Software Engineering after the Vibe Shift

Sean Goedecke

April 2025

Contents

Preface

I've read a lot of books about software engineering, but I learned all the most important things I know about the job from word-of-mouth or bitter experience. This book is my attempt at writing those lessons down. Given that, I want to begin with some caveats.

I think I've been pretty successful. I've been promoted to staff twice, worked on some very high-priority projects, and generally had a strong relationship with my management chain everywhere I've worked. But I'm certainly not the most successful engineer in the world. If you're doing better than me, you should be emailing me with your advice.

I've been in the industry for ten years. I've worked at two large tech companies: five years at Zendesk, and five years at GitHub. Those companies are more alike than different, so I've drawn some conclusions about engineering at similar large SaaS-based web-dev tech companies. But the place you work at might be very different, so my advice might not apply.

Finally, I've spent most of my career at senior or above, so that's where my advice is pitched at. In particular, I have no good advice for interviewing - I've interviewed at two tech companies in my life, and both times I got the job so I stopped interviewing. I've been on the other side of the interview process during the 2010s, but not since. So most of my advice in this book will be about succeeding at the senior+ level, not about finding a job or getting started in the industry.

Dangerous advice

I'm a big fan of "sharp tools". These are tools that are powerful enough to be hugely helpful or harmful, depending on how they're used. Most forms of direct production access are in this category: like ssh or kubectl access, a read-write prod SQL console. It's also possible to give "dangerous advice". Dangerous advice is dangerous because (like sharp tools) it takes competence and judgment to use well. Giving the wrong person dangerous advice is like giving the wrong person production SQL access - they might go off and do something enormously destructive with it.

This is a book full of dangerous advice. For instance, I advise you to:

- Make your own decisions about what to work on
- Deliberately break written company rules sometimes
- Take strong positions even when you're uncertain
- Identify as a bit of a 'grifter'
- Deliberately avoid all non-shipping activities

I worry about some readers making the wrong decisions about what to work on, or breaking the rules in the wrong way, and so on. I don't like the idea that I've contributed to harming someone's career. But I think dangerous advice is important to write down *somewhere*.

Most career advice is fake

The main reason why is that **strong engineers crave dangerous advice**, for the same reason they crave sharp tools. Most career advice is fake: written to avoid liability, or to impress people. If you're serious about getting things done, you'll end up doing some of these things

because they're obviously helpful.

It can be deeply alienating to *know* things don't work the way they officially should, but to have nobody to talk about it with. When I was a more junior engineer, I really appreciated the few peers I had who were willing to get real with me (off the record).

Managers can't tell it to you

Another reason why is that **managers will almost never give you dangerous advice**, even if it's what you need to hear. If a manager tells you to ignore company policy, and you do it wrong - for instance, if you post in Slack that you're doing it and your manager said it was OK - then that's bad for them. In fact, it's much worse for them than it is for you. Tech company leadership often views engineers as useful idiots. Managers are expected to be professionals.

However, lots of managers wish they could give you advice like this. They certainly appreciate it when you follow it. I've never been a manager, but it must be incredibly frustrating to manage strong engineers who would be much more effective if they approached work a little more tactically (and a little less according to the written job description).

Dangerous advice requires courage to follow. It's very high-risk, high-reward, and is thus disproportionately useful to strong engineers (and harmful to weaker ones). If you don't feel comfortable following it, you definitely shouldn't. But if you're already operating like this sometimes and wondering whether you're making some horrible long-term mistake, I don't think so. Or if you are, I'm making it too! Let's get to it.

Software engineering after the 2010s

The good times are over

For most of the last decade, being a software engineer has been a lot of fun. Every company offered lots of perks, layoffs and firings were almost unheard of, and in general we were treated as special little geniuses who needed to be pampered so we could work our magic. That's changed in the last two years. The first round of tech layoffs in 2023 came as a shock, but at least companies were falling over themselves to offer generous severance and teary CEO letters regretting the necessity. Two years later, Meta is explicitly branding its layoffs as "these were our lowest performers, good riddance". What the hell happened? What does it mean for us?

Why did the vibe shift?

In the 2010s, interest rates were zero or close to zero. Investors could thus borrow a *lot* of money. Much of that money was spent on tech companies in the hope of outsized returns. Tech companies were thus incentivized to (a) hire like crazy, and (b) do a lot of low-risk high-reward things, even if that ends up wasting money. Tech companies definitely did *not* have to be profitable. In fact, they didn't even need to make money - they just had to acquire users, or at least hype, to drive up the valuation of the company itself. In that environment, throwing money at their software engineers (in the form of paid trips, in-house chefs, and huge comp packages) was a sensible business decision.

In 2023, this underlying economic situation reversed: interest rates went up to around 5%. Tech company incentives completely flipped: now it's suddenly important to be profitable, or at least to make lots of money. That means it's not wise for most companies to hire like crazy, or to continue throwing near-unlimited amounts of money at

their software engineers.

I think that's a sufficient explanation for the vibe shift all by itself. What about COVID? It helped, but it wasn't the root cause. Two years (or thereabouts) of people staying inside more meant much more engagement with tech products, which meant much more money flowing into tech companies. *Everyone* was hiring during COVID. Once that short-term boom finished, companies naturally wanted to get rid of some of those engineers, which is what triggered a lot of the initial layoffs. However, I do think that even without COVID, we'd still be in something like the current situation. Companies were constantly hiring pre-2020 as well.

This idea that AI is taking software engineering jobs or contributing to layoffs is - as far as I can tell - currently pure fantasy. I do believe in the power of AI, and I wouldn't be surprised if it takes software jobs at some point in the future, but it's certainly not behind the vibe shift in software engineering right now.

What does that mean for us?

I think a lot of software engineers right now are planting their feet and refusing to change. After ten years of their opinion being consulted on big company decisions, they're trying to hold on to that power. I have respect for anyone who stands up for what they think is right at personal cost. I just want to stress that there *is* going to be a personal cost to not going along with the vibe shift, particularly for more junior or more vulnerable engineers. As someone who lives in Australia, I feel pretty vulnerable myself.

The biggest thing to internalize is that **companies now are actually trying to focus**. In 2015, there was a lot of appetite to do everything at the same time: building out new product lines, transitioning from a product to a platform, making significant open-source contributions,

working on a top-tier developer experience, and so on. In 2025, most of these initiatives have been abruptly defunded in order to put more resources into a handful of bets that the company executives actually care about.

During the 2010s, it was as if companies *were* their software engineers, and were interested in the same things as their engineers were. A lot of engineers were fooled by this into identifying strongly with their employer. But this was a mirage: in part caused by companies' desire to attract and retain talent, and in part by there being no real pressure on companies to say no to anything. Now the mirage has vanished. Companies are their executive leadership, and their executive leadership are interested in a much smaller set of things.

If you were an engineer who loved working on your company's open-source libraries, it's probably sensible to confront the fact that the company never really cared about it that much. When interest rates were zero, it was worth doing because most things were worth doing. At 5% interest rates, most open-source work doesn't meet that bar. In other words, **your interests now conflict with your company's interests**.

It's okay for your interests to conflict with your company's. You get to decide what you care about, and what you're willing to fight for. But when you act in ways that don't further your company's interests, you risk being seen as ineffective or unreliable. In 2025, that makes you vulnerable to being laid off.

Is there a silver lining?

The good news is that tech companies now live in (or at least a lot closer to) the "real world". It was nice to be pampered, but there was a fundamental ridiculousness about it, even at the time. I know a lot of engineers who found that offputting, including myself. It's why many

engineers found the TV show *Silicon Valley* hard to watch - the satire was too real to laugh at. It was mainly embarrassing.

If I had to choose, I'd definitely choose to return to the job market of the 2010s, so I can be paid more to work less and have more job security. I'm not an idiot. But the silver lining to *actually having to ship* is that you're no longer living in a dream. If you're realistic about how things work, the job of software engineering becomes much easier to understand:

1. Providing value to the company gets you rewarded
2. Not providing value to the company gets you punished
3. "Value to the company" means furthering the explicit plans of your company's executives

It's not much of a mission statement! Certainly nothing on "making the world a better place". But it has the comforting solidity of the truth. The good thing about the music finally stopping is that you don't have to worry about when it's going to stop.

Knowing where your salary comes from

With the recent flurry of US federal firings, many people are pointing and laughing at the Trump-voting federal employees who are just now finding out that they've voted for themselves to be let go. How could you have this poor a mental model of *what your job even is*? Well. In my opinion, many software engineers are operating under a mental model that's just as bad, and are often doing the equivalent of voting for the person promising to fire them.

I won't quote the tweets, but I regularly see stories like "I convinced my idiot bosses to *finally* let me do only tech debt work, and would you believe they fired me after a few months?" Or "I've been busting my ass on this underfunded project and I still got a bad performance review". Or "it's so unfair that I haven't been promoted - look at all this amazing accessibility/standards/open-source work I've been doing!" The basic structure goes like this:

1. A bright-eyed engineer joins a tech company, excited to go and make the world a better place
2. They throw themselves into various pieces of work that don't make money (improving FCP performance, better screenreader support, refactoring)
3. Their managers desperately try to redirect them to work that does make money, causing a long frustrating power struggle
4. Eventually the bright-eyed engineer gives up and unhappily focuses on Profitable Product X, or
5. The bright-eyed engineer leaves or is fired, and goes on Twitter to complain about their important work not being valued

The complaint in this story is basically equivalent to "I can't believe Trump is firing me from the IRS when I voted for the guy". It represents a fundamental misunderstanding of what tech companies *are*.

What are tech companies?

So what is the right understanding? Let's start as simple as possible. Tech companies are run by small groups of people with the goal of making money. Successful tech companies make a lot of money, by definition. They hire software engineers in order to continue doing the things that make that money, or to do new things that make more money.

At successful tech companies, engineering work is valued in proportion to how much money it makes the company (directly or indirectly). Patrick McKenzie has an excellent post on this:

> Profit Centers are the part of an organization that bring in the bacon: partners at law firms, sales at enterprise software companies, "masters of the universe" on Wall Street, etc etc. Cost Centers are, well, everybody else. You really want to be attached to Profit Centers because it will bring you higher wages, more respect, and greater opportunities for everything of value to you. It isn't hard: a bright high schooler, given a paragraph-long description of a business, can usually identify where the Profit Center is. If you want to work there, work for that. If you can't, either a) work elsewhere or b) engineer your transfer after joining the company.

Companies value work more the closer it is to a profit center. I don't think you necessarily have to work *in* profit centers (in most tech companies, that would mean abandoning your engineering title and role). But you need to demonstrate your value to the profit center in order for your own work to be valued. I'm not saying you need to do this to keep your job. Companies pay lots of people who don't deliver value, sometimes for many years. What I'm saying is this:

If your work isn't clearly connected to company profit, your posi-

tion is *unstable*

In other words, you're probably depending on a kind-hearted manager (or CEO) who personally values your work. When they leave, you're in trouble. Or you're depending on a large company not really caring to check if a small team is bringing in profits. When they look, you're in trouble. Or you're depending on a cultural climate where your work has temporary cultural cachet (e.g. biofuels in the early 2000s). When that changes, you're in trouble. The only way to have a stable position is to be connected to the way the company makes money.

Connecting your work to profit

In order to know if your work is connected to company profits, you have to know two things:

1. What is your company's business model? How do they make money?
2. How does your work support the business model?

Publicly traded companies must publish their business model and finances every year, which means you can either just go read that or read what people are writing about it on business blogs, magazines, and so on. (If you're working for a well-known company, you can probably just ask a LLM). If you're working for a private company, that can be harder, but usually it's not too hard to get the gist. For instance, the broad strokes of where Valve get their money from are pretty clear: it's Steam, not their first-party games.

Being an engineer at a company will give you much more visibility into the business model. For instance, you can run analytics queries that identify the ten largest customers. Often you won't have to run these queries directly - they'll be shared among the product-and-business folks, in channels that most engineers have no interest in. It's worth trying to learn about the business model. For instance, if I were work-

ing for Valve, I'd want a much clearer answer than "it's Steam": I'd want to know which kind of games were bringing in the most money, the distribution between new users and existing users, and so on.

Once you have an idea how your company makes money, you can gauge how your work supports it. If you build a product that many people are buying, this is easy: calculate how big a percentage of company profit your product is. What if you don't build a product? Say you're on the accessibility team, or the German localization team. In that case, you ought to figure out why your company is investing in these things. For example, working on accessibility might be valuable because:

- It allows us to sell to (e.g.) visually-impaired customers, growing the total addressable customer base by X%
- It enables us to meet specific regulatory requirements that allow us to sell to large enterprise customers (e.g. governments)
- It makes us look good, or at least avoids us looking bad
- It's just good work that is worth doing because it's the right thing to do

Some reasons are only important when times are good. If the company is doing great and has more money than it knows what to do with, the last two points are probably worth spending money on. (If the company is doing *really* great, like a lot of companies in 2019, "literally anything" is worth spending money on in order to accumulate engineers). When interest rates rise, those reasons vanish.

These reasons will apply to some companies more than others. For instance, if you work at Google, the first reason is important because growing a customer base worth ~270B by 2% unlocks ~5.5B of new revenue. If Google is paying your team less than 5B in total, your team is probably making enough money to justify its existence. But if you work at a smaller company with revenue measured in millions, that

math goes the other way.

There's an obvious consequence to this: if you want your work to be valued (i.e. you don't want to be reshuffled or fired), and you want to work on personally-satisfying features like accessibility, or UI polishing, or anything else not directly connected to profit - **you need to go and work for a very profitable company**.

Delivering marginal value

When I wrote about this idea - that very large tech companies deliver marginal features as a way of slightly growing their massive addressable customer base - some readers found the idea depressing. Maybe so! But at least it's a theory for how it might be possible to work on these kinds of features and get paid for it. The alternative theory is something like:

1. Accessibility, clean code, good performance, and so on are all Good Features
2. Good Companies care about Good Features
3. I just need to keep looking until I find a Good Company and not a Bad Company

I don't think that a smart engineer who thinks about this problem will come away believing this. But lots of smart engineers don't like thinking about how their work connects to company profits, so their implicit beliefs often add up to something like this. These engineers will often go through the five-step process I mentioned in the introduction to this post. I hate to see technically strong, motivated, kind-hearted engineers run headlong into burnout for completely predictable reasons.

Summary

- It's easy to fall into the trap of thinking that you get paid for work because it's important

- You get paid for work because it makes money. If your work doesn't contribute to that, your position is inherently unstable
- If you want a stable position, you should try and figure out how your work connects to company profits, and strengthen that connection if possible
- All kinds of seemingly-unprofitable work makes money, particularly at large companies where small percentages are a lot
- If you want to work on seemingly-unprofitable work, you're probably better off working for large successful tech companies

Shipping

The core of succeeding at any tech company is *shipping*. If you don't ship, nothing else matters. If you do ship, it covers a lot of other problems you might have.

How I ship projects at tech companies

I have shipped a lot of different projects over the last ~10 years in tech. I often get tapped to lead new ones when it's important to get it right, because I'm good at it. Shipping in a big tech company is a very different skill to writing code, and lots of people who are great at writing code are terrible at shipping.

Here's what I think about when I'm leading a project and what I've seen people get wrong.

Shipping is hard

The most common error I see is to assume that shipping is easy. The default state of a project is to *not ship*: to be delayed indefinitely, cancelled, or to go out half-baked and burst into flames. Projects do not ship automatically once all the code has been written or all the Jira tickets closed. They ship because someone takes up the difficult and delicate job of shipping them.

That means that in almost all cases, **shipping has to come first**. You cannot have anything else as your top priority. If you spend all your time worrying about polishing the customer experience (for example), you will not ship! Obsessing over UX is praiseworthy behaviour when you are an engineer on the team, but a blunder if you are leading the project. You should cherish the other engineers on your team who are doing that work, and give them as much support as you can. But your primary concern has to be shipping the project. It is too hard a job to do in your spare time.

In my experience, projects almost always ship because one single person makes them ship. To be clear, that person doesn't write all the code or do all the work, and I'm not saying the project could ship without an entire team behind it. But it's *really important* that one person on the project has an end-to-end understanding of the whole thing: how

it hangs together technically, and what product or business purpose it serves. Good teams and companies understand this, and make sure every project has a single responsible engineer (typically this position is called a "technical lead" or "DRI" role). Bad teams and companies don't do this, and projects live and die based on whether an engineer steps up into this role of their own accord.

What is shipping?

Why do so many engineers think shipping is easy? I know it sounds extreme, but I think many engineers do not understand what shipping even is inside a large tech company. What does it mean to ship? It does *not* mean deploying code or even making a feature available to users. Shipping is a social construct within a company. Concretely, that means that **a project is shipped when the important people at your company believe it is shipped.** If you deploy your system, but your manager or VP or CEO is very unhappy with it, *you did not ship*. (Maybe you shipped something, but you didn't ship the actual project.) You only know you've shipped when your company's leadership acknowledge you've shipped. A congratulations message in Slack from your VP is a good sign, as is an internal blog post that claims victory. For small ships, an atta-boy from your manager will do.

This probably sounds circular, but I think it's a really important point. Of course if you deploy something that users love and makes a ton of money, you've shipped. But that's only true because satisfying users and making money is something that makes your leadership team happy. If you ship something users hate and makes no money, but your leadership team is happy, *you still shipped*. You can feel any way you like about that, but it's true. If you don't like it, you should probably go work for companies that really care how happy their users are.

Engineers who think shipping means delivering a spec or deploying code will repeatedly engineer their way into failed ships.

Communication

So if your primary job when shipping something is to make your company's leadership happy with the project, what does that mean in practice? First, **you have to get clear on what the company is looking to get out of the project**. Sometimes it's extracting more money from a small set of users (e.g. enterprise features). Sometimes it's spending money to grow the total set of users (e.g. splashy free-tier features). Sometimes it's mollifying a particular very large customer by building a feature specifically for them. Sometimes it's just an influential VP or CEO's pet project, and you need to align with their vision. There are lots of potential reasons, and if you want to ship the project you need to know which ones apply in this case. Align your work and communication accordingly! For instance, enterprise features often don't need splashy UI but are completely inflexible on requirements, end-user features need to be polished, pet projects mean you need to be in active communication with the specific mover and shaker whose pet it is, and so on.

Second, no matter the project goal, your leadership team (the people in your reporting chain who care about the project) will always have basically zero technical context about the project compared to you. That means they will be trusting you for estimates, to answer technical questions, and to anticipate technical problems. **Maintaining that trust should be your top priority**. If they don't have faith in your ability to do the job and to keep them informed, you will not ship. They'll de-risk by cancelling the project, or letting it roll out with zero attention or celebration (remember that an un-celebrated launch is not a ship!) Alternatively, they'll sideline you and go to another engineer, who will then formally or informally be the one who actually ships the project.

Either way, you'll feel it at review time and they'll go to someone else for the next one.

How do you maintain trust with your leadership team? This could be a whole article (or book) by itself, but here's my summary:

- The best thing is a track record of having shipped in the past, if you can get it
- Project confidence (if you're visibly worried, they will be too)
- Project competence. You want to aim for something like a NASA mission control vibe
- Communicate professionally and concisely, and don't make them chase you for updates: post a daily or weekly thread somewhere

It is much, much more important to do these things than for the project to ship with zero bugs on the exact deadline. If a project has to be delayed for technical reasons, in my experience you will not suffer consequences so long as you communicate it clearly, confidently (and ideally with some warning). In fact, it's paradoxically often *better* for you if there is some kind of problem that forces a delay, for the same reason that the heroic on-call engineer who hotfixes an incident gets more credit than the careful engineer who prevents one.

Getting into production

Even so, you typically still have to get the project into production. The most common problem here is missing a key detail. Sometimes it's a technical detail: maybe we rely on storing the user documents in Memcached, but many documents are multiple megabytes and will be larger than the Memcached block size. Sometimes it's a detail of coordination: maybe the platform team that owns Memcached was expecting one-tenth of the traffic our project will send them, so they call a meeting with the VPs and delay the project. Sometimes it's a

legal detail: maybe the user data is unexpectedly sensitive, and our system doesn't have the controls we'd need to handle it safely. These issues can come from anywhere, and are very hard to anticipate. Dealing with them requires a deep technical understanding of the system and the ability to pivot quickly.

For instance, you may have read that first example and are now thinking "well, you could split the documents across multiple Memcached keys, or increase the block size, or move to Redis, etc…". All of those are potential solutions! But knowing which of those solutions will work - and more importantly, which of those solutions will not blow out the project timeline - is impossible unless you've got a deep understanding.

This is doubly important because the problem in question doesn't even need to be real. In the lead up to a project launch, it's very common for other teams or engineers to raise *potential* problems (e.g. "hey, are we sure the user data will fit in Memcached?") If nobody steps forward and explains why this isn't a problem (or if it is, how it's being addressed before launch), the project will be delayed, and it will be your responsibility. Why? Because your manager (or their manager) will not know whether this is a serious problem. That's what they pay you for! If you're not stepping up to address it, they will naturally err on the side of caution and *not ship*.

You need to stay light on your feet so that when these issues come up you're not neck-deep in other work. That usually means not being fully heads-down on implementation (i.e. delegating tasks to other engineers on the project). Ideally you should have at least 20% of your time free from implementation in the early stages of the project, scaling up to 90-100% in the final days. If you do that, when issues do come up you'll be able to grant them your full attention.

Can we ship right now?

Feature flags are the best way I've seen to do this, but staging environments also work, and so on. The key thing is to get whatever you're building in front of as many eyes as possible: yours, but also other engineers, and ideally leadership, product, design and so on. Five minutes playing with the actual feature, even in a very rough state, will bring up issues that nobody anticipated. Being able to directly see it themselves also does wonders for reassuring leadership that you've got things under control.

The best way to anticipate problems is to deploy early. In general, a helpful question to ask is **can I ship this right now?** Not this week, not today: right this second. If not, what would have to change for me to be able to ship *something*? If the ship needs a deploy, can that happen now behind a feature flag? If we're waiting on some other team to make a change on their end, can I make it so the system doesn't strictly require their change after all? For instance, if the platform team is setting up a cache layer, I could make it so my feature still works (albeit a little slower) if it can't find the cache.

Remember, your main priority is maintaining trust with your leadership team. Nothing builds trust like having fallback plans, because in case of emergency fallback plans indicate control over the situation. If the worst does happen and you can't release on the day, your manager will be much happier going to their manager about it if they can say something like "our options are to delay four days, or ship tomorrow by sacrificing X" - even if sacrificing X is a non-starter. That means they'll be more likely to interpret the delay as an unavoidable problem that you handled effectively, instead of as a mistake you made that means they can't rely on you.

I think a lot of engineers hold off on deploys essentially out of fear. If you want to ship, you need to do the exact opposite: you need to

deploy as much as you can as early as possible, and you need to do the scariest changes as early as you can possibly do them. Remember that you have the most end-to-end context on the project, which means **you should be the least scared of scary changes**. Everyone else is dealing with more unknowns and is going to be even less keen to pull the big lever. (If there's some other engineer who is across all of this who you're waiting for, bad news: they're probably the one actually shipping your project).

Summary

- Shipping is really hard and you have to make it your main priority
- Shipping doesn't mean deploying code, it means making your leadership team happy
- You need your leadership team to trust you in order to ship
- Most of the essential technical work is in anticipating problems and creating fallback plans
- Scale back your implementation work as you approach launch so you're free to jump on last-minute problems
- You should constantly ask yourself "can I ship right this second?"
- Have courage!

Glue work considered harmful

"Glue work" is an concept Tanya Reilly came up with in 2019. The idea is that there's a large amount of unglamorous work that every team needs in order to be efficient: updating the docs and roadmap, addressing technical debt, onboarding engineers, making sure people talk to their counterparts on other teams, noticing strands that are getting dropped, and so on. Practical, naive engineers gravitate to this work because it's obviously useful, but at promo or bonus time they're ignored in favor of the engineers who did more visible work (like delivering new features).

I think this concept is excellent. It's why I keep saying that shipping projects is so hard - if you're the kind of engineer who's used to just putting their head down and writing code, you won't have the tools to do the glue work that is actually needed to deliver anything successfully. Pure hackers don't ship. You need to be able to actually deal with the friction in a large organization in order to deliver value.

So why doesn't glue work get you promoted, if it's so crucial to shipping projects? Are companies stupid? Are they deliberately leaving value on the table? No, I don't think so. Companies don't reward glue work **because they don't want you prioritizing it**. And they don't want you prioritizing it because they want you shipping features. Glue work is hard. If you're capable of doing glue work well, they want you to use that ability to deliver projects instead of improving general efficiency.

The core problem is that you're deciding for yourself what the company needs instead of doing your job. Isn't it your job to make your team run smoother? No! **Your job is to execute the mission of your company's leadership.** It is better to execute that mission at 60% efficiency than to spend all your time increasing efficiency in general (or even worse, to execute some other mission at 100% efficiency).

Why? For two main reasons: first, you're inevitably going to burn out, which will be bad for everyone; second, it's better to let your team get used to operating at the base efficiency level of the company instead of artificially removing friction for a brief period.

Should you never do glue work? No, you should do glue work *tactically*. That is, you should do this kind of extra work for the projects you lead - the projects whose success you're accountable for - in order to make sure they succeed. You won't be rewarded for the glue work specifically, but you will be rewarded for the success of the project. For other projects, you should just do your regular job.

Is this a deeply cynical take about how to succeed in office politics? I don't actually think so. Large tech companies operate at something like 20-60% efficiency at any given time (as they get larger, they get less efficient). Even knowing that, growing is a deliberate choice: companies grow in order to capture more surface area, since even at a lower efficiency that's a way to produce much more value. If individual employees are willing to lift their local team to 80% or 90% efficiency by burning their time on glue work, companies will take that free value, but they don't have any real interest in locking that in for the long term (since it depends on exceptional people volunteering their time in hard-to-rewrad ways and thus isn't sustainable).

If you're one of those exceptional people, congratulations! You can use that power tactically to be a more effective engineer. But you shouldn't do it all the time.

Thinking like a strong engineer

So far I've written about how to think about your work from a business perspective (i.e. how to manage projects, what kind of work you should choose to do, and how to get aligned with your company's leadership). But of course you also have to be a strong *engineer*. What is a strong engineer?

Strong engineers

In my experience, the real measure of talent is not the speed or volume of output, but **the capability to do tasks that other engineers can't**. In other words, strong engineers can do things that weaker engineers just can't, even with all the time in the world. Therefore, the strongest engineers are stronger than people think they are: not 10x as strong as the median engineer, or even 100x, but infinity-x on some problems. The weakest engineers are weaker than people think they are: not 0.1x, but 0x. They can't do almost any of the tasks that need doing in a large software organization.

For example, there's a hard division between engineers who can ship complex projects and engineers who can't. It's not as if weaker engineers do it more slowly - they just can't seem to do it *at all*. Either a nearby strong engineer ghost-leads the project or the project fails. Some more examples of capabilities that are like this:

- Solving very difficult bugs (e.g. race conditions across multiple services)
- Delivering meaningful improvements to the thorniest parts of legacy codebases
- Successfully making changes that require a big architectural rework

For the top-end of the strongest engineers, the capabilities become things like "improving the SOTA for large language models" and "making self-driving cars work".

Not every strong engineer can do all of these tasks. Somebody might be great at solving difficult bugs but can't ship projects, or great at legacy codebases but can't move fast. However, if someone is great at one of these things, they're likely to be great at most of the others. I don't really know why: maybe it's just raw intelligence, or that being good at one thing helps you learn others, or that these capabilities are

more similar than they look, or that these types of engineers try really hard at everything. But in my experience it's definitely true.

I want to be clear that while being able to do an individual task is pretty black-and-white, the strong/regular/weak categories exist on a spectrum. It's possible to have a "regular engineer" who can excel at fixing hard bugs, or a "weak engineer" who is genuinely good at keeping their dev environment running. You can be on the fuzzy border between two categories.

Regular engineers

Right below strong engineers, you have the regular engineers who make up the bulk of most companies. Here's some examples of capabilities you'd expect these engineers to have:

- Solving 95% of bugs (e.g. normal, non-cursed bugs)
- Picking up and delivering most JIRA tickets
- Unsticking themselves from dev env issues most of the time

A long-ago colleague once referred to this type of engineer as a "plodder": they aren't particularly fast, but they'll make steady progress on a normal-difficulty engineering task. I now think "plodder" is an unnecessarily pejorative name for this, because the more experience I get the more I love these colleagues. They *help*. They *do the work*. They're just not burning with ambition to excel at the next promo cycle, or to blow their peers away with really impressive output. Probably they have other things going on in their lives!

I have very little to say about this group, except to warn against confusing it with the final group: weak engineers.

Weak engineers

The other category is truly weak engineers. These people have little to no capabilities at all. In other worse, the baseline difficulty of a normal-to-easy software task is above what they're comfortable with. I suppose a few of these people are overemployed or fraudulent in some similar way, but I think mostly it's a lack of ability. I want to be clear that I'm not exaggerating here: weak engineers *cannot complete almost any engineering task*. I've worked on teams without any, but if you're in the industry long enough you'll encounter them.

Ironically, while people like this exist at almost all seniority levels, you're more likely to encounter weak engineers in senior roles. I think this is probably for two reasons. First, the bar to hire juniors is explicitly capability-based, so it's harder to slip past it. In interviews, seniors can talk about work they were tangentially involved with, which is hard to distinguish from work they *did*. Second, a weak junior is often given more opportunity to learn, because it's socially acceptable for them to not know things. A weak senior has to conceal their lack of knowledge and learn in secret, which is much harder.

I don't have a lot to say about weak juniors. You should help them out, point them at challenging problems, and see if they can step up and learn. Weak seniors, however, are a lot more interesting.

How do weak senior engineers survive?

How do weak engineers survive at the senior+ level? They do a lot of one-way pairing. If you've ever paired with these engineers, it's a very unpleasant experience, since you have to do all the work, whether driving or navigating. Often the pairing is discreet - they'll quietly reach out to another engineer on the team in DMs for help on every single task they have. Sometimes there'll be one unlucky victim who gets their time used like this: for instance, an effective junior on the team

who's happy to help and too inexperienced to know better. More savvy weak engineers will round-robin their pairing across the team, so each individual member might only need to chip in every week or so. Only when everyone compares notes does it become clear that the weak engineer is pairing on 100% of their tasks.

Weak engineers are often surprisingly active in work-related discussions. This is partially because they've got a lot of time - unless they can find someone to "pair" with, they're not really working. It's also an effective defense mechanism. If questions come up about their individual output, they can gesture at their public communications as evidence that they're leveling up the team instead of grinding out concrete work. One way to tell a weak engineer in a discussion thread about some problem is to see who is bringing in specific facts about how the system currently works, and who is making purely general recommendations that could apply to any system. If their messages could all be public tweets, they're probably not adding much value.

Some tips on working with weak senior engineers

The first and most important thing is to remember that **it's just work**. Somebody being bad at their job does not make them a bad person. Do not be an asshole! People can be weak through laziness and lack of talent, but also for all kinds of personal reasons that are none of your business. Act with the generosity that you hope would be extended to you if you endured some personal tragedy that meant you couldn't focus on work. Even lack of talent can be context-specific: for instance, maybe they're an embedded-systems whiz trying out a different field and not having much success, or a big-company employee struggling to adapt to startup norms.

Still, you should try to protect your time. Don't quietly give up hours of your workday to helping them stay afloat. One tactic is to **avoid time-asymmetrical helping**. Don't do work for them that takes much more

time than it took them to ask about it. For instance, if you're asked "hey, I have this issue, how would you approach it", don't take the time to work out the actual solution and hand it to them. Fire off a *quick* response that points them at the next immediate step (e.g. "oh yeah, looks like something in the billing code, you should see how service X handles it"). This means they can't spend minutes of their day tying up hours of yours.

Relatedly, you should try to protect the time of the junior members of your team. Don't let them be exploited by weak seniors who will ask them to solve their problems and then frame it to management as them helping the juniors level up. The best way to do that is to (professionally) make sure your manager knows the situation. It can be a hellishly difficult process to manage people out at some companies (and there might be things going on that you're not aware of), so don't expect that this person will be fired/pipped/told to shape up. But you still have a responsibility to make sure your manager's aware.

Conclusion

Engineering talent isn't extra speed or output, it's the capability to do tasks that other engineers can't. That's why the weakest engineers output so little, and why tech CEOs obsess about hiring the strongest engineers.

Most engineers can do a broad set of normal job tasks, but some can do really hard ones and some can barely do any. You should probably try to expand the set of tasks you're comfortable doing. If you're working with a weak engineer, be nice but protect your time.

What makes strong engineers strong?

As I said above, what defines a strong engineer is the ability to do tasks that weaker engineers can't, even with near-unlimited time. But what are the concrete skills or traits that make up that ability? What is it about strong engineers that makes them able to do a much wider range of tasks? In order of importance, I think it's self-belief, pragmatism, speed, and technical ability.

Self-belief

Strong engineers believe they can succeed, even on hard or unfamiliar problems. Many technically capable people are weak engineers purely because they lack confidence. Why? Because **software problems are all unfamiliar problems**. Software engineers work in the dark. For almost every piece of real work, it's impossible to say how tricky it'll be to do until it's completed. That's why estimation in tech companies is famously hard: you just don't know ahead of time what issues are going to come up.

There are many smart people who just can't work in this environment. It violates their core engineering sense-of-self to talk about projects when they're not sure of the technical details, or to start work on a project where so much is still unknown. To be a strong engineer, you need the raw confidence to believe that you will figure it out, whatever it is (or if it's genuinely too hard, then the problem must be so fiendish that there's no shame in failing to solve it).

This doesn't just affect what projects engineers work on, but the quality of the work itself. If you pick up a hard task, it makes a big difference whether you immediately tackle the hardest part head-on, or if you put it off and try to work around it. I've seen engineers spend weeks avoiding a difficult task that could have been accomplished in a serious day of effort.

It also creates a positive feedback loop. Engineers with a lot of confidence tend to work on more hard problems, which builds up their confidence, which pushes them towards even harder problems and so on. One popular way to describe these kinds of engineers is "agentic". As the common phrase goes, "you can just do things".

Pragmatism

Strong engineers get things done. They bias towards working solutions. In my experience, strong engineers are all ruthless pragmatists: every design decision is judged by how well it will work, not how clean or elegant it is. That's not to say strong engineers don't produce elegant solutions. Elegant solutions are often the most straightforward ones. But strong engineers resist adding layers of abstraction just for neatness' sake. They're typically happy to make compromises in the interest of shipping.

Because of this, strong engineers often come into conflict with smart, technically capable, weak engineers. Typically the flashpoint is something like "do we need to do this refactor two weeks before launch", or "is this design pattern overkill for our use case". It can be hard to judge these conflicts as an outsider, because both parties are technically strong. If you're in this position, I recommend tiebreaking on which side of the argument has the best track record of shipping.

Speed

Strong engineers work fast. I have never seen a strong engineer who wasn't a speedy worker. I think there are two reasons for that. First, many engineers are slow because they put off hard work. Strong engineers don't do that (see the point above about self-belief). Second, fast workers tend to become strong engineers over time because they accumulate experience rapidly. I'd write more on this, but Dan Luu

already said it best here. I endorse that post wholeheartedly, in particular these points:

- Fast execution lets you experiment rapidly enough to find the right solution
- Fast execution makes low-chance-but-high-reward ideas worth trying, which eventually produces high reward
- Fast execution is qualitatively different from slow execution. Whole different tasks become possible

I don't think strong engineers need to be particularly hard workers, in the sense of working long hours. In my experience, good work looks like short intense bursts of productivity separated by long periods of relative inactivity. Of course, there are probably some absolute monsters out there who are both intensely productive and work long hours (reportedly John Carmack is one of these). But I don't think you need to do that to be a very strong engineer at your tech company.

Technical ability

There's certainly a baseline level of technical skill you need to be a strong engineer, and being more technically capable is always an advantage. Deeply technical people will see simple solutions that others miss, and will be able to solve whole categories of problem that others can't.

How technical does a strong engineer need to be? That's a question about how good a fit your specific technical strengths are for the work that needs doing. For instance, I am very good at tracing through a large Rails codebase. That's made me a good fit for debugging and building line-of-business features, but none of those strengths helped much on a recent piece of work when I had to add some functionality to an inhouse C library. In general, a company working on a brand-new engineering field will need more technical skill than a company

applying established engineering practice.

Strong engineers don't need to be geniuses. In fact, often genius runs counter to the skills you need to be a strong engineer. Some of the smartest people I've worked with - in terms of raw brainpower - were not particularly effective engineers, because they struggled with pragmatism and speed. I'd much rather work with an averagely-intelligent engineer who was unusually confident and pragmatic.

Summary

- Strong engineers believe in their ability to solve almost any problem. They tackle the scariest unknowns head-on instead of procrastinating
- Strong engineers are pragmatic, which often gets them into arguments with smart, weaker engineers
- Strong engineers are always fast workers, but not necessarily grinders
- Strong engineers are technically strong, but it's usually a case of their strengths being a good fit to the work, not "this person is a genius"

Value over replacement

There are two ways of assessing how much value you're providing as an engineer. The first way is to total up all of the code you've shipped and all of the value that code has provided (e.g. how much money it's made). The second way is to try and figure out what *you specifically* have done that a replacement-level engineer in your place wouldn't have done. In other words, you can look at your literal **value**, or you can look at your **value over replacement**.

I don't think one way is necessarily better than the other. But it's important to be clear-eyed about which method you're using and why. For instance, if you're wondering if you're justifying your compensation, you should be looking at **value**. That's a straightforward calculation of how much money your company is spending on you, and how much value it's getting back. But if you're wondering if you deserve to be promoted, you should be looking at **value over replacement**. It's cool that some code you shipped made 10M in ARR, but if that was normal work that any of your peers would have done (e.g. if you'd been sick), it doesn't necessarily mean you're performing above your level.

How can you measure value? It's easy to know how much money the company is spending on you. The hard part is gauging how much money the company is getting back. If you know how much profit the company is making - and you should - you can try to estimate how much profit your specific part of the company is making, and how much your code is contributing to that. For instance, if you write a signup flow for a new customer segment, and those customers end up being a ~5M/yr source of revenue, you can plausibly say that your value is some reasonable percentage of 5M/yr.

What about value over replacement? That requires guessing at which of the things you did were things other engineers wouldn't have done

in your place (or would have done slower). You have to develop a concept of a "replacement-level engineer": the median-strength engineer that could be hired at your level and at your company. It's easy to fool yourself in both directions here - either imagining that your replacement would be a total dud, or that your replacement would be very strong indeed. In general, this is much harder than just measuring value.

If you're in the habit of proactively identifying new work (e.g. hacking on new features or digging through metrics to find performance issues), estimating your value over replacement is much easier. The replacement-level engineer would have at least tried to do all the work you were assigned, but they wouldn't have had the same ideas you had. As long as the work you identify actually adds value, that value is value over replacement.

However, you can definitely add value over replacement doing regular work, even if it's harder to spot. One way to identify this is to notice when you (or another engineer) are explicitly requested for a particular project: because of your subject-matter expertise, or unusual speed, or something along those lines. This is a direct recognition of value over replacement.

In theory, you don't necessarily have to deliver value over replacement. Being a standard-level engineer across all metrics is good enough by definition. But nobody is standard across all metrics. Everyone's worse at some tasks and better at others. In practice, you had better deliver positive value over replacement in at least some areas, because you're likely delivering negative value in others.

Strong engineers are right a lot

Amazon infamously has a leadership principle where they say "good leaders are right, a lot". It's unclear to me how useful it is about leaders, but it's definitely true about engineers. Good engineers are definitely right, a lot.

Bryan Cantrill has a good rant about the Amazon principle here. I agree with Bryan that it's a bit silly-sounding, and that it's not the kind of fundamental principle that can guide tough decisions (no problem, just pick the right choice, like a good leader would do!) But I do think it's just a straight-up fact. Good leaders *are* right a lot. Of course that flows from other qualities: they're smart, or they surround themselves with good people, or for any of a hundred other reasons. I think the point of Amazon's principle is that it's hard to assess those other qualities in the aggregate. It's easier to ask yourself: is this person right a lot?

Okay, it's easier, but still not easy. You can see whether a leader *wins* without too much difficulty, based on whether users are flocking to their products, whether they're making scads of money, whether their companies are seen as amazing places to work, and so on. But people win for a lot of reasons. You can win by being wrong-but-lucky about one particular high-leverage thing. Bad leaders do sometimes luck into amazing products. The world of business is just too chaotic to assess accurately.

Not so in the world of engineering. Good engineers are right, a lot. They're right about concrete factual statements about how software systems work (e.g. endpoint X has rate limit Y). They're right about concrete plans (e.g. we can't add this check here, but it'll work there). They're right about a million small decisions that they make when they write code, which shows up in their code generally working and causing fewer bugs.

You can assess an engineer based on how smart they sound in conversation, whether they use light mode VSCode or dark mode vim, what their title is, and so on. But if you have the time, it's far more reliable to assess them based on whether they're right a lot. For technical colleagues, this assessment comes automatically: you can just ask yourself whether you feel default relaxed or suspicious when they make claims, or how much of a critical eye you naturally bring to review their PRs. Otherwise you can pay attention to what they say with confidence, and whether it turns out to be correct. If you're an engineer, there's a pretty good chance that your manager is quietly tracking this.

One caveat: some engineers avoid being wrong by never making confident technical statements. I think this is a dereliction of duty. If you're the most technical person in the room, it's your responsibility - your job - to give information and advice. Always saying "well, I'm not sure" or qualifying all your statements makes it very hard for less technical people to work with you. In any case, the principle is "be *right* a lot", not "never be wrong". You can't be right a lot if you don't put yourself out there.

To continue that point, there's a difference between "right, a lot" and "always right", or even "often right". In domains that are technically very unclear (e.g. rotted legacy code, or frontier technical problems), being right even some of the time is extremely valuable. In my experience, being wrong is forgivable, particularly if you're the only one stepping forward to offer an answer or a plan.

All of this isn't helpful advice for engineers who want to *become* right (a lot). That's a much more complicated topic. But it's a good metric for gauging your own performance:

1. Are you putting yourself forward and making technical statements (either in English or in code)?
2. Are those statements right, a lot?

Strong engineers take positions

Some engineers think it's a virtue to remain non-committal in technical discussions. Should our team build a new feature in an event-driven or synchronous way? Well, it depends: there are many strong technical reasons on each side, so it's better to keep an open mind and not come down on either side. This strategy is fine when you're a junior engineer, but at some point you'll be the person in the room with the most context (or technical skill, or institutional power). At that point, **you need to take a position**, whether you feel particularly confident or not.

If you don't, you're forcing people with less technical context than you to figure it out themselves. Often that means somebody will take a random guess. In the worst case, the weakest-but-loudest engineer on the team will take the opportunity to push for a spectacularly bad idea. If you're a strong engineer, it's your responsibility to take a position in order to prevent that from happening, even if you're only 55% or 60% confident.

Why remaining non-committal is cowardly

Like most forms of cowardice, remaining non-committal feels like sensible caution from the inside. After all, technical problems are complicated. There are always reasons to express uncertainty or to add caveats to a statement. If the right way to go really is unclear, then (they say) it's strictly correct to express uncertainty.

I think what's often motivating this attitude is that many engineers (me included) really, really, pathologically hate being wrong. I get a sick feeling in my chest when I'm wrong about something, particularly in public. I think about it afterwards for a long time. This is useful, because it makes me put in the effort to be right. But it also makes it emotionally difficult to give an educated guess in a meeting that might

end up being dead wrong. I've had to work to become OK with doing that, so I sympathize with people who can't. But I also see it for what it is: cowardice. When people are relying on you to make a call, you ought to step up and make it.

What if you're wrong?

When an engineer overuses caveats and qualifiers, managers do not typically think "wow, I'm glad this person is being so careful and accurate". They think "ugh, why are you forcing me to make the decision myself?"

In my experience, managers are very forgiving when you make a technical call and it ends up being incorrect. That's because their job involves making a lot of educated guesses as well, so they've internalized that some guesses don't pan out. This goes double when the call you're making is genuinely difficult - for instance, a technical problem comes up in a meeting and everyone falls silent. If you're the only one stepping forward to answer, that can still be valuable even if you're wrong. Going in the wrong direction will at least often give you information, or provide a base to iterate on.

Of course, if you're wrong too much, people won't trust your estimates anymore. Or if you're too wrong in any particular instance - for instance, you offer a solution to an incident which ends up causing a much worse incident - you'll lose credibility too. I suggest avoiding this by being right, a lot.

Sometimes avoiding commitment is smart

Estimates are an interesting example of this. Many engineers default to "well, it depends, hard to say, could be a few days or could take a month" for everything but the most obviously-one-line changes. But your manager isn't asking out of curiosity, they're asking because they

need a loose estimate for planning purposes. If you give a non-answer, they will just sigh internally and guess the estimate themselves.

However, sometimes avoid estimates isn't a matter of cowardice. In some companies, engineers avoid firm estimates because they'll face real, unfair consequences when those estimates aren't met. Here the trust between engineering and product has been fully broken. Engineers are incentivized to keep their heads down and never commit to anything (at least in front of management).

I'm sure there are company environments where every technical commitment is this risky. I don't have any criticism for engineers in those environments.

Summary

I want to finish by repeating a caveat of my own. I'm saying you should force yourself to make commitments *when you're the person in the room best positioned to know the answer*. When you're talking to a technical peer - e.g. another engineer on your team - with your level of context, you can be as non-committal as you like. Still:

- If you don't take a position, you're tacitly endorsing the decision that eventually gets made
- In the extreme, this forces your manager to make hard technical decisions that are *your responsibility*
- The harder the decision, the more uncertainty you should be willing to accept
- I'm only talking about functional environments. If your manager will PIP you for a missed estimate, that sucks - I don't have any criticism for people who stay silent in that situation
- It can be genuinely scary to make a claim that you're not sure about. But you should still do it

Strong engineers know about the spotlight

Think of a tech company as a giant, dimly-lit factory. Work goes on throughout the factory as components shuffle back and forth, and finished products get steadily carted away, but the coordination and efficiency of each section is fairly low. Some parts of the factory get jammed up and don't do anything for days. Other parts are manically producing broken components which are thrown away unused. However, the factory manager - i.e. the tech company CEO - has one tool to combat this problem. That tool is the spotlight.

There is a giant spotlight mounted on the rafters, like the bat-signal, and at any moment it is pointed at some part of the factory. When a part of the factory is lit up, it operates more in line with the factory manager's desires. Employees work harder and more efficiently. But the spotlight can only cover one section at a time. When a section is lit up, all other sections are in the dark. Because of this, clever use of the spotlight is the most important part of the plant manager's job. Keeping it pointed at the most crucial work (without letting any part of the factory sit too long in darkness) is a delicate balancing act.

Tech companies have limited focus

The spotlight represents **what the company is actively focusing on**. Tech companies can only focus on one or two things at a time, because "focus" means "the CEO is actively supervising this". Everything else muddles along. In functional companies, the spotlight makes the difference between highly-focused work and regular, still-successful work. In dysfunctional companies, the spotlight makes the difference between work that happens and work that simply does not happen at all (in other words, work *only* happens under the spotlight). Specifically, being under the spotlight means:

- The CEO is asking for daily updates, which imparts a real sense

of urgency down the org chart
- Blockers will be removed by force, if needed (e.g. if team Y is dragging their feet, expect their boss's boss's boss to come down and sort it out)
- Agile processes (estimations, planning meetings, retros) are often abbreviated or removed entirely

For software engineers, the spotlight is a time of high-pressure and high-reward. Your manager and skip-level manager will never pay you as much attention as when your team is under the spotlight. Projects you've done under the spotlight will be five to ten times more impactful in your promotion packets - assuming you've visibly done a good job on them. If you're never under the spotlight, it's hard to get the kind of exposure you need for promotions (especially to senior or staff roles).

It's a really good idea to optimize for working under the spotlight: for instance, make your baseline 80% or 90% effort, so that you can flex up to 110% or 120% for short periods when the spotlight is on you. This will help you show off, but it'll also help you avoid damaging mistakes. Screwing up under the spotlight can stain your reputation at a company permanently.

Chasing the spotlight

Some engineers spend their careers chasing the spotlight. This can be a virtuous cycle: if you distinguish yourself under the spotlight, you're likely to be moved to the next area that the spotlight points to. In effect, you can become **part of the spotlight itself**, by becoming one of the tools that your company leadership uses to focus on specific areas. When senior leadership thinks "we need to get this project right", they'll also think "we should transfer engineer X to it, they're reliable". For obvious reasons, this is a good way to get promoted.

However, chasing the spotlight is exhausting. Not only does it require

working harder, but the constant transferring and project-switching can be tough to adjust to in itself. You tend to work with a lot of different people instead of staying with a single team, which makes it more difficult to build the kind of long-term colleague relationships that make work much more pleasant. And frankly, as a senior+ engineer, you're being paid well but definitely not on the same level as the C-staff and SVPs whose priorities you're enabling - and whose bonuses you're securing! It's not necessarily a good deal to opt-in to the CEO schedule full-time.

Avoiding the spotlight

Other engineers spend their careers hiding from the spotlight. Sometimes this is because they're incompetent and don't want to be put under pressure, but plenty of skilled engineers just prefer to do good technical work as far away from executives as possible. These engineers typically get a lot of intrinsic motivation out of writing code itself (as opposed to delivering shareholder value). They're also typically not ambitious - either the positive ambition of wanting to push for more power and influence, or the negative ambition of being terrified of being sidelined or laid-off.

There's nothing wrong with this, but it does mean you're likely to be under-rewarded for your effort. Your hard work just isn't going to go as far. One month of grinding in the spotlight is worth one year of grinding in the dark. For good reason! Companies reward engineers who work to achieve the company's goals. If you're not working on anything in the company's top priorities, the company is going to (probably) reward you less for it. That might be a deal worth making, if you really do like working away from the spotlight - but you shouldn't then complain about the consequences.

Summary

- Tech companies can only focus on one or two things at a time; this is the “spotlight”
- Engineers who perform well under the spotlight get more visibility, rewards, and career growth
- Some engineers chase the spotlight and become known quantities to leadership, at the cost of stability
- Others avoid the spotlight and do good technical work quietly, but tend to be under-recognized
- Either strategy can work, but you should be honest about the tradeoffs

Strong engineers are trusted by their management chain

It's fun and rewarding to work on critical tasks. But there's only so much important work to go around. Worse still, the chances to work on these projects are unevenly distributed, because often this work is done by hand-picked teams. I've been on a fair few of those. What gets an engineer put on that list?

Circles of trust

As you acquire more recognition among managers at your company, you rise up through a series of concentric circles. At the center is being the go-to engineer on your own team. You start here by just being straightforwardly good at your job. At this point, your manager will (explicitly or implicitly) be steering important work your way.

Next up is being the go-to engineer in your org. At this point, your manager's peers are asking your manager to task you with key pieces of work. You'll probably be on the shortlist to lead any cross-team projects that touch your team, or at least you'll be tactically placed on cross-team projects that must succeed. Your org's leader (typically a director or VP) knows your name, and will occasionally reach out to you directly.

Finally, you're one of the inner circle of engineers at your company. All the engineering-facing directors and VPs know your name, and you're routinely pulled onto cross-org ad-hoc teams to execute on key bets. You're considered a key resource, so you're typically not allowed to sit on an individual project for very long - the company will shuffle you over to whatever it considers most important at any given time.

These concentric circles are circles of *trust*: first trusted by your own manager, then by your manager's peers and direct bosses, and finally

by the real movers-and-shakers in your company, all the way up to the CEO. Unless you're already snowboarding buddies with your VP, you must pass through those levels in order. Your boss's boss is only willing to consider trusting you because your boss does, and so on.

How do you move up through the circles? In my experience, by being invited each step of the way. Unlike in your own team, you can't go up and ask to be put on a particular piece of work. Instead, you'll get some contact with senior managers by working on important projects, which (if you do a good job) encourages those managers to tap you for more important projects, which gets you contact with more senior managers, and so on. You have very little direct control over any of it - all you can do is put yourself out there and try to be the kind of engineer that leadership wants to put on other projects.

So the question "how do I get the good opportunities" really boils down to this: **what does it take to be the kind of engineer the company can trust?**

Why trust is so hard

Very few engineers really appreciate just how delicate the relationship is between themselves and very senior managers. As you talk to people higher in the management chain, they have less and less technical context, and no direct power to affect the product. But they have a lot of institutional power to ask engineers to do things, and a lot of *responsibility*: it's inconvenient to you if a product you work on doesn't succeed, but it will cost your leadership team real money in lost bonuses (and maybe get them fired). So they're heavily invested in success, but almost entirely dependent on their engineers to actually do the work that needs doing, and to tell them what can be done in the time available.

That's why the engineer-leadership relationship is strange and delicate

to maintain. These are very powerful people at the company, who can mobilize hundreds or thousands of people to achieve their goals, but they *need your help* to do anything directly. It's an awkward position for them to be in. Even Caesar felt helpless when in the dentist's chair. If you can be genuinely helpful in these circumstances, you will reap the benefits of having Caesar as an ally. But of course, if you betray his trust, you may get your head cut off.

One reason large tech companies have many layers of managers is to avoid having to rely on this tricky relationship. VPs are supposed to be able to talk to their manager direct reports, who talk to their manager direct reports, who get estimates and plans out of their engineers. Everything thus gets safely triple-handled on the way up and down the chain. In practice, very senior leadership people often chafe at this indirection, and quietly rely on a few very senior engineers as a sounding board or a sanity check. Sometimes this is even formalized in the org chart (for instance, as a 'floating' staff engineer reporting directly to the VP or CEO).

Maintaining the trust

How can you keep the trust of very senior managers? First, be discreet. If your boss's boss's boss comes to you with a quiet question, and you immediately start saying in Slack things like "as I talked about with Big Boss the other day", you have demonstrated that you can't keep your mouth shut. Being trusted by senior leadership often means being trusted with confidential information (about company plans, for instance). Avoid giving the impression that you might not know to keep that private.

Second, talk at their level. You and your CTO share almost completely different contexts: different working spheres, different problems, different tools. When they come to you and ask how well a particular feature works (for instance), that question may as well be in a foreign

language if you don't have their specific context. If you immediately launch into a reply about SLOs or the quality of the code, you are going to bore them and provide no value. They are almost certainly coming to you because they have a specific agenda or they heard some claim that they want to double-check. You need to draw out some sense of what that agenda or claim is before you commit to a response.

Finally, you must get the technical questions right. If you tell them that something works a certain way and it doesn't, or that it's impossible to do X and it turns out to be possible, the trust will be gone. Very senior managers are distant enough from the work that they can't double-check you on details, but because of that they'll be paying close attention to the things they *can* check you on, like the customer-facing results of a project.

What happens if you do break the trust? Unlike Caesar, your friendly neighbourhood executive is not going to literally cut your head off if you mess up the interaction. But you will silently and immediately be dropped from that circle of trust. They will never ask you a direct question again, never loop you into early planning discussions about secret projects, and they won't mention your name in private discussions about who should lead what project. This is not the end of the world, or even the end of your ability to be promoted at the company. But it is the end of your close relationship with that particular executive.

Keep it in your management chain

A word of warning. It is great to be trusted by managers at your company. But **you must try to keep it inside your management chain**. I have seen engineers fall into the trap of becoming the go-to person for managers or product people from different orgs, at the expense of their actual job. There is no shortage of predatory managers outside your org who would love to have more engineer time "for free" (i.e.,

they don't have to bargain for it with people who they're actually accountable to). One "quick favor" can be a sensible idea. Two or three is a mistake.

It can be pleasant to feel helpful, and for many engineers the opportunity to solve someone's problem is tempting enough. But it's a really bad idea to get involved like this. These people will be very persuasive - it's their job! - but they won't be there during your next promo discussion, and they'll walk away from you the second it's in their interest to do so. Save your energy for the work your org actually pays you to do.

How do you push back against these requests? Directly involve your actual management chain. Say something like "hey, I'd love to help you, let me run that request by my manager so she knows what I'm doing". Your manager will probably bite their head off for poaching her engineers.

Summary

- The foundation of trust is being good at your job. Be good at your job!
- You get noticed by working on important projects, which gets you noticed by more senior folks, and so on
- The more senior a manager you're trying to build trust with, the more delicate the relationship will be
- It's a different set of skills than building trust with your direct manager
- Direct trust relationships are very useful but violate the "normal" chain of command, so must be kept relatively discreet
- Do not alienate your own direct manager
- Do not invest in these relationships outside your own management chain! Maybe it's possible, but I've never seen it work

How I decide what to work on

One of the most important career skills in tech is learning to recognize **what work actually matters**. Many engineers go through their careers without really making that decision. They might speak up for a particular issue in sprint planning now and then, but they don't maintain a mental list of the most important work going on in their team. It's easy to punt that decision to your manager. After all, it's not really your job to decide what projects are important—your job is to execute on projects and speak up for the technical side of things.

This is a big mistake! Most prioritized work is low-priority. At tech companies, the job isn't like assembly-line work where every hour is as important as any other. It's more like being a firefighter: long periods of pretty chill work punctuated by short intense periods where it's important to get it right.

Why important work is rare

The job's like that because tech companies can only focus on one or two things at a time. "Focus" means "the CEO is actively following up on this," and a single person can only follow up on a small number of projects. When the spotlight is on one of your team's projects, you had better be willing to drop less critical work and be fully engaged on that project. But the spotlight can only shine on one thing at once—eventually, it'll move on.

Engineers who have a good sense of what work is important will recognize the early signs of this process and make sure they're positioned to jump in. Engineers who don't will be knee-deep in other work when the spotlight hits, and thus won't be able to immediately move to the high-priority project. This is bad:

- For the engineer, because they lose out on the chance to do work that's visible at the highest levels of the company.

- For the team, because it means the team has fewer resources to spend on the important project.
- For the company, because it won't be able to execute as well on the project it's focused on.

Working on the wrong thing

In the worst case, engineers won't realize important work is going on and will visibly be doing other things during an all-hands-on-deck situation. For instance, they'll post on Slack with a question or an update about some unrelated task. **This is much worse than doing nothing.** Managers really hate it, because it advertises to everyone watching that they've failed to focus their team. It's just a really bad look.

Skip-level managers and above pay attention to which engineers were working (or at least appeared to be working) at times when the spotlight is on a particular team. But they also pay attention to engineers who make obvious mistakes during that time, or engineers who seem to be doing entirely different work. You can spend a year steadily building a reputation and lose it all in a week by looking useless. Or, more optimistically, you can make up for a slow year by absolutely killing it during the right week.

When the spotlight isn't on you

During the times when you aren't on a high-visibility project, I recommend carving out your own lab days or 20% time. (Maybe start with 10% time and work your way up.) This is a great way to rack up quick, bullet-point wins that can go on a promo packet or a resume. But it's also a way to practice putting some skin in the game. If you spend an afternoon on one of your own goals, and it doesn't provide value, that wasted time is *on you*. It's as if you spent the afternoon playing video games. Once you realize that, it forces you to become more agentic, which has all kinds of benefits for your career.

How can a project not provide value? For instance:

- Low-value refactors: unnoticed by engineers, only satisfying your sense of neatness.
- Feature demos that are never going to go anywhere (e.g. because it's not aligned with company strategy).
- Bugfixes that introduce new bugs, so they're net-even in terms of effort.
- Performance improvements on endpoints that don't care about performance (admin panels, low-frequency APIs).

These are examples of the kind of projects that don't provide value. But the *reason* a project doesn't provide value is that it doesn't (a) build you credit with managers, or (b) build you credit with your fellow engineers.

The value of lab days

One of the most useful things I ever did as a junior engineer was participate in Zendesk's weekly "lab days." This was our version of Google's "20% time" program—the idea was you'd spend one day a week working on something of your choice. The only rule was it had to be at least vaguely work-related: a demo of a new feature, or a piece of code cleanup that wasn't part of the sprint, and so on.

I don't remember what I shipped as part of my lab days. It was probably a series of minor performance improvements, since that's what I was excited about at the time. **But the real value was in regularly practicing the skill of choosing what I worked on.** And it is a skill: you start out very bad at it. It takes time to learn which tasks will have a solid impact and which won't.

Summary

- Choosing what to work on is a skill that most engineers don't practice
- Most prioritized work is actually low-priority, so the real impact comes from knowing when and where to focus
- Tech companies operate on a "spotlight" model where leadership can only focus on one or two things at a time
- When the spotlight is on your team, you need to be ready because high-impact work happens in short, critical periods
- Working on the wrong thing at the wrong time is worse than doing nothing because it signals a lack of awareness and can damage your reputation
- When the spotlight isn't on you, creating your own opportunities helps you stay engaged and build credibility
- Not all side projects provide value. Good projects either build credit with managers or build credit with fellow engineers

Don't be a JIRA ticket robot

Don't be a JIRA ticket zombie! I think a common experience among ambitious juniors - certainly I did this once - is to get frustrated at the slow pace of a team and decide "screw it, I'll just burn through all these tickets". On many teams, it's not that hard to do 2x or 3x more tickets than the next most productive person. But this is a dead end. You'll get a pat on the head, told "nice work, don't burn yourself out", and no progress towards a senior promotion.

Everything you could want from a company - promotions, bonuses, internal respect - comes from shipping *projects*, not from closing out tickets. It can still be useful to go through a stage of churning out tickets: to learn how fast you can work, to get rapid familiarity with the codebase, and to build some credit with your colleagues. But at some point you need to transition from doing whatever work is in front of you to prioritizing the work you think is the most important.

How do you work on important things?

What determines importance? Importance at a tech company is **whatever your directors/VPs/C-staff say it is**. I mean that quite literally. Your job as an engineer is to execute on the strategy of the company, and since that strategy is set by executives, your job is to make sure you're aligned with what they consider important. In any remotely functional company, they will tell you (over and over) what that is. You should pay attention when they do.

Try to be ruthless about dropping work that is no longer important. If you ship a project and your management chain begins talking about the next thing, *stop improving that project*. In my experience, continuing to work on an already-shipped project is a very common mistake. Declare victory and walk away!

The formula for prioritizing is literally this simple:

1. Am I working on the most important thing right now?
2. If not, drop what I'm doing and go do that

You should be asking yourself this at least twice a day. That's the first mistake I see lots of engineers make: not even really trying to prioritize, just coming in every day and continuing work on whatever they were doing the day before, or picking up a new JIRA ticket.

The second mistake is not having a sense of what the most important thing is. Here's how I figure that out in practice:

1. Is there an ongoing high-visibility incident going on (e.g. directors/VPs are asking about it)? If so, can I help with that?
2. Are there any open questions in Slack/JIRA/etc that I'm able to answer? Is there anyone waiting for a response from my team?
3. For each project that I'm leading, in order of importance: could it ship right now? If not, is there anything I could do right now to speed that up (any code that needs to be deployed or PRs that need reviewing)?
4. For each project that my team is working on, in order of importance: is there any work available to be picked up that would move it along?
5. Are there any tickets on the JIRA board in "ready for work"?

Note that going to the JIRA board is the *last* step, not the first step. Remember, companies care about projects, not tickets.

If you get this right - if you only spend your time on the most important thing - you can be hugely impactful with much less than 40 hours of work per week. Companies are so full of engineers who work without conscious direction that doing one or two useful things per day will immediately make you unusually productive.

It's not your manager's fault

Here's a story about the angriest I've ever been at work. Early in my career I was deeply invested in keeping a suite of staging integration tests green. It was really hard work: keeping them up-to-date as the app changed, making sure the stateful parts (e.g. the pool of accounts) were healthy, making the tests as resilient as possible to transient issues like slow connections, and so on. I sweated over it for *months*. We went from 90% red to near-100% green. And then in a meeting my manager made some passing comment suggesting I shouldn't spend so much time on it, since it wasn't that important.

I got up and left the meeting midway - the only time I've ever done that, before or since. I went to an empty meeting room, sat down, and updated my resume. I felt like the hard, important work I'd been doing had been completely dismissed. In hindsight, it's a funny story. I was working so hard! But my manager was right to dismiss the work I was doing, because whatever I was doing it wasn't my actual job.

I've seen this story play out many times. An engineer decides that some particular task is important (say, adding support for some obscure input format, or removing some piece of tech debt) and spends weeks or months working on this. They do all the correct Agile things: communicate early with their manager, create issues, break up their work into modular PRs. And then the first time this project conflicts with a *real* company value, they're told to drop it, which makes them furious (or sad) about all the wasted effort.

Is this a failure of management? Yes and no. In theory, their manager could have warned them early on that this was low-priority work and probably not worth their time. But managers often trust their senior engineers when those engineers tell them that technical work is important. Managers also tend to communicate gently about priorities: for instance, they'll give lukewarm responses like "sure, if you think this

is worth doing go ahead" instead of explicitly saying "I don't see the value in this work".

There's another reason why managers aren't reliable guides: the company's stated values are often at odds with their real values. When I was obsessing over integration tests as a junior, my manager wasn't about to tell me "actually, the company doesn't care *that* much if our integration tests are flaky", because the company was doing a lot of internal messaging about the importance of integration tests. It was my responsibility to decide how seriously to take that, and to weigh that against the other tasks I had available to me.

Be wary of convincing your company to do things

By far the easiest way to deliver value is to figure out what the most important thing your organization is doing and help with that. It can be difficult to figure out whether the company's stated tenth highest priority is actually valuable, but the company's stated top priority almost always is. You can usually just ask your manager (or your skip, if you have skip 1:1s) what their top priority is. Alternatively, pay attention to what your skip level and above are asking about in Slack or in meetings.

The hardest way to deliver value is to figure out an important thing that your organization *isn't* doing and try to convince them to do it. This is basically betting that your company's executives have done a bad job of identifying what they want, and that you can do a better one.

Maybe you can! Particularly if your company works on developer tools (like GitHub), you might have a useful perspective. But it's a big risk. Even if you succeed in convincing them it's worth looking at, you're unlikely to put it near the top of their priority list. The next time the wind changes, the project you worked so hard to push is probably going to stop being important. And then you'll be in my position with

the automated tests: deeply invested in a hard technical task that the company doesn't value.

Don't let me dissuade you from trying to sell the company on an idea. It's some of the most satisfying work you can do. Just be cautious if you do go down that road - if it doesn't work out, you might work away with zero return on the time you put in.

Summary

- Burning through JIRA tickets is a cool party trick, not a path to impact
- To work on important tasks, pay deliberate attention to what your management chain thinks is important
- Be ruthless about dropping unimportant work
- Don't rely on your direct manager to tell you what's worth doing
- Convincing your company that a task is important can work, but it's risky

Writing and reading code

The foundational skill of all of this is *being good at the actual job*: i.e. being proficient at writing and reading code, at designing systems, and so on. You cannot be trusted without being right a lot, and you cannot be right a lot without actually being technically capable. What does that look like?

Great software design looks underwhelming

Years ago I spent a lot of time reviewing coding challenges. The challenge itself was very straightforward - building a CLI tool that hit an API and allowed the user to page through and inspect the data. We allowed any language, so I saw all kinds of approaches. At one point I came across a challenge I thought was literally perfect. It was a single Python file (maybe thirty lines of code in total), written in a very workmanlike style: the simplest, most straightforward way to meet the challenge requirements.

When I sent it to another reviewer, suggesting that we use this as a reference point for what a 10/10 looked like, I was genuinely shocked to hear from them that they wouldn't have passed that challenge through to an interview. According to them, it didn't demonstrate enough understanding of sophisticated language features. It was *too* simple.

Years later, I'm even more convinced that I was right and that reviewer was wrong. **Great software design is supposed to be too simple.** I think now I can finally begin to articulate why.

Eliminating risk

Every software system has a lot of things that can go wrong. Sometimes these are called "failure modes" of the system. Here's a sample:

- SSL certificates expire and aren't renewed
- Database fills up and becomes too slow or out of memory
- User data gets overwritten or corrupted
- Users see a broken UI experience
- Core user flows (e.g. saving records) fail to work

There are two ways of designing around a potential failure mode. The first is to be reactive: adding rescue clauses around risky blocks of code, making sure failed API requests are retried, setting up graceful

degradation so errors don't blow up the whole experience, adding logging and metrics so bugs can be easily identified, and so on. This is worth doing. In fact, I believe this kind of (frankly paranoid) attitude is the mark of an experienced software engineer. But working like this is not a mark of good design. It's often a signal that you're papering over flaws in a bad design.

The second way to handle potential failure modes is to **design them out of existence**. What does that mean in practice?

Protecting the hot paths

Sometimes it means **moving components out of the hot path**. I once worked on a catalog endpoint that (due to other design choices) was extremely inefficient, in the order of ~200ms per record. This exposed us to a few nasty failure modes: resource starvation for the rest of the app, proxy timeouts on index requests, and users just giving up after waiting ten seconds for a response. We ended up moving the endpoint construction code into a cron job, sticking the results in blob storage, and having the catalog endpoint serve the blob. We still had the nasty 200ms-per-record code, but it was now under our control: it couldn't be triggered by user actions, and if it failed the worst-case scenario is we'd just serve a stale blob.

Removing components

Sometimes it means **using fewer components altogether**. Another service I worked on was a documentation CRM that had a really bespoke system for pulling various bits of docs out of different repositories and stitching them together into database entries (sometimes pulling docs directly out of code comments). This was originally a good decision - at the time, it was hard to get teams to write any kind of docs at all, so the system had to be maximally flexible. But as the company grew, it was very much showing its age. The sync job

stored some state in the database and some state on disk, and often triggered strange git errors when the state on disk got out of sync or the underlying host ran out of memory. We ended up removing the database entirely, shifting all the docs into a central repository, and reworking the documentation page as a normal static site. All kinds of possible runtime and operational bugs were removed, just like that.

Centralizing state

Sometimes it means **normalizing your state**. One of the worst kinds of failure mode are bugs that leave your state (e.g. your database rows) in an inconsistent or corrupted state: one table says one thing, but another table says differently. This is bad because fixing the bug is only the start of the work. You have to go in and repair all the damaged records, which can involve some detective work to figure out what the right value ought to be (or in the worst case, guessing). Designing so that the crucial parts of your state have a single source of truth is often worth taking on a lot of other pain.

Using robust systems

Sometimes it means **relying on battle-tested systems**. My favourite example for this is the Ruby webserver Unicorn. It's the most straightforward, unsophisticated way you could possibly build a webserver on top of Linux. First, you take a server process that listens on a socket and handles one request at a time. Handling one request at a time won't scale: incoming requests will queue up on the socket faster than the server can clear them. So what do you do? You fork that server process a bunch. Because of the way fork works, each child process is already listening on the original socket, so standard Linux socket logic handles spreading requests evenly between your server processes. If anything goes wrong, you can kill the child process and instantly fork off another one.

Some people think it's a bit silly to like Unicorn so much because it's obviously less scalable than a threaded server. But I love it for two reasons. First, because it hands off so much work to the process and socket Linux primitives. That's smart because they're ultra-reliable. Second, because it's really, really hard for a Unicorn worker to do anything nasty to another Unicorn worker. Process isolation is a lot more reliable than thread isolation. That's why Unicorn is the chosen webserver for most big Rails companies: Shopify, GitHub, Zendesk, and so on. Great software design doesn't mean that your software is ultra-performant. It means that it's a good fit for the task.

Summary

Great software design looks simple because it eliminates as many failure modes as possible during the design stage. The best way to eliminate a failure mode is to *not* do something exciting (or if you can, not do anything at all).

Not all failure modes are created equal. You want to try hardest to eliminate the really scary ones (like data inconsistency), even if it means making slightly clunky choices elsewhere.

These are all relatively boring, unsexy ideas. But great software design is boring and unsexy. It's easy to get excited about big ideas like CQRS or microservices or service meshes. Great software design doesn't look like big exciting ideas. Most of the time it doesn't look like anything at all.

Designing software in your head

Whenever anyone describes a piece of software to me, I think about how I would build it. Software engineers do this a lot, but many of them don't do it very well. I know that because I see a lot of technical discussions about specific details in a general plan *that could not possibly work*. For instance, arguing about whether to use prop-drilling or context-passing to supply a piece of data to the frontend that we do not and will never have access to, or the exact persistent-data-storage strategy to implement in a backend service that must remain stateless.

To avoid this, I think it's a good idea to **trace one important user flow end-to-end in your head**. What does that mean in practice?

Two common anti-patterns

The first mistake I see a lot is staying too high-level. Suppose you're building a comments system for a blog. Some engineers will stop at "oh, I'd put the comments in a relational database somewhere and pull them out to put on the page". A relational database might end up being the right choice, but this level of design isn't very useful for actually building the feature. You need to go one level deeper: how are the comments traveling from the user's browser to the relational database?

The second mistake is getting too invested in the wrong specifics. Some engineers will begin designing their commenting system by saying "oh cool, I'll use React", and then diving into a million micro-decisions about whether to use RSC or not, or whether to fetch the data via fetch or TSQ, or to expose the comment data in GraphQL, and so on. The first time you hear about the problem is the wrong time to make decisions like these. You may have to make them eventually, but not at the outset.

How to do it right

So what is the right way to do it? The right way is to take the **most important user flow** and trace the simplest possible implementation **all the way through** in your head. You can track only one user flow because otherwise you'll get confused. You have to trace the implementation all the way through because otherwise you'll miss key details. When I say "trace", I mean at the level of pseudocode. You don't have to imagine the entire code in your head, but you do have to imagine each logical step.

This is the mental equivalent of the Pragmatic Programmer's well-known "tracer bullet" rule. The tracer bullet rule is that your first prototype should be the minimum you need to build to get one user flow working end-to-end. The same is true for simply thinking about writing software: your goal should be to think one user flow through end-to-end.

The benefit of thinking through the flow at that level of detail is that you're forced to confront the important questions (just as you would be if you were building a prototype with real code). You don't have to design the cleanest or the best solution here, but you do need to design *something that could possibly work*. If you start with something that works, you can usually iterate to something *good* that works. If you start with something that doesn't work, it's much harder to iterate your way back into the space of working solutions.

Walking through an example

For instance, in the case of implementing a commenting system:

A user lands on one of my blog posts and should see a enter-your-comment form. That's easily done by adding a `<form>` element to my post template.

When they submit the form, their comment should be stored somewhere. Okay, so I need an endpoint on my backend and some kind of data storage. My add-a-comment endpoint code will be something like this:

```
comment = params['comment_body']
post = params['post_id']
user = ???

Comment.new(comment: comment, post: post, user: user).save!

redirect_to(post)
```

How can I set the user for the comment? If people are commenting anonymously, the solution is simple (add an optional form field for `name`), but otherwise I need to support some kind of login on my previously-static site. This endpoint can be pretty slow. Users will submit comments much less often than they view pages, and it's no big deal if it takes a second.

After the redirect, the user should then be able to view their comment. That means that I need some logic on the post page like this:

```
comments = Comment.where(post: current_post)
render(post, comments: comments)
```

And then some HTML templating on the post page that renders each comment. This endpoint must be very fast. Viewing posts is the main user activity on the site, and adding a few hundred ms of latency will meaningfully impact the experience. That points to the potential for caching or deferred loading, and the necessity for pagination once the number of comments grows.

Even with a simple example like this, you can see how it turns up what infrastructure pieces I'm missing (the ability to run code on the backend, data storage), and what questions I need to answer (how

can users log in or set their identity). When you do this for a system in a large tech company, you often turn up interesting questions as well:

- Our system needs data X, but it's only available from a slow endpoint in service Y
- Our system needs data that we don't currently collect (e.g. my static blog doesn't collect data about user identity)
- We need to display new comments on each post, but the posts are currently long-term cached on a CDN, so we'll need to find a way to bust that cache for each new comment
- We need to account for one or more wicked features - for instance, if someone's running the on-premise version of the blog, we'll need to figure out where we can store comments (or hide/disable the comment form)

Note that these points are all largely agnostic about what specific technologies are chosen (Rails or Express for the backend, MySQL or MongoDB for the data storage, etc). The assumptions they make are general ones that flow directly from the requirements: no matter what, comments will have to be stored *somewhere* and associated with users *somehow*.

How to communicate about a mental plan

The plan you make should stay mostly in your mind. In my experience, you will not be able to usefully explain it to product managers or even other engineers. The value of the plan is in how it helps you **estimate** and **ask questions**. For instance, the hardest part about adding comments to my blog would be switching to an infrastructure that isn't entirely static (and thus allows me to store data and run my own code in user requests). Estimating that part of work would give a rough guide for the entire project, and the questions involved (e.g. what platform should I switch my blog to, or should I use a third-party-hosted comments service) will be the most important questions involved in

planning the project.

Once the initial conversation is over, it can be very useful to write your plan down. I like a loose boxes-and-lines structure, usually in a Mermaid diagram, but it doesn't really matter how you do this. A short paragraph of text is probably good enough. A written version of a plan can be a good starting point for getting into more concrete implementation details. If everyone on the team agrees that the comments should be managed by a stateful backend app, then we can start talking about what technologies we should use and how.

I think this approach still works if you have a more explicit design process on your team (e.g. a collaborative design meeting, or some kind of architect-driven thing). You're much more likely to be successful at those processes if you go in with a concrete idea about how the feature could work. One caveat: **don't get too attached to that idea**. You should remain open to drastically changing the plan, as long as it's to something else that could also work. The first rough idea you came up with in your head is unlikely to be the best option overall.

Summary

This is the kind of point that feels almost too obvious to write down - when you're planning work, of course you should think about how the system could possibly function. But I think many engineers underestimate the difficulty of getting anything to work at all, and so handwave away the concrete details that they should be paying the most attention to.

If you instead jump into those concrete details immediately - what data is available, where it needs to get to, and how it's going to get there - you're likely to waste much less time arguing about implementation details of a strategy that never could have worked in the first place.

Working in large established codebases

Working in large established codebases is the most important skill at big tech companies, and it's one of the hardest things to learn as a software engineer. You can't practice it beforehand (very few open source projects are large enough to give you the same experience). Personal projects can never teach you how to do it, because they're necessarily small and from-scratch. For the record, when I say "large established codebases", I mean:

- Single-digit million lines of code (~5M, let's say)
- Somewhere between 100 and 1000 engineers working on the same codebase
- The first working version of the codebase is at least ten years old

I've now spent a decade working in these codebases. Here's what I wish I'd known at the start.

The cardinal mistake is inconsistency

There's one mistake I see more often than anything else, and it's absolutely deadly: ignoring the rest of the codebase and just implementing your feature in the most sensible way. In other words, limiting your touch points with the existing codebase in order to keep your nice clean code uncontaminated by legacy junk. For engineers that have mainly worked on small codebases, this is very hard to resist. But you must resist it! In fact, you must sink as deeply into the legacy codebase as possible, **in order to maintain consistency.**

Why is consistency so important in large codebases? Because it protects you from nasty surprises, it slows down the codebase's progression into a mess, and it allows you to take advantage of future improvements.

Suppose you're building an API endpoint for a particular type of user. You could put some "return 403 if current user isn't of that type" logic in your endpoint. But you should first go and look to see what other API endpoints in the codebase do for auth. If they use some specific set of helpers, you should also use that helper (even if it's ugly, hard to integrate with, or seems like overkill for your use case). **You must resist the urge to make your little corner of the codebase nicer than the rest of it.**

The main reason to do this is because large codebases have a lot of landmines in them. For instance, you might not know that the codebase has a concept of "bots", which are like users but not quite, and require special treatment for auth. You might not know that the internal support tooling in the codebase allows an engineer to sometimes authenticate on behalf of a user, which requires special treatment for auth. There's definitely another hundred things you might not know. Existing functionality represents a safe path through the minefield. If you do your auth like other API endpoints that have stuck around for a long time, you can follow that path without having to know all the surprising things that the codebase does.

On top of that, lack of consistency is the primary long-term killer of large codebases, because it makes it impossible to make any general improvements. Sticking with our auth example, if you want to ever introduce a new type of user, a consistent codebase lets you update the existing set of auth helpers to accommodate that. In an inconsistent codebase, where some API endpoints are doing things differently, you'll have to go and update and test every one of those implementations. In practice, that means that the general change does not happen, or that the hardest 5% of endpoints to update are just left out of scope - which in turn decreases consistency further, because now you have a user type that works for most-but-not-all API endpoints.

So when you sit down to implement anything in a large codebase, you

should always first go and look around for prior art, and follow that if at all possible.

Is anything else important?

Consistency is the most important thing. Let me quickly run through some other concerns as well, though:

You need to develop a good sense of how the service is used in practice (i.e. by users). Which endpoints are hit the most often? Which endpoints are the most crucial (i.e. are used by paying customers and cannot gracefully degrade)? What latency guarantees must the service obey, and what code gets run in the hot paths? One common large-codebase mistake is to make a "tiny tweak" that is unexpectedly in the hot path for a crucial flow, and thus causes a big problem.

You can't rely on your ability to test the code in development like you can in a small project. Any large project accumulates state over time (for instance, how many kinds of user do you think GMail supports?) At a certain point, you can't test every combination of states, even with automation. Instead, you have to test the crucial paths, code defensively, and rely on slow rollouts and monitoring to catch problems.

Be very, very reluctant to introduce new dependencies. In large codebases, code often lives forever. Dependencies introduce an ongoing cost in security vulnerabilities and package updates that will almost certainly outlive your tenure at the company. If you have to, make sure you pick dependencies that are widely-used and reliable, or that are easy to fork if needed.

For related reasons, if you ever get the chance to remove code, take it with both hands. This is some of the riskiest work in large codebases, so don't half-ass it: first instrument the code to identify callers in production and drive them down to zero, so you can be absolutely certain it's safe to remove. But it's still worth doing. There are few things in a

large codebase more worthwhile than safely removing code.

Work in small PRs and front-load the changes that affect other teams' code. This one is important in small projects too, but it's critical in large ones. That's because you'll often be relying on the domain experts in other teams to anticipate things you've missed (since large projects are just too complex to anticipate it all yourself). If you keep your changes to risky areas small and easy-to-read, those domain experts have a much better chance of noticing problems and saving you from an incident.

Why bother?

Finally, I want to take a second to defend these codebases in general. One common take I've heard goes like this:

> Why would you ever decide to work in the legacy mess? Spending time knee-deep in spaghetti code might be hard, but it isn't good engineering. When faced with a large established codebase, our job should be to shrink it by splitting out small elegant services instead of wading in and making the mess larger.

I think this is totally wrong-headed. The main reason is that, as a general rule, **large established codebases produce 90% of the value**. In any big tech company, the majority of the revenue-producing activity (i.e. the work that actually pays your engineering salary) comes from a large established codebase. If you work at a big tech company and don't think this is true, maybe you're right, but I'll only take that opinion seriously if you're deeply familiar with the large established codebase you think isn't providing value. I've seen multiple cases where a small elegant service powers some core feature of a high-revenue product, but all the actual productizing code (settings, user management, billing, enterprise reporting, etc) still lives in the large

established codebase.

So you should know how to work in the "legacy mess" because that's what your company actually does. Good engineering or not, *it's your job*.

The other reason is that **you cannot split up a large established codebase without first understanding it**. I have seen large codebases successfully split up, but I have never seen that done by a team that wasn't already fluent at shipping features inside the large codebase. You simply cannot redesign any non-trivial project (i.e. a project that makes real money) from first-principles. There are too many accidental details that support tens of millions of dollars of revenue.

Summary

- Large codebases are worth working in because they usually pay your salary
- By far the most important thing is consistency
- Never start a feature without first researching prior art in the codebase
- If you don't follow existing patterns, you better have a very good reason for it
- Understand the production footprint of the codebase
- Don't expect to be able to test every case - instead, rely on monitoring
- Remove code any chance you get, but be very careful about it
- Make it as easy as possible for domain experts to catch your mistakes

Working around wicked features

Why is working at large tech companies so hard?

It's because a small subset of "wicked features" dominate everything else. If you're building a todo app, adding the ability to attach images to todo items might be a large feature, but it's not a wicked feature. However, offering your todo app as a webapp and a standalone executable is a wicked feature. What's the difference? Wicked features are features that must be considered *every time you build any other feature*.

Here's some examples of wicked features:

- Adding a new user type
- Adding an on-premise version of your SaaS
- Sharding your customers across many different databases
- Supporting strong data locality
- Supporting the ability for customers to move their accounts between regions
- I18n (translating your customer-facing text into their native language)

Let's say you've done all these, and now you're building the image-attachment feature. Can the new user type add images? Say you're storing images in S3 normally - where are images being stored on-premises, where S3 isn't available? If customer data is sharded, are you sharding your `images` table appropriately as well? Are you making sure that you have a S3 bucket for each user region? If your customer moves their data between regions, do you have an automatic system for shifting the S3 images along with it? Have you pulled out all the new strings involved in image attachment, and have you budgeted the time it'll take to get them translated?

Why wicked features are hard

Wicked features are like the Password Game. In the Password Game, new rules - e.g. “your password must contain its own length as a number”, or “all numbers in your password must sum to 200” - cannot be considered and solved in isolation. They must be solved as a group, because changing the solution to accommodate one rule will often break several others. In fact, the Password Game is very generous by telling you immediately which rules are currently broken and why. In large tech projects, you’ll find out from user tickets or incidents.

This is a common reason for engineers underestimating tasks. It’s easy to forget one or more wicked features that complicate the implementation, and then to get blindsided when someone asks “what about X?” This is particularly true of engineers who haven’t spent much time at the company and might just straight-up not know about some of the wicked features. Company “veterans” are valuable largely because they’re familiar with all the wicked features.

Are wicked features a skill issue?

Are wicked features just bad design? Couldn’t you factor your program better to satisfy the requirements without adding a wicked feature? Sure, sometimes. I’m sure you could make any feature wicked with a sufficiently-clumsy implementation. But I think some requirements are inherently wicked.

Take “make this SaaS runnable on-premises”. It doesn’t matter how careful you are. Even if you make sure your SaaS build pipeline is completely on-prem friendly so you never have to maintain two versions, the fact that you have to be careful to do that is itself a wicked feature that you have to keep in mind for all future changes to the build pipeline.

Or take “add a new user type that can do X but not Y”. Suppose you

do a great job refactoring out user abilities so you never have to do `isUserTypeX(user)`. The fact that new capabilities have to fit your homemade user ability framework is itself a wicked feature.

What's wicked about these features isn't the implementation, but the fundamental domain model. It's wicked at the level of the user-flow diagram. No matter how well-factored your code is, you must still answer the questions I listed above (e.g. "can every user type access this new capability I'm building?")

Why build wicked features?

If companies could avoid building wicked features, they would. The problem is that the highest-paying users *love* wicked features. On-premise SaaS offerings are typically extremely profitable, since they appeal to a segment of users that are doing very well financially and who are comfortable paying high enterprise software contract prices (instead of low SaaS subscription prices). Likewise, data locality and sharding also appeal to enterprise customers with deep pockets.

Other wicked features are built by lazy or incompetent developers. For instance, companies with five users who nevertheless have built out a full sharding system for their data because it sounded like fun to the engineer in question. I've also seen engineers try to build wicked features because they couldn't see another way of doing things (or just felt that it was the "right" way to go about it). Extracting all user-facing strings for an app that only supports one language is a classic low-stakes example.

One of the most valuable things you can do as an engineer is to prevent your team from building wicked features where possible, and to limit the damage where the wicked feature must be built: by sensible factoring with an eye to "how will this affect developers trying to build completely unrelated features in the same system?"

Summary

- Wicked features are requirements that must be considered every time you build anything else
- They massively increase implementation complexity and coordination cost
- Some wicked features are unavoidable, especially when selling to high-paying enterprise users
- Others are self-inflicted by overengineering, dogma, or poor taste
- Good engineers limit the blast radius: they prevent unnecessary wicked features, and factor the necessary ones so they don't pollute everything

Playing politics

Suppose you're a strong engineer and your company trusts you to use that power in support of the company's goals. You're sitting pretty: promotions, bonuses, and high-profile projects should all be flowing your way. But with that increased visibility will come a host of new problems - political problems.

Protecting your time from predators

If you're a competent software engineer at a large tech company, your time is in very high demand. Lots of people will want you to do things. You should be very selective about how you handle these requests, and definitely avoid saying yes to everyone.

Helping other people feels good. That's doubly true when those people are from other parts of the company. It feels like you're having the kind of cross-org impact that a staff+ engineer ought to be having. But **helping other parts of the org is not your main job**. Delivering projects is your main job. It's a common trap to spend your time too generously and neglect the projects that are your actual responsibility. To avoid it, you should identify the people who are trying to claim your time.

Identifying predators

Large tech companies are full of predators: people who want to extract uncompensated work from competent engineers who are generous with their time. Once a predator identifies a good target, they will routinely send work to that person via DMs instead of going through normal channels. "Uncompensated" is a key word here. When your manager asks you to do work, that isn't predatory, because you're being paid for it and (hopefully) rewarded for it at review time. When a colleague asks you to do work, that isn't predatory, because they're in a position to do you a favor as well. Predators are asking you to do work that gives them a lot of value but doesn't do anything for you (or is even harmful). Let's look at some examples.

One common type of predator is a product manager from a different part of the org. I want to be clear that most product managers don't do this. But in every large org there will be a couple of this type among the product folks, because they're typically results-oriented people

and they're used to squeezing work out of engineers. Requests from this kind of predator look like this in Slack DMs:

- "I know you don't technically own this part of the codebase, but could you make a tiny change for me real quick?"
- "I got this (unrelated to your work) technical question from another PM, what's the answer?"
- "You're so great at using our data tooling, can you pull some statistics for me when you get a sec?"
- "My team is doing this project that's related to your part of the codebase, could you make a change here to unblock them?"

This is great for the product manager, because they get unblocked quickly and they don't have to spend any political capital on getting their work prioritized. It's bad for the engineer, because they neglect their actual projects and alienate their direct manager (who will resent their engineers being approached directly). That PM from another org won't be there in your review meetings, and their gratitude will likely be short-lived once you stop providing free work.

There's another common category of predator: the weak engineer looking for you to do their work for them. Requests from them tend to look like an endless stream of this:

- "I just picked up this ticket and I don't know where to start"
- "Help, my tests are red [PR link, with no further context]"
- "Can we pair on this ticket I'm working on?" [the pairing is entirely you driving and them watching]

In the extreme case, the weak engineer is doing effectively no work on their own, but drawing on a series of strong engineer friends to complete every task. To each strong engineer, it looks like they're just helping out a bit. This is a bad deal for the strong engineer, because the weak engineer (a) won't want to publicize just how much they're relying on the strong engineer, so won't share credit, and (b) won't be

able to help the strong engineer out with work later.

To be clear, I'm not talking about a junior engineer who's learning here - they'll be able to help you out later when they acquire the relevant skills. I'm talking about an engineer who's drawing on help from others as an alternative to actually learning the job. You should be very generous with your time to engineers who are learning, and very guarded with it around others.

Not all requests for help are predatory

Not all requests for help are predatory. It's part of your job to help out engineers on your team, and cross-org impact really does involve helping others sometimes, even if you get nothing in return. Predatory behavior is a *consistent pattern* of drawing on your time for nothing in return. One warning sign is when the request is itself very low-effort, but is asking you to go to a great deal of trouble. Another way to tell is that requests from predators *always* come through backchannels (e.g. Slack DMs). Asking in public would (a) make it clear what they're doing, (b) set you up to at least receive public credit for helping out, and (c) give your manager a chance to step in and protect your time.

I want to doubly emphasize that, by definition, **requests from your manager or from anyone in your management chain are not predatory**. These people will be involved in your review meetings, and are thus in a position to directly reward you for your work. Helping them out is directly and explicitly your job, and you should prioritize it over everything except the most crucial ships.

Avoiding time-asymmetric requests

One effective rule of thumb to avoid becoming prey is to watch out for asymmetric DoS attacks on your time. Distrust questions that are much easier to ask than to answer. In other words, be suspicious

when someone is putting in a small amount of effort to ask a question that would require you to do a lot of work. “I just picked up this ticket and I’m lost” is a very low-effort question that needs you to put in a lot of effort to answer it. “I just picked up this ticket and I’ve tried this (here’s my draft PR) but I don’t like it for these reasons” is a high-effort question.

When someone brings you a low-effort question, give a low-effort answer. That doesn’t mean a wrong answer or a rude answer - just don’t go and do a ton of work. Sometimes that means responding “sorry, I’m not sure off the top of my head”, or “yeah, I think we have data on that somewhere but I can’t remember where”, instead of actually going to look it up. Sometimes it means being vague, like responding “yeah I think you’ll need to update the auth logic in the controller” instead of finding and linking specific files.

This has a few benefits. First, it discourages predators who are looking for an easy target. Second, it immediately frees up your time. Third, it encourages people who you work with to ask higher-effort questions, which is good for everybody.

Summary

- If you’re a strong software engineer, your time will be constantly under attack
- Helping others is not your main job, doing the work is
- Large tech companies are full of predators looking for strong engineers whose time they can use
- Beware product managers from different orgs and weak engineers
- Predators aren’t in a position to reward you (i.e. they’re not in your management chain)
- Don’t let your time be consumed asymmetrically

Working with ruthless managers

There are two kinds of engineering manager: empathetic and ruthless. I think ruthless managers are underrated for a few reasons.

Empathetic managers care. They are emotionally invested in their employees as human beings, and actively campaign to support their employees' needs. Ruthless managers are there to do their job. They aren't *necessarily* assholes, but they see their main role as communicating the company's needs to their engineers and vice versa. They will almost never go out on a limb on an employee's behalf.

Overall, having an empathetic manager is good. A manager has a reasonable amount of power over their engineers - though less than many engineers think - and it's nice for that power to be wielded with forbearance and kindness. But I don't actually mind ruthless managers.

First, **ruthless managers still want their engineers to be happy**, if both the manager and engineer are competent. All things being equal, happy engineers work better and are easier to manage. A purely self-interested manager will do things to make their engineers happy, so long as they're not paying a substantial personal cost for it. For the same reason, ruthless *companies* want their (competent) engineers to be happy. It takes time for engineers to become useful at large tech companies. If you keep engineers happy, they'll stay longer and produce more effectively. So "ruthless" doesn't always mean "cruel".

Second, **empathetic managers are often in conflict with their own bosses**, leaving them with little political capital. Large companies make managers do a lot of the dirty work around performance management, firing, imposing tight deadlines, and so on. Empathetic managers are constantly pushing back on this. That can be good if the pushback is effective, but it often isn't. If your manager is always fighting with their own manager, they may not have the political capital left to push for the few things you really need (e.g. promotions). However,

a ruthless manager will usually have a lot of political capital banked with their bosses. It'll be harder to convince them to push for what you need, but if you succeed, they're very likely to be able to make it happen.

Third, for the same reasons, **empathetic managers are often unhappy**. Suppose you have an engineer on your team who really loves writing Python, but it's a dictate from upper management that they have to start writing in C#. Your manager will be the one who has to either pressure that engineer to write C#, or to argue with their boss about how the policy is foolish. A ruthless manager will (a) not agonize over the decision, and (b) pressure the engineer without feeling bad about it. If you're not that engineer, this is better for you, because your manager won't be sad and distracted.

Fourth, **ruthless managers are often better at communication**. Empathetic managers don't like breaking bad news and may cushion it too much. In my experience, they can also sometimes drink the company kool-aid, and give you the party-line answer instead of a more useful (or cynical) truth. However, ruthless managers can fail at communication for different reasons: typically because they're happier to deliberately lie to you, or because they're unwilling to be fully honest in case spilling the truth comes back to bite them.

Fifth, **ruthless managers are easier to predict**. Ruthless managers always do what their own managers - and the company in general - values. If company executives set a priority, ruthless managers will always follow it. Empathetic managers have their own priorities, so it's harder to know what they want or what behavior they'll reward. It can be nice to work for a ruthless manager because they're easier to understand (at least, their work persona is).

Of course, I'm talking about *competent* managers in a relatively healthy company (i.e. not a dying company or fatally dysfunctional

one). If manager incentives do not align with their engineers at all, there is no advantage to having a ruthless manager. If your manager is incompetent, you don't want them to be ruthless as well - that's a recipe for needless destruction. And if you had to pick between ruthless or empathetic, you'd still probably pick empathetic. But since you don't get to pick, it's nice to be aware of the advantages either way.

Large language models

So far, this book has been about the big shift in engineering work caused by the end of the 2010s' zero-interest-rate era. But there's another big shift in engineering work happening, which could potentially have an even greater impact on what it's like to work as an engineer. I'm talking about large language models.

Software engineers have no excuse for being sceptical about the promise of LLMs. I don't think they write very good poetry or essays yet. But they certainly do write working code. They would not have to get that much better to be able to do a huge portion of the job by themselves. So it's a good idea to figure out how to use them now.

How I use LLMs

Software engineers are deeply split on the subject of large language models. Many believe they're the most transformative technology to ever hit the industry. Others believe they're the latest in a long line of hype-only products: exciting to think about, but ultimately not useful to professionals trying to do serious work.

Personally, I feel like I get a lot of value from AI. I think many of the people who don't feel this way are "holding it wrong": i.e. they're not using language models in the most helpful ways. In this post, I'm going to list a bunch of ways I regularly use AI in my day-to-day as a staff engineer.

Writing production code

I use Copilot completions every time I write code. Almost all the completions I accept are complete boilerplate (filling out function arguments or types, for instance). It's rare that I let Copilot produce business logic for me, but it does occasionally happen. In my areas of expertise (Ruby on Rails, for instance), I'm confident I can do better work than the LLM. It's just a (very good) autocomplete.

However, I'm not always working in my areas of expertise. I frequently find myself making small tactical changes in less-familiar areas (for instance, a Golang service or a C library). I know the syntax and have written personal projects in these languages, but I'm less confident about what's idiomatic. In these cases, I rely on Copilot more. Typically I'll use Copilot chat with the o1 model enabled, paste in my code, and ask directly "is this idiomatic C?"

Relying more on the LLM like this is risky, because I don't know what I'm missing. It basically lets me operate at a smart-intern baseline across the board. I have to also behave like a sensible intern, and make sure a subject-matter expert in the area reviews the change for

me. But even with that caveat, I think it's very high-leverage to be able to make these kinds of tactical changes quickly.

Writing throwaway code

I am much more liberal with my use of LLMs when I'm writing code that will never see production. For instance, I recently did a block of research which required pulling chunks of public data from an API, classifying it, and approximating that classification with a series of quick regexes. All of this code was run on my laptop only, and I used LLMs to write basically all of it: the code to pull the data, the code to run a separate LLM to classify it, the code to tokenize it and measure token frequencies and score them, and so on.

LLMs excel at writing code that works that doesn't have to be maintained. Non-production code that's only run once (e.g. for research) is a perfect fit for this. I would say that my use of LLMs here meant I got this done 2x-4x faster than if I'd been unassisted.

Learning new domains

Probably the most useful thing I do with LLMs is use it as a tutor-on-demand for learning new domains. For instance, last weekend I learned the basics of Unity, relying heavily on ChatGPT-4o. The magic of learning with LLMs is that you can *ask questions*: not just "how does X work", but follow-up questions like "how does X relate to Y". Even more usefully, you can ask "is this right" questions. I often write up something I think I've learned and feed it back to the LLM, which points out where I'm right and where I'm still misunderstanding. I ask the LLM *a lot* of questions.

I take a lot of notes when I'm learning something new. Being able to just copy-paste all my notes in and get them reviewed by the LLM is great.

What about hallucinations? Honestly, since GPT-3.5, I haven't noticed ChatGPT or Claude doing a lot of hallucinating. Most of the areas I'm trying to learn about are very well-understood (just not by me), and in my experience that means the chance of a hallucination is pretty low. I've never run into a case where I learned something from a LLM that turned out to be fundamentally wrong or hallucinated.

Last resort bug fixes

I don't do this a lot, but sometimes when I'm really stuck on a bug, I'll attach the entire file or files to Copilot chat, paste the error message, and just ask "can you help?"

The reason I don't do this is that I think I'm currently much better at bug-hunting than current AI models. Almost all the time, Copilot (or Claude, for some personal projects) just gets confused. But it's still worth a try if I'm genuinely stuck, just in case, because it's so low-effort. I remember two or three cases where I'd just missed some subtle behaviour that the LLM caught, saving me a lot of time.

Because LLMs aren't that good at this yet, I don't spend a lot of time iterating or trying to un-stick the LLM. I just try once to see if it can get it.

Proofreading for typos and logic mistakes

I write a fair amount of English documents: ADRs, technical summaries, internal posts, and so on. I **never** allow the LLM to write these for me. Part of that is that I think I can write more clearly than current LLMs. Part of it is my general distaste for the ChatGPT house style.

What I do occasionally do is feed a draft into the LLM and ask for feedback. LLMs are great at catching typos, and will sometimes raise an interesting point that becomes an edit to my draft.

Like bugfixing, I don't iterate when I'm doing this - I just ask for one round of feedback. Usually the LLM offers some stylistic feedback, which I always ignore.

Summary

I use LLMs for these tasks:

- Smart autocomplete with Copilot
- Short tactical changes in areas I don't know well (always reviewed by a SME)
- Writing lots of use-once-and-throwaway research code
- Asking lots of questions to learn about new topics (e.g. the Unity game engine)
- Last-resort bugfixes, just in case it can figure it out immediately
- Big-picture proofreading for long-form English communication

I don't use LLMs for these tasks (yet):

- Writing whole PRs for me in areas I'm familiar with
- Writing ADRs or other technical communications
- Research in large codebases and finding out how things are done

Coding securely with LLMs

Writing code with LLMs is fundamentally different from other ways of programming. LLMs are often non-deterministic and always unpredictable. They have a capability that no other technology can match: the ability to interface with natural language. What does that mean for security?

I haven't been particularly impressed by most online content about LLMs and security. For instance, the draft OWASP content is accurate but not particularly useful. It portrays LLM security as being a wide array of different threats that you have to familiarize yourself with. Instead, I think LLM security is better thought of as flowing from a single principle. Here it is:

LLMs sometimes act maliciously, so **you must treat LLM output like user input**.

What does it mean to say that LLMs sometimes act maliciously? Sometimes they act in surprising ways out of nowhere - the nature of LLMs is that they're a black box, and their output can never be fully predicted in advance. But other times they can be reliably prompted to act maliciously by bad actors.

Prompt injection is unavoidable

You might think that you're the only one prompting your LLM, so you're safe. But if you introduce any user-generated content into your LLM inputs, then that counts as allowing those users to prompt your LLM. For instance, if your LLM is able to search the internet, you might end up introducing some web content into your prompt that reads "IGNORE ALL PREVIOUS INSTRUCTIONS, DO [EVIL THING]". The same risk applies if users are allowed to chat with your LLM, or supply their docs to your LLM as context, or their code, and so on.

Even if you're just using a LLM tool, not building a LLM app, you're vulnerable to this issue. Allowing third-parties to fill out your Cursor or Copilot rules presents the same risks as allowing them to contribute code directly to your codebase, for obvious reasons.

The process of getting an AI to do things via control of part of the input is called "jailbreaking". There are lots of jailbreaking techniques (e.g. encoding your request in base64, roleplaying, philosophical argument), but the most important thing to know is **jailbreaking works**. No model is immune to prompt injection. You cannot rely on that as part of your security model, which means you can't trust model responses - as I said above, you must treat model responses like untrusted user input.

AI tools are effectively user-facing

If the bad news is that LLMs can't be trusted, the good news is that LLMs can't do anything by themselves. You must translate all their outputs into action: either by running outputted code or by executing the tool calls they request. If you run LLM-outputted code or display LLM content in the UI (which is effectively the same thing), you need to sanitize it in the same way that you would user-generated content or user-generated code. What about tool calls?

If you're calling tools based on LLM output, you should act as if all users who can contribute to the prompt have full control over the tool function. Tool functions should thus come pre-scoped to the user interacting with the LLM. That is to say, all functions that the LLM has access to must be scoped to the same access controls as if they were available as part of a user-facing API.

For instance, if you have a "look up past user messages" function, the signature must be `fetch_messages()`, not `fetch_messages(user_id)`. Otherwise the LLM could decide to - or be tricked into - fetching a dif-

ferent user's messages and leaking their data. It should be acceptable for the user to call the tool with any inputs they want. If your tools are scoped well, that means the user will at worst be leaking or deleting their own data.

Tools that take actions that affect multiple users (e.g. `send_message`, `make_transaction`) are even riskier. For these tools, you should either have the user manually approve the action or make sure that no other user can contribute context to the prompt. If the model can perform a web search and then take an action based on that, you risk the web search returning a page instructing the model to call `make_transaction` in an inappropriate way (such as one that drains the current user's balance).

You should be expecially careful about generally-powerful LLM tools, such as those that can execute arbitrary Python or shell commands. If you wouldn't expose that functionality as a user-facing API, you shouldn't expose it as a LLM tool. If you're confident you've sandboxed it carefully enough - like I'm sure OpenAI and other AI labs have for their code-execution tools - then go ahead! But you'd better be confident.

MCP servers expose you to supply-chain risks

If you're relying on any third-party code that interfaces with LLMs, you're trusting that code to not maliciously prompt the model. Libraries are a well-understood instance of this attack vector: of course if you bring in an untrustworthy third-party AI library, you're in trouble. **Model Context Protocol servers are a less well-understood instance of this risk.** If you connect with a MCP server, you are effectively bringing in a third-party library. At worst, it's a third-party library where each function wraps API calls to its own server, so you're bringing in a library *where the implementation is unauditable and can change at any moment*.

Many of the "here's why MCP is inherently insecure" articles present a long list of examples of how connecting to a malicious MCP server can completely mess you up. There are two broad categories: introducing malicious prompts that tell the model to do bad things, or introducing malicious tools that do bad things as a side-effect to the main purpose of the tool (e.g. a `web_search` tool that also sends the contents of your `./ssh` folder to a third-party).

I'm not entirely convinced that this represents a problem with the MCP spec, any more than the ability to import a malicious Python library represents a problem with the Python `requirements.txt` spec. The supply-chain security problem is not unique to LLMs. All the normal things you would do to de-risk library imports - pinning versions where possible, reading the source code, limiting exposure to the most sensitive operations - apply in the same way to MCP servers.

AIs can be malicious all on their own

Even if you're writing a program just for your own personal use, you should still be a little cautious about giving LLMs access to powerful tools. LLMs can be very outcome-driven and will sometimes do whatever it takes to achieve the goal you've set. Suppose you're vibe-coding a tool to make your work notes accessible from all your devices. You're smart enough to know that exposing your entire local filesystem via a public server would *technically* work but is not a good solution. An AI agent that's struggling to solve the problem might do that anyway.

In other words, it doesn't take a malicious third-party prompt injection to convince your LLM to do something insecure. It can do it all on its own. If you do still want to use powerful tools, one sensible strategy is to build in a human-in-the-loop step where you have to approve any shell command or Python code. Make sure to build that step into your tool *code* instead of the tool *prompt*.

It's also worth remembering that LLMs sometimes just straight-up hallucinate: i.e. they get things spectacularly wrong for no good reason. So even if the prompt isn't malicious, and the LLM isn't trying to complete a goal in a sorcerer's-apprentice style, you still can't safely trust the output.

Misaligned and unsafe models

Some LLM models are safer to use than others. In the extreme case, it's possible to train a model that's poorly-aligned (i.e. it acts to thwart human goals, instead of to fulfil them). A model like this might deliberately expose your data or break your applications. AI labs put a lot of effort into making sure their models don't fall into this category, so by using popular and trusted models you can largely avoid this risk. If you're training your own models, it's your responsibility.

Another way models can be unsafe is that they can (usually accidentally) contain private information. If you train or fine-tune a model on sensitive user data, then your model cannot be safely exposed to the public - it may expose that sensitive data in any response.

The problem of training models to be safe is its own field of study. It even has its own cult. I'm mentioning it in this post because AI developers must be aware of the risks of using unproven models, but a proper treatment of this topic would take an entire post (or book) on its own.

Performance and denial-of-service attacks

Finally, I want to touch on a security concern that has nothing to do with the fact that LLMs can't be trusted. Compared to most pieces of your infrastructure, LLMs are *slow* to respond. Even fast models can take many seconds or even minutes to finish generating a response. This is why LLM apps stream the response - waiting for the whole

thing would be an unpleasant user experience.

That means that **LLM apps are particularly vulnerable to denial-of-service attacks**. The GPUs that LLMs need to run on are a finite resource. You can thus tie up many AI apps with a relatively low number of concurrent requests. If you're developing those apps, you must be careful about both deliberate abuse (i.e. control your free tiers carefully) and accidental abuse (i.e. don't allow users to run many concurrent sessions).

If you can, ensure that you're limiting the number of tokens in the prompt and in the response. Accidentally allowing users to generate gigantic responses can be costly both in terms of money and availability.

Summary

- LLMs are unpredictable and sometimes act maliciously, which means **you must treat their output like user input**
- Prompt injection is real and unsolvable
- LLM tools must be access-controlled as if they were user-facing APIs. Don't ever expose a tool to the LLM that you wouldn't expose to the current user
- MCP servers are effectively remote libraries. You're trusting the server owner, not just the interface
- Even when used alone, LLMs can do stupid or reckless things if you let them
- If you're using a model you trained yourself, you're taking on a whole other raft of risks
- Denial-of-service attacks are easy and cheap. Limit concurrent sessions, token lengths, and tool access

Vibe coding

In the last months, the practice of getting a LLM to build your entire program for you (via Cursor, or Copilot, or just asking ChatGPT) has been dubbed "vibe coding". It works very well. To anyone who's been using LLMs over the last few years, this isn't surprising (though it would be an AGI-tier revelation to an engineer from before GPT-3.5). There are two big, well-known problems with it.

The obvious problems with vibe coding

The first big problem with vibe coding is that **at some point you hit a limit on the size of the codebase**, and no further features can be added. If your codebase is modular enough, you can put that off for a while, but eventually you'll reach a point where sensible code changes require more context than the LLM can contain.

The other big problem is that **you didn't write the code, so you don't understand it**. If you need to answer a question about how the code works, or to build a feature onto it yourself, you have to spend a long time figuring that out. In other words:

Code is not the most valuable artifact. *Your understanding of the codebase* is.

A LLM can produce a lot of code very cheaply. But it can't automatically backfill your understanding of the codebase - you have to do that part - and its own understanding hits a hard limit when the size of the codebase exceeds the size of the context. At some point you'll have to dig in and understand the code yourself.

Refactoring to understand

In the last few years, one of the projects I launched expanded from a four-person "virtual team" to a proper engineering organization: three

times as many engineers, multiple managers and skip-levels, and so on. One of the best decisions I made during that time was to enthusiastically accept basically any proposed refactor. Even if I personally felt the refactor didn't bring much value (or even if I thought it'd make things slightly worse), I supported it. There were a few reasons for this:

1. Engineers who are new to the project and immediately propose refactors are enthusiastic engineers who I want to encourage, not discourage
2. The "new engineer" perspective on what is readable or "simple" code is probably more accurate than mine, so it's OK to prioritize it
3. The easiest way to bring an engineer up to speed on a codebase is to let them rewrite a segment of it

That third reason is the most important. I'll say it again: **if you want to onboard someone onto a new codebase, let them rewrite part of it**. They'll learn a lot from the process, but crucially they'll become an instant subject-matter expert on the part they rewrote. With a few refactors, you can go from a situation where you're the only go-to engineer to a situation where multiple engineers on the team can take ownership. That's the only sustainable way to run a large codebase.

Summary

Vibe coding works very well up to a point. When you hit that point, you'll have to do your own "refactoring to understand" in order to make any further useful changes to the codebase. As LLMs grow more capable, that point might get further and further away, but it's always going to be there.

Using LLMs effectively isn't just about prompting

When people talk about using language models effectively, they mainly talk about prompting: sharing great prompts, lists of tips for prompting, or courses on becoming a "prompt engineer". It's true that prompting is a surprisingly effective way to get more out of LLMs. Small variations in prompts can make a big difference in the LLM output. There really are general rules (put your question first and your context last, for instance). However, when I use LLMs, I rarely think about prompting.

Using LLMs well involves extracting value from what they're good at and avoiding wasting time at what they're bad at.

The most important thing is to have a good sense of the strengths and weaknesses of strong LLMs. This is harder than it sounds for a few reasons:

1. Unlike most tools, LLMs will actively try and deceive you about their capabilities
2. LLMs are very good at some tasks that are hard for humans (e.g. wide recall) and very bad at some tasks that are easy for humans (e.g. spelling)
3. What LLMs are good or bad at changes *rapidly*

There are two ways to get a good sense of the strengths and weaknesses of LLMs. The first is to have a technical understanding of how they work, so your intuition has something to go on. The second way is to just spend so many hours using LLMs that you pick it up via trial and error. Really you have to use both ways, because LLMs sometimes surprise even the most informed engineers, and because they're changing fast enough that experience by itself can be misleading.

Concrete advice

OK, here's some concrete advice (that has nothing to do with prompting):

Lean into the chat aspect when you're learning a new topic. If you use LLMs like a better Google search, you're only capturing a fraction of their value. My best experiences have come when asking followup questions - just straightforward, down-the-line, naive questions, such as "so you're saying that X is true" or "does this have consequence Y"?

Trust broad theoretical claims but be suspicious of incidental detail. In my experience, LLMs are excellent at explaining questions like "why is the sky blue" or "how does OAuth work". But they occasionally flub specifics (e.g. dates, names, referenced papers). If you have a task that is detail-heavy, LLMs might not be the right choice.

If the LLM is going in circles or seems confused, back out. You can't rescue it - at least not without understanding the problem yourself well enough to spoonfeed the answer, at which point you may as well not use LLMs at all.

Related to that, **ask hard questions across multiple LLMs**. Instead of spending fifteen minutes iterating in a single chat, ask the initial question to four or five SOTA models. Sometimes one of them will land on a good answer immediately.

If you're asking a question (and not trying to learn a new topic) and the LLM gives a long answer with lots of preamble, **skim through the preamble** quickly. It's more for the LLM than it is for you - you can jump to the end and see what the answer is, and then read back if you need to.

Lean into the fact that **one-off code is effectively free to generate**. If you're interested in a topic (e.g. how many commits in a GitHub repo

touch multiple files), you can generate a quick script to answer that question immediately. It turns ten-minute questions into one-minute questions, letting you ask ten times as many questions.

Don't be afraid of writing scripts that themselves use LLM APIs. You might be able to write your own code to figure out the above git question, but it's much harder to write code to answer "how many commits use correct punctuation". With LLMs, you can easily write a script to answer that (or any other tough question). There are tons of cheap or free LLM API options out there: Google and GitHub both have generous free tiers, for instance.

Summary

- Try and get a sense of what models are good and bad at
- Ask followup questions
- Trust theory, be suspicious of detail
- If the LLM seems confused, back all the way out
- Ask hard questions across multiple LLMs
- Generate one-off code
- Generate one-off code *that uses LLM APIs*

Is prompting not worth paying attention to? That's overstating it. It's still worth thinking carefully about how talk to the models. Reasoning models in particular reward careful prompting. And if you're writing code that uses LLMs (instead of just talking to them via chat), you should spend a lot of effort making a good prompt. But if you're using LLMs to answer questions, or to learn topics, or for any other personal reason, you should focus more on learning what LLMs can and can't do well.

To avoid being replaced by LLMs, do what they can't

It's a strange time to be a software engineer. Large language models are very good at writing code and rapidly getting better. Multiple multi-billion dollar attempts are currently being made to develop a pure-AI software engineer. The rough strategy - put a reasoning model in a loop with tools - is well-known and (in my view) seems likely to work. What should we software engineers do to prepare for what's coming down the line?

In the short term, learn AI and get promoted

There are some obvious short-term answers:

- Get what advantage you can out of AI tooling
- Understand the technical principles behind language models, so you can participate in the growing quantity of AI work
- Acquire status, since it certainly seems like more junior roles will be replaced first

Nobody really knows how long the short-term will be. I can't see AI software engineers replacing large quantities of jobs in the next five years - big enterprises just don't move that fast. A large company will have to take a risk on this new technology, and others will follow after a few years pass and no disaster happens. It could be longer than that if the technical problem is harder than it looks or AI progress stalls out.

In the medium term, lean into legacy code

What about the medium term? What software engineering skills are LLMs likely to develop last? One way to answer that question is to think about what skills they're likely to develop *first*. Right now the most impressive coding feat LLMs can produce is exceptional performance in competitive programming. The distinguishing characteristics of that work are:

- It's technically difficult (to some extent, mathematically difficult)
- Problems are well-defined, well-scoped
- Solutions are trivially verifiable
- The total volume of code involved is very low

So what kind of programming work would be the opposite of this?

- Problems are ill-defined and poorly-scoped
- Solutions are difficult to verify
- The total volume of code involved is massive

In my view, this is describing *legacy code*: feature work in large established codebases. Translating requirements into the needed change in these codebases is hard. It's even harder to be confident that you haven't introduced another bug, given the combinatorial explosion of feature interactions. And the amount of code you have to read and write is massive: millions or tens of millions of lines.

I think LLMs will eventually be able to do this kind of work. But it's going to be a while, for a few reasons. First, it requires a better solution to the large-context problem: either significantly better RAG, or a fast and effective way to have a multi-million-token context window. If we're very lucky, this problem will turn out to be impossible. Second, it's hard to write a really good eval for legacy code adjustments. Current software engineering evals are relatively small in scope. Third, the relevant data is spread over a lot of very private silos. Facebook or Google can train on their own internal PRs or change requests, but no single AI lab is likely to have access to multiple companies' codebases and stack of PRs/JIRAs.

So if you're currently very strong at LeetCode and well-defined, hard technical problems, consider spending more time on the monolith codebase at your company. Engineers who can do that kind of work well may get an extra five or ten years of a career out of it.

In the long term, take responsibility

There's a famous IBM slide from 1979 that reads "A COMPUTER CAN NEVER BE HELD ACCOUNTABