Of Contents

Learning new things

The following pointers should help most junior or mid-career developers move forward, deal with changing technology, and build complex systems while following the standard processes in the software engineering paradigm and discovering new best practices. Apply first principles when you can. Learning to break down problems into smaller pieces is one of the most important skills in life.

Mastery

Technical mastery implies a high ratio of value shipped to hours worked.

This means you can discern tasks that add value and help your team focus their energy in that direction. It also means you know how to avoid work that doesn't provide the team/company value - the best engineers can even steer whole teams away from work that isn't that useful.

I often get asked, "How do I know if I'm making the best use of my time?". There will almost always be tasks you can fill your time with to "feel" busy. The real trick here is making sure you are working on the right things. If you want to move mountains, focus on tasks that move the needle, even if that movement is small.

Some questions you can ask yourself:

- What are my goals? Are the tasks I'm focused on lining up with those goals?
- Could there be something I could do differently or better?

Even asking yourself such questions can be extraordinarily powerful.

Think critically and formulate well-reasoned arguments

Critical thinking is the ability to use cognitive skills to think independently in order to make thoughtful decisions. Invest in this skill to improve your clarity of thought.

As engineers, we can sometimes rush to solve a problem right away so it feels like we're making progress or looks like we're being responsive to stakeholders. This can introduce risks if we aren't fully considering causes and consequences. Put another way, critical thinking is thinking on purpose and forming your own conclusions. This goal-directed thinking can help you focus on root-cause issues that avoid future problems that arise from not keeping in mind causes and consequences.

In broad strokes, some of the questions I like to ask based on critical thinking are:

- How do we know we're solving the right problem?
- How do we know we're solving the problem in the right way? (i.e. balancing rigor and efficiency, given our understanding of the problem and constraints)
- If we don't know the sources of our problem, how can we determine the root cause?
- How can we break the key question down into smaller questions that we can analyze further?

- Once we have one or more hypotheses, how do we structure work to evaluate them?
- What shortcuts might we take if we're under constraints (time pressure) without unduly compromising our analytics rigor around the question?
- Does the evidence sufficiently support the conclusions?
- How do we know when we are done? When is the solution "good enough"?
- How do I communicate the solution clearly and logically to all stakeholders?

I've found these questions often help. Sometimes we'll address the symptom of a problem, only to discover there are other symptoms that pop up. At other times, we might quickly ship a solution that creates more problems later down the road. With a lens on critical thinking, we might challenge assumptions, look closer at the risk/benefit, seek out contradictory evidence, evaluate credibility and look for more data to build confidence we are doing the right thing.

For example, a common mistake I've seen engineers make is assuming correlation implies causation (i.e. just because two things correlate does not necessarily mean that one causes the other). A critical thinker might push back on assumptions such as this, asking why we believe them to be true.

Critical thinkers:

- Raise mindful questions, formulating them clearly and precisely
- Collect and assess relevant information, validating how they might answer the question
- Arrive at well-reasoned conclusions and solutions, testing them against relevant criteria and standards
- Think open mindedly within alternative systems of thought, recognizing and assessing, as need be, their assumptions, implications, and practical consequences
- Communicate effectively with others in figuring out solutions to complex problems

Note: Critical thinking has both aspects of being a "soft skill" and a "hard skill", so is included in this write-up.

Building a strong base

Master the fundamentals and apply them repeatedly to acquire new skills.

The long-term value of learning the fundamentals is that they are transferable. The short-term is that they help you make better decisions and can make code more efficient.

Transferable skills

Transferable skills are those you can take with you from project to project. Let's talk about them in relation to the fundamentals.

The fundamentals are the foundation of any software engineering career. There are two layers to them - macro and micro. The macro layer is the core of software engineering and the micro layer is the implementation (e.g. the tech stack, libraries, frameworks, etc.).

At a macro level, you learn programming concepts that are largely transferable regardless of language. The syntax may differ, but the core ideas are still the same. This can include things like: data-structures (arrays, objects, modules, hashes), algorithms (searching, sorting), architecture (design patterns, state management) and even performance optimizations (e.g. eager vs lazy evaluation, memoization, caching, lazy-loading etc). These are concepts you'll use so frequently that knowing them backwards can have a lot of value.

At a micro level, you learn the implementation of those concepts. This can include things like: the language you use (JavaScript, Python, Ruby, etc), the frameworks you use (e.g. React, Angular, Vue etc), the backend you use (e.g. Django, Rails, etc), and the tech stack you use (e.g. Google App Engine, Google Cloud Platform, etc). There involve details that can be valuable to gain expertise in to be effective, but are not always transferable.

By learning the fundamentals, you gain the skillset and tools to then ignore the fundamentals and grow.

That said, pragmatically, no one has time to learn everything at the beginning of their careers. There comes a point when you shouldn't over-index on the fundamentals and learn what is needed to actually build applications for the real world. This is where the "learn by doing" approach comes in.

Efficiency

Understanding the fundamentals well can help you write more efficient code. This includes concepts such as time complexity (the time it takes to run your code), memory usage, and the trade-offs between performance and maintainability. These ideas allow you to make trade-offs that are helpful when building any reasonably large application. Speed is often critical for modern applications and can often impact end-user experience in a noticeable way.

Better decision-making

Having a good understanding of macro and micro fundamentals can help you make better decisions.

You can use the knowledge you've gained to make better decisions about which technologies to use and which ones to avoid based on the goals and constraints of any project. This can help you avoid the pitfalls of choosing the wrong technology or the wrong tool for the job.

> "You haven't mastered a tool until you understand when it should not be used." - @kelseyhightower

Software engineering involves thinking about many different layers - the core languages, the implementation, the infrastructure, the tools, and the people. Only having a surface-level appreciation for these layers can absolutely let you build faster. But really knowing the fundamentals (including O(n) time complexity) can help you go that much farther, especially when the landscape of languages and frameworks changes over time.

Related reading:

- The value of fundamentals in Software Engineering
- Why learning the fundamentals matters
- Learn the fundamentals of a good developer mindset

Focus on the User and the rest will follow

Start with the user experience and work backwards to the technology you need.

Steve Jobs once famously said, "you've got to start with the customer experience and work backward to the technology. You can’t start with the technology then try to figure out where to sell it.".

This quote has stuck with me, because as engineers, it's far too easy to start from a place of wanting to use specific solutions - whether due to popularity, developer experience or just personal preference - and try to find a way to rationalize using them. Instead, we should focus on who we're building for, what problems they have, and how the current options available are falling short.

Great user experiences come from combining both points of view - both the customer and technology. Show people what you think they want and pay attention to what they say. There is of course, immense nuance to this problem space - what engineering choices will allow you to deliver a great experience on mobile hardware? What choices will impact engineering velocity? or scale? or hiring?. Ultimately, we benefit from having a relentless focus on the customer first and then navigating what allows us to address their needs within the constraints we have to work with.

The best software is built by engineers who have empathy for their users.

Business success depends on customer satisfaction which often translates to user experience in the case of software. Understand how the end-users experience the product or service. Make sure your solutions do not hamper their ability to do their jobs efficiently. If you are in a position that allows you to interact with end-users directly, attempt to understand their needs and pain points better.

Upgrading your skills

Choose what's right for your use case and not the flavor of the month.

It's OK to use "boring” technology (what’s tried & tested) vs. the hype train. Languages, frameworks and libraries often evolve. Choose what helps deliver a great final product. When starting off a new project, begin with "boring" tech (but well understood) and then intentionally decide out of it to select the best tool for a problem.

When picking new skills to learn or use, don't be afraid to choose something that's boring and not in the news. FOMO may not be productive when it comes to technology whether it be languages, frameworks, and libraries and tools. While it's important to know what to use, your main goal is to deliver an excellent final product. Please don't chase the new and shiny technologies unless you think they add value to your solutions. At the same time, don't shun something because it is not being talked about enough.

Take advantage of new projects to learn new tech.

At the same time, personal and hackathon projects can be a great opportunity to learn new tech. Many of us have fewer opportunities to start something completely brand new, as opposed to working on an existing codebase where many decisions have been made. Such projects can be a low-risk way to research new tech, evaluate its strengths and weaknesses (at a small scale) and build up some first-hand knowledge that could be valuable to you in the future.

Be curious and never stop learning

Write about what you learn. It pushes you to understand topics better. Sometimes the gaps in your knowledge only become clear when you try explaining things to others. It's OK if no one reads what you write. You get a lot out of just doing it for you.

Learning should be a continuous process - people who claim to know everything about a particular technology are often not experts. Real experts are proficient with the technology but realize there is always scope for learning and improvement. Curiosity drives learning - so if you are curious about a new framework, google it, read the docs, try the tutorials, read the source! Learning need not happen in a classroom. It can happen anywhere, anytime. Take half an hour each day to read a chapter from a textbook, listen to a technology podcast, read development blogs or learn a new programming language.

It's powerful for leaders to admit when they don't know something.

Having this confidence lowers the expectation that Senior Engineers have to know everything. You absolutely don't need to have all the answers, but being able to admit you're human and are committed to figuring out how to solve problems with your team is the important part.

Leaders also admit when they make mistakes.

It's important to teach your team how to handle mistakes with humility and the desire to learn and improve. The real world isn't perfect and it's totally okay to show your team it isn't perfect to prepare them for it.

Be a caretaker, rather than an owner.

In the early stages of open-source projects, it's common to think like an owner. You often directly own proving out value, working on features, answering issues and advocacy. This can be great for getting something adoption, but may not be the best way to scale a project later down the line when staffing changes or your own time gets limited.

After the initial crunch, another way to think of evolving your role is towards being a caretaker, rather than an owner. A caretaker might focus on scaling themselves out. This can be done by sharing as much knowledge as is possible with other maintainers, contributors and the community (via design docs, code comments, and other documented best practices). It also helps to grow the pool of reviewers with enough context to make the right decisions when you're no longer as involved.

This is often what projects need to be sustainable many years down the road.

Depth and breadth of skills

Consider if being a jack of all trades and a master of one is right for you.

One of the greatest skills you can master is learning how to learn. This should be a priority over say, just going deep on particular programming language or framework. It helps to stay curious. Once you have experience with this, you may question if you should aim to be a specialist or a jack of all trades.

I personally like the idea of T-Shaped engineers. These are engineers who are a deep expert in one or a small number of skills (the vertical bar of the T), but who have a basic understanding of many other skills needed to build and run a product (the horizontal bar). Some teams like to rotate team members through a range of different specializations to build more T-Shaped team members.

I've found that in mid-large sized teams, it's been effective to have folks who possess specialized skills in one area and the skills, versatility, and aptitude for collaboration to fill in for other people if necessary.

To experience is to learn

When learning a new language, focus on building something tangible with it that gives you first-hand experience.

If you are learning a new language, you need not memorize all its syntax or documentation to become a good developer. It's more important to know how to solve problems. Earn experience by writing a lot of relevant code or learning from existing code. The results should help you write efficient code in that language. As mentioned here, "The main value in software is not the code produced, but the knowledge accumulated by the people who produced it". However, please don't experiment in production when experimenting with new technology.

Technical Complexity

Generic vs Specific code

Write code specifically for the problem at hand, but try to spot places where you can afford to make it a little generic.

Often, we attempt to code things as generic as possible, and end up making what is effectively code soup that doesn't help accomplish the problem. Instead, building specifically for this problem, but trying to spot places it can be made just a little bit more generic, has altogether eliminated times I know I would've had to refactor again later if I hadn't been thinking of it.

There are several principles commonly discussed that talk about design complexity. From the extreme programming world, you have:

- YAGNI or You aren't gonna need it, which states that programmers should not add functionality until it is necessary.
- Do the simplest thing that could possibly work - to make rapid progress rather than plan for the future.

Both these principles aim to prevent over-engineering. However, these principles could be abused to create multiple simple solutions which do not integrate well.

At the other end of the spectrum, you have the Abstraction principle that aims to reduce duplicate structures in the code whenever practical through abstraction and generalization. I prefer to take the middle ground between extreme abstraction and extreme simplicity by making code just a little generic. The AHA (Avoid hasty abstractions) principle promotes a similar idea.

Deep modules

Write code that solves complex problems for other developers but exposes functionality through a lucid interface.

If you are an API designer or developer - your responsibility is to provide an interface to simplify complex functionality for other developers. The purpose is defeated if the interface is too difficult to understand and imposes a cost on the programmer using it. This idea is reflected in the concept of Deep Modules - "The best modules are those with the greatest benefit and the least cost. The benefit provided by a module is its functionality, and the cost of a module is its interface."

While the simplicity of an interface is desirable, complex problems sometimes require complex code to solve them (this if not a universal rule, but is often true). This complexity is better off embedded in code. When complex functionality is abstracted, the value provided to the end-user or interface user is higher.

An API with multiple visible functions and classes encompassing some functionality is more complex and challenging to search when compared to another with the same functionality implemented using fewer public functions/classes. New functions and classes add to the cost of the interface for maintenance programmers and library users.

Learning on a maintenance project

When working on legacy code in older systems, understand the difference between code that should stay and code that should go.

Any senior engineer should make an effort to understand the difference between code that should stay and code that should go.

It's important to understand the difference between code that should stay and code that should go. Large, long-term production systems are going to have some bad code or code that doesn't have a good reason to remain anymore. It's healthy to appreciate why something is there (good reason? bad reason?). Remove the bad code and keep the good code.

I've worked at many companies where folks assume what is legacy code is untouchable or is designed the way it is for a good reason, lost to the sands of time. This can lead to fear of change where you just keep on adding abstractions to a weak foundation.

The software industry has reached a stage where many projects deal with the maintenance and migration of old or legacy systems. Don't get frustrated if you find yourself in one such team. There is much domain-specific knowledge that you can gain by looking at old code. While there may be good reasons for older code/validations existing in production, it's healthy not to assume every single line is still relevant.

Some software engineers are wary of touching code working in production for fear of introducing a bug. So they include conditions and repeat some code for newer use cases. Such workarounds may save time at that instant, but they become a maintenance nightmare over time. Don't assume that the existing code is blessed or infallible. There may be some aspect of scalability or efficiency previously overlooked that you could address.

Learning on a green-field project

Experiment, innovate, fail fast and get better at solving problems.

Your learning journey is entirely different when you are tasked with building a system from scratch. As you start prototyping or implementing features iteratively, you learn what works and what doesn't. Agile methodology and the fail-fast principle help you validate your ideas earlier with fewer resources. They enable you to divide and conquer complex problems.

Definition of done

Defining what is "done" is time-saving because it helps you estimate the effort required, plan for development, and avoid unnecessary revisions later.

Another Agile principle that comes in handy when dealing with complexity is agreeing on the definition of done. In addition to meeting user requirements and acceptance criteria, this could include other conditions such as code reviews, testing, documentation, etc.

Phased roll-outs

A single large release may be divided into a series of lower-risk well-understood rollouts.

Rollout plans are as important as the architecture and the code when planning releases for large-scale production systems. Phased releases with iterative development help you better manage risks due to significantly large changes. You can also create release strategies with the development and testing strategy to have an end-to-end plan for a complex release.

Systematic debugging

When debugging, you should try to resolve the issues systematically and rigorously to address all the test conditions.

Always read the error messages (and the stack trace). There's likely valuable information in there that will help you isolate the problem so you can resolve it. A surprising number of engineers ignore the insight error messages can give offer before looking for debugging help. Assume your machine is telling you what's wrong and may be correct, rather than assuming that making small edits and constantly re-running the code will fix the problem faster. If you write a solution that throws an exception and aren't reading the exception carefully, you may be wasting time. Often the error or exception message is a big hint what's actually wrong.

Design Docs

Importance of design docs

Design documentation should not be an afterthought but an integral part of the software engineering process.

A design document is a ubiquitous tool that can help you gain consensus from your peers or other teams who need to interface with your part of the system. Feedback from others enables you to identify gaps and refine your design. Design docs also serve as a valuable aid for engineers who would join the team in the future. It would help them understand the problem space and the trade-offs and alternatives considered when designing the solution. Design docs provide a space to capture all participants involved in the design and their contributions as part of the document history. This helps others understand who drove specific decisions and whom to contact for further elaboration.

Documentation process

Coordinate reviews for the design doc and compare the design as it evolves with the original doc to verify that all the relevant constraints are being addressed.

While one person can document the design, the actual design process occurs during a series of whiteboard meetings, random in-person discussions, slack threads, or email/phone discussions. Only after you put it down on paper can you identify contradictory commitments and see if the different parts you had discussed fit together. After creating the initial draft, coordinating a review ensures that all parties concerned are on board. However, it may happen that the implemented design does not match what is documented because something changed along the way.

Communication

Be humble, communicate clearly, and respect others. It costs nothing to be kind, but the impact is priceless. Some may say good communication costs energy and thoughtfulness. There should be more energy for compassion.

Communication is a critical part of the soft skills or people skills required to become an effective, productive, and efficient software engineer. Miscommunication can lead to incorrect functionality, incompatible code, or offensive team dynamic. Communication can help people understand requirements better and prevent issues from being escalated.

The world may imagine software engineers to be people who spend their day writing code. However, to ensure that our products are helpful to others, we have to synchronize our efforts with others in the team and business and user expectations. This makes collaboration and communication the critical pillars of our jobs.

Junior developers mostly communicate with other team members, test engineers, and team leaders to share ideas and discuss alternatives for problem-solving. As we grow in our careers, the quantity of communication required to do our jobs effectively goes up. The number of emails, meetings, and public talks increases. We have to communicate with business leaders, managers, stakeholders, and team members. The more specialized your work, the greater the risk that others will not easily comprehend you.

Customized communication

Use language, concepts, and levels of detail relevant to your audience.

Whatever our level of understanding of a problem or a situation, when we discuss it with others, we have to tailor our words so that they can quickly grasp what is relevant to them:

- When talking to a business person, talk about the business impact of what you are doing. Avoid using overly technical jargon.
- When talking to engineering management, communicate the technical impact or challenges.
- When talking with a decision-maker, you describe available options and their implications and risks, not the details of how the options work.
- When providing a status update, be aware of what else has transpired and how your update is relevant to the project goals.

The same principle applies when writing emails and presenting to a larger audience. Write what is relevant to the person or people receiving the message. You may have to defend your ideas when presenting. Phrase questions and responses in a thoughtful manner. Knee jerk responses are usually detrimental to communication.

Being kind and considerate

Being nice is a superpower - wield it.

Being calm, kind, and helpful can take you further than cutting someone off. Be nice to people within your team as it will help make the team stronger and successful. Be friendly to those outside your team as well. Treat all the functions in your organization (HR, finance, or marketing) with equal respect. You may not help them directly, but you can always understand their work and empathize with them. Congratulate or appreciate others when they have done well or received accolades. Kindness is contagious. People you have been nice to are more likely to respond to any requests for assistance in the future.

Be liberal in telling folks they're doing a great job.

While it's important to give feedback when improvements are needed, it's also critical to give folks positive feedback if things are going well. This helps your team know that they're making a difference and are valued.

The power of NO

Saying no is better than overcommitting.

Most of us are not great at saying "no" where more work is concerned. It is either because they don't realize that 'no' is an option, or we enjoy the challenge. However, overcommitting is a liability as it can lead to delays. Letting the other person know what is already on your plate and providing a reasonable estimate of how long it would take is a sign of respect. It gives the other person a chance to consider their options - ask someone else or extend their timeline. Management will not ask you to deliver something in record time if they know that it will significantly impact the quality of the product. If you are a senior manager, empower your team to say no to bad ideas.

> "A senior developer (or any productive person) is good at saying no. People will ask for more of your time than you can spare. You can gently but firmly say no, direct people elsewhere (delegate), or ask people to discuss with your manager whether more of your time can be allocated to help them." [1]

You can't please everyone - be extremely mindful when saying "yes" vs. "no".

The counterpart to leaders saying "no" to everything is saying "yes" to everything and failing to set clear boundaries. Taking on more scope than can actually be executed reasonably well with your current resources can lead to heartache for you, your team and eventually your customers. This is particularly important for leaders to absorb as others will look to you to set the norms on when they should say "yes" or gently push back.

Acceptance and respect

Admit when you don't know something and be open to asking for help, even from juniors.

It’s okay to admit when you don’t know something. One of the most important skills in software is being able to find answers and learn from them

As a senior leader, learn to accept that juniors around you may be more aware of a project's technical nuances. It is okay to admit when you don't know something and let the junior engineers explain it. They will respect you more for your honesty and interest in learning, and you will get a better picture of what is going on and add value to it. As a junior engineer, you should explain technical concepts to seniors either openly or behind closed doors, depending on their comfort level.

Information sharing

Use meetings and Q&A sessions to ask the right questions, exchange knowledge and inform the team.

When running a meeting, don't be the only person talking. Meetings are an opportunity for others to share ideas and provide honest feedback - so listen and make space for others to contribute.

Junior engineers may shy away from asking too many questions. If you are a senior, you can prompt them to ask the right questions by bringing up the context. When fielding questions, let the person asking know that you are glad they brought it up.

Flexibility

Defend your opinions stridently but also review them every time you have new evidence that contradicts them.

Listening to other opinions is a key part of communication. It's essential because there may be more than one solution to a problem. Rather than being adamant about your views, listen and evaluate other options. Maybe they will bring forward an aspect that you had earlier overlooked. Paul Saffo's principle of "*Strong opinions weakly held*" tells us to defend our opinions stridently but also review them every time we have new evidence that contradicts them. It is a scientific evidence-based method that does not consider the person who came up with an idea or opinion.

Maintaining a record

A friendly email after an informal meeting helps reaffirm the key points or commitments made during the discussion.

A downside of exclusively verbal communication is that it can be forgotten or misremembered. Keeping a record of everything that transpired and getting a sign-off on relevant discussions eliminates this risk. If you or another person has agreed to help with a task, then confirm the deadline via email to ensure that everyone, including your supervisor, is on the same page. Keeping a record of such unplanned work would also be helpful during an appraisal discussion.

Good faith

Know when to keep quiet and observe the dynamics at play.

There may be situations where you don't understand some decisions, or they do not make sense for technical and business reasons. This may happen in multi-team discussions. Participate in good faith and assume that people will not risk being publicly malicious. Possibly you do not have the complete picture, or they have different priorities. Ask questions and state your opinions without getting angry or frustrated about the final decision.

Seniority

We aspire to grow in our career, either in our role or capabilities. While some are interested in senior technical positions, others wish to go for leadership or management roles. In either case, there are some key characteristics that people higher up in the seniority order exhibit. Throughout your journey, you may have mentors to guide your growth. Here's my approach to developing the qualities that can prepare you for a senior role.

Seniority and strategic thinking

Don't fail to make decisions or act given uncertainty.

Very often you will find that it's better to make any decision rather than no decision at all. It at least allows others to know what direction you're leaning towards. Sometimes as leaders we don't spend enough time reflecting on what decisions our teams are expecting us to make, but are not, because we're not 100% certain we have all the facts. We can and should try to build as complete a picture of the details needed to make confident decisions as we can, but this isn't always possible (e.g. in a time crunch). This can lead to long periods of waiting/uncertainty for teams where it can help to actively better yourself on how to make decisions even given limited information.

Leaders are people who have broadened their horizons to think strategically and lay out the roadmap for others.

Your ability to think and plan strategically and apply your thinking to larger scopes should ideally grow with experience. As an individual contributor, you may focus on assigned tasks or the features you are working on. The impact of your work extends beyond specific tasks and projects as you climb the ladder. When weighing options, you learn to look at the larger picture in terms of benefits and constraints. The scope of application of soft skills also increases. For example, if earlier you were making decisions for a team or addressing other engineers in your team, your choices and communication impact multiple teams as you grow.

Leading by example

Teach your team to fish. Don't always solve problems for them, but gently guide them to develop the skills to solve themselves.

Engineering leaders empower. As you become more senior, it helps to give up your toys, coach, delegate & enable your team to succeed. It's how you scale effectiveness. This can be done by asking good questions more than (just) giving answers.

You lead by example when assessing challenging problems and ask relevant questions when someone offers a solution.

Seniors in the technical track are responsible for coordination, negotiation, and consensus-building within and outside their teams. They contribute to improving the overall team output and not just their own. As a senior engineer, you may occasionally code to acquire new skills or understand the ground reality, but that is not a part of your job description. Instead, you are someone who ensures that there aren't any missing pieces in the architecture diagram or loopholes in the code. You should be able to explain your decisions with evidence or reasons for how they would provide technical or business value.

A senior engineer should be good at architecting software systems and human systems or teams. You can lead a diverse group of engineers, delegate tasks to them, mentor them to care about code quality/performance/simplicity. You can give feedback when required and defend them where necessary. At the same time, you should be able to market yourself, your work, and your ability to solve challenging problems to gain visibility in the organization. Overall, *you should manage your relationships with people within your team and the management*.

Scale your effectiveness.

The world’s best engineering feats are accomplished by a team of engineers, not individuals. So, if you are trying to accomplish more, or show you're ready to become more “senior” in your company, multiply your effectiveness through collaboration and mentorship. Demonstrate how this adds value not only to yourself, but to other members of your team.

I felt like I was on the path to being a senior engineer at Google when I realized that to scale myself, I had to shift my mindset from "me" to "we". By collaborating with others, sharing what I learned, and focusing on lifting the skills and the expertise of people around me, we started to get so much more done.

When you start out as an individual contributor, you may not have a dedicated “team” you lead, but you can find like minded people to collaborate with (maybe aligned with your goals) and work together to accomplish a lot more than you could alone. As you get more senior, you evolve this thinking towards building out teams and continuous growth of your effectiveness.

Imposter syndrome

Accepting that it is okay to make mistakes, not know answers, or seek guidance can help overcome imposter syndrome.

All of us have felt inadequate for a particular role or job at some point or other. *Imposter syndrome* is genuine and very common. It can affect even those who are evidently successful. You might feel like an imposter even as others look up to you for advice. You may never be cured of the syndrome, but it will push you to be curious and learn new things.

Mentoring

Mentoring others

Be the guardrail by giving timely information so that your mentees do not end up in a completely incorrect place but instead gain mastery by doing things themselves.

You may find yourself in a mentor or mentee role at different times in your career. Mentoring need not be a formal process. You can look for opportunities to mentor others or allow yourself to be mentored even informally. Mentoring others gives you a chance to learn people skills yourself. Following are some key points to remember while mentoring.

Mentoring is about guiding people to discover answers themselves rather than giving them ready-made solutions. Allow your mentees to experiment when solving their problems. They are in the best position to assess risks and benefits. However, please give them the tools required to find answers. If it's a technical problem, suggest ideas and options to try out, but let them do the actual legwork. Let them share what they think and listen closely, ask questions, and engage in a dialog.

If someone fails to figure out solutions by themselves, show them how you would approach the problem and why you chose a particular pattern to solve it. Teach them how to analyze results or debug issues. Share your thought process as you diagnose the problem, try out solutions, implement them and debug them. Share your problem-solving techniques and not just the answer.

Organization-wide mentoring

Ensuring that mentorship is a part of a senior engineer's role also helps retain crucial domain knowledge when someone moves to another team, position, or organization.

Suppose you are sincere about mentoring someone, and it is also a part of your job description. In that case, you have to make time in your schedule for mentoring activities. This will allow you to do it properly and make a difference in your mentees' lives. Some organizations may also have a defined process for the mentor/mentee discussions based on the career progression ladder and requirements for each step.

Mentee's role

Mentors can offer you advice, but you are the only one who can take the initiative and act on any advice to manage your career and growth.

Suppose you are a junior engineer who wishes to grow in an organization. In that case, there is only one piece of advice for you. Find strong mentors who can help you navigate the growth ladder.

You will come across coaches, mentors, or colleagues you look up to throughout your career. They can offer you advice on how to develop your skills but you are the one who can act on it. When assimilating advice, beware of blanket statements regarding technology. Different situations need different principles, and what worked for one project may not work for another.

Effective teams

Building Trust

Trust can unite team members to work towards a common goal while bureaucracy will divide them.

When engineers come together for open-minded and unbiased brainstorming, it paves the way for new ideas and different perspectives that drive innovation. This leads to highly efficient and productive teams. However, effective collaboration among team members is only possible if the communication and relationships amongst team members are healthy. Here are some pointers for building, maintaining, and becoming a part of effective teams.

Building trust is the most critical component of team building. Trust amongst team members across the hierarchy is necessary to get things done fast and for teams to be effective. Team members may use different software engineering processes such as reviews and tests for reviewing project health. However, these processes become tedious and bureaucratic without trust. For example, you will probably nitpick less during a code review if you trust an engineer with some code.

Understanding the business model

Understand the business impact of the change.

When you receive a new set of requirements, understand the motivation behind them. Don't skim over the 'purpose' or 'business goals' section of the requirement documents. Ask questions to understand the business model and its relation to these requirements. An existing codebase or talking to subject-matter-experts (SMEs) can provide insights about the domain and the architecture. Refer to the documentation or map features and use cases to system processes and data flows.

> "A lot of software engineers love to solve problems with a technical challenge. It can be more rewarding to understand the business side of things and be able to come up with cost effective solutions. Remember that your users/customers are also people trying to do their job, and get through the day or week, just like you are. Try not to make their lives harder than they already are." [1]

Increase your impact

Perceptiveness and astuteness about the business-software equation increases the impact of your work.

Getting a 360-degree view of the business and the product helps you contribute positively to the team and your projects. If you understand how sales or marketing think, you are better equipped to make the right decisions and do high-impact work. As your impact on a team's success increases, your job satisfaction and pay will improve. Your seniors will recognize your ability as a self-starter who can work independently without supervision and drive overall efficiency by doing what is suitable for the team, project, and business.

Work/life balance

If you are someone who has gained mastery over technical abilities, human factors, and domain knowledge, your skills as a software engineer will invariably be in demand. People in your team and organization will consult you. In addition to your engineering commitments, you will become a victim of collaboration overload. Ad-hoc requests can eat into your time and stop you from doing what you are passionate about.

Time management

Optimize your calendar for Deep Work

Block time on your calendar for focused deep work. I've done this for years and found it highly effective for writing design or strategy docs or just working on a hard technical problem. Deep work is distraction-free, high concentration work that creates a lot of value in a little time. Cal Newport's Deep Work covers this topic well.

Attention residue is an idea Cal talks about being why working deeply for extended periods is so beneficial. Every time you're switching from one task to another, a residue of your attention remains stuck thinking about the previous task. This makes it hard to work with the necessary focus on what's really important.

Deep work maximizes the amount of productivity you squeeze out of a limited time by focusing on a *single* task. No distractions, no Twitter, no chat or email. You reserve deep work for tasks that are cognitive heavy. I highly recommend trying it out.

I’ve also found that changing my location can sometimes help with deep work. We can sometimes fall into the trap of associating a specific place (like a desk, room or building) with a particular kind of task and adding a little variety can help reinvigorate us.

Avoid fracturing your working hours

When an hour of work gets broken up into chunks of just a few minutes due to distractions, you become stressed. Identify the causes of distractions (whether it's you or others) and fix it. Otherwise your day won't be as productive.

Working in excess isn’t part of a good work ethic.

You can never work harder than everyone in the world. Many companies hold up overworked employees as the "standard", falsely concluding this is the same as having a good work ethic. Success comes from many factors and not just overworking.

Constantly trying to outdo your standards is unrealistic.

I have been guilty of this a lot. If you want to develop calm and avoid a crazy work environment, you must get comfortable with enough. As a manager or lead, your team might take your lead on how to approach this. Being OK with enough can set a good example.

Time is finite. Instead of trying to seek more time, eliminate unnecessary tasks.

Lots of guidance talks about rearranging work better. The real problem is trying to accomplish too much to begin with. Ruthlessly eliminate work that's unnecessary and time wasting vs. trying to manage limited time.

You don't have to know every last thing going on.

Many of us are afraid of missing out on every new story or update. This is one reason people get obsessed about checking Twitter, Reddit, Instagram etc every hour. I've certainly been through this. In reality, most of this information is just not that important. Instead, try switching to summary views of this news or set limits on how often you check it.

There are further thoughts on this topic in "It doesn't have to be crazy at work" by Jason Fried.

It's best to proactively save yourself from exhaustion by learning to say no, knowing when to stop, and planning your time to include breaks between work.

Time management and maintaining a good work-life balance are crucial for engineers at all levels. Regularly working overtime can lead to burnout and stress. Stress can cause other physical and mental health complications. It may be tempting to resolve an issue before you call it a day, but it could become a habit over time.

Encourage breaks, holidays, and vacations both for yourself and your team.

Your health and family are vital. If you realize this as a senior engineer and set an excellent example for others in a team, it will foster overall wellbeing and happiness. On the other hand, exhaustion and burnout can lead to toxicity in the workplace.

Update estimates as your understanding of problems improves

There will almost always be a customer or stakeholder for your work that will want to know when a project or task can be delivered and if this cost is worthwhile. This is totally reasonable. Sometimes they want to match a deadline or there are dependencies elsewhere that need to support your engineering work requiring planning.

Software deadlines are notoriously difficult to predict accurately. Deadlines that are based on estimates should only be given when projects are at a particular stage. When time passes, estimates should get updated as we learn more about the team's ability to solve the problem (the "informed" estimate). The first estimate (the "sizing") is often the least reliable, however it is a starting point that can get refined over time. This initial estimate is often very conservative - should the product requirements, UX or dependencies be unclear, a larger conservative estimate is often helpful for that first "size". I often have the best success here when such estimates are approached collaboratively with PMs so we are all on the same page about refining them.

The trouble with software estimates is when the first rough estimate gets cemented as the plan of record rather than a first draft. When teams on the critical path adopt it but view adjustments to the estimates as a hiccup by engineering (vs. step 1/n of an informed estimate) this can be an issue. Once a project has the greenlight, figure out the details better - this may mean an estimate of three months becomes two (or four) based on a deeper understanding of what will address the requirements.

You almost always want the estimates driving your schedule vs. having the schedule drive the estimates where possible. In my teams, while we do sometimes have unmovable deadlines (e.g. a conference), if the estimates overshoot these dates that's (often) fine - changing messaging (e.g. "preview"), framing ("coming in the near future") or punting to a future are always options we can discuss with leadership. I of course acknowledge this is not always trivial. When schedules do try to be pulled in, you can break work up into must-have vs. nice-to-have (and move these to a future sprint) then review if the must-haves meet your deadline.

Should schedules still be too tight, there are other questions you can ask, such as "Can we add additional engineers to this project?" and "is there a large reduction of scope that would still make shipping on time compelling?".

Cancelling projects is sometimes the right (if uncomfortable) call.

I hate this one, but cancelling a project can sometimes be the healthiest long-term decision for your team and organization. This is especially true if it is cancelled before it has had a chance to launch, gain traction then ultimately has to be deprecated because staffing on it can no longer be sustained. In case folks are wondering, yes I read Killed By Google. Aim to minimize the circumstances leaning up to projects getting cancelled as much as you can. I recently cancelled a multi-year project and it was rough.

When can this happen? You can make decisions about investing in a new project that are the right ones for a point in time. At that point in time, the stars may well have aligned (market-fit, organizational buy-in, staffing commitments) for it to completely make sense. A year down the line, things can change - the market, leadership, importance of the project. **It's critical to regularly check on whether the assumptions you made when a project started continue to remain true through its lifetime.**

The more you can sustain confidence the assumptions are still true, the better a shot you have at the project being able to successfully launch and continue to be supported. Cancellations are hard for a number of reasons, not least that there are real people with real emotions who invested in building what they had hoped would launch. As a leader, navigating folks back out of a cancelled project onto others that successfully launch is complex, but important for getting folks back to a place of psychological safety, trust and happiness. On the customer side, be mindful of user trust and how your long-term decisions can impact this.

On technical debt: An ounce of prevention is worth a pound of cure

Titus Winters defines technical debt as "the difference between the code and systems we have today vs what we wish we had" with certain kinds of debt having a higher impact than others. Some debt can be due to mistakes that weren't caught early (oversight), others due to what was learned after-the-fact (hindsight) and some are due to the landscape of technical systems changing (context).

I have found that consistently prioritizing tackling technical debt is sometimes hard as you can't always quantify the bugs that didn't manifest or outages that didn't happen because you "paid debt down enough". Sustaining team interest in this kind of work and rewarding it during performance reviews is also really important. The cost of the "cure" however can be so much higher once problems really start to pile up over time. Similar to pollution, over the course of multiple years, prevention of technical debt is a cheaper strategy than mitigation at a later point.

What can you do to prevent debt building up? Technical leads should regularly devote time in sprints to clean-ups and paying debt down in addition to building out new features. Reviewers should be cognizant of pushes for short-term velocity that may actually lead to problems further down the line. Managers and directors should be mindful of approving new projects that overlap with existing ones, unless you're certain the trade-offs are worthwhile(e.g. addressing debt in existing system just isn't worth it vs. building something new). Monitoring of project health is really important on top of all of this.

Without breaks and a good work/life balance, you or your team can burnout.

Burnout is a form of exhaustion due to workplace stress that hasn't been successfully managed. I've seen many engineers hit burnout during the pandemic due to work stress, but it has always been present in tech. These days, I ask my reports, "how are your stress levels doing? what can I do to help?" in each 1:1.

My experience with burnout is that it happens slowly and ends in apathy. You slowly start to feel low on energy, unmotivated, and exhausted all the while trying to cope with work stress as best you can. You question if there's something wrong with you, but fail to realize your body is working overtime to compensate for the lack of energy you have. You keep pushing yourself harder and harder, but eventually it feels like there's not much left to give.

I hit burnout about 5 years ago, however am happy to say I turned it around. What led to it? It was an avalanche of things. I had been putting work first for years, working longer and longer hours and not saying "no" enough. I never took enough breaks or vacations. I was averaging 5 hours of sleep a night. When I was home, I was so low on energy that I wasn't "being present" nearly as much as I should have for my family.

The "fix" was doing the opposite of those things: take breaks, get more sleep, squeeze more value out of the hours I do work, delegate better, and have a clear "stopping time" for work.

As managers, to avoid our reports burning out, I think it's important to try encouraging our teams to use their vacation time, take breaks and periodically check folks are actually doing okay where stress is concerned.

Executing in large organizations can feel slow. There are ways to navigate this.

I've had many conversations with engineers that boil down to "why is shipping X moon-shot in (big org) so hard?". There's a great analogy by Alex Komoroske comparing large organizations to slime molds. That is to say, executing even simple things can start to feel way way slower than you might expect, due to coordination headwinds. Organizations have complex systems, structures and dynamics and headwinds rise when the number of people who must coordinate on a project increases.

There are many forces at play here, including underestimation of the difficulty of other's tasks (e.g. if they're building a dependency). You can't ignore these problems as it can make the dysfunction spread. One way to work through such headwinds is to decouple things as much as possible so they can land on an OK timeline and eventually converge towards shipping X.

Rather than tackling all of X from the get-go, you can avoid solely shooting for the moon-shot (large risk efforts) and instead define roof-shots (safe steps to unlock value) that move you closer to your goals. I strongly recommend reading Alex's deck if this problem sounds familiar.

Focus on problems vs. projects

Let's imagine your users have an unsolved need (e.g. a problem). When you're an engineer attached to a particular project, it's normal to ask how *your specific project* is going to solve this problem (a local maxima). In a large organization with projects of similar shapes, it's very possible to see multiple engineers trying to independently think this way ("how does my project solve this problem?"). When you own a portfolio of projects however, this may not quite be clear-cut. What if users may use many of your projects together? Wouldn't it be weird if they each solved the problem in a slightly different way unaware of each other's approach? Instead, you want to ask "what's the right end-to-end solution to this problem?" and walk back to what project or changes to a series of projects will holistically address this need best. It may require getting folks working on the multiple related projects to collaborate more deeply. This can however result in a better, less confusing story for your users at the end of the day.

Conclusion

> "Surround yourself by excellence and work with people who are the best as what they do" - Brian Staufenbiel

Invest in friendships and relationships with folks you can learn from. Be open to their guidance, mentorship, their successes, and their failures. Never be afraid to ask for help or insight. In a lot of cases, it's just a question away.

At every stage, remember that mastery over technology, business domain, and human resources at a given organization has to be cultivated over time. An organization cannot hire masters from another and expect them to be productive from day one. If you are a good engineer, you will contribute to your organization's growth. In return, new avenues will be available to you, allowing you to acquire new skills and grow yourself.

With thanks to Leena Sohoni, Joshua Cruz, Kara Erickson, Jeff Posnick, Houssein Djirdeh and Sriram Krishnan for their kind feedback and contributions.

Addy Osmani

Addy Osmani (Adnan Osmani) is an engineering manager at Google working on Chrome. He is author of open source projects like Yeoman, TodoMVC and Material Design Lite. He has also written books like Image Optimization, and Learning JavaScript Design Patterns.

Tweet Share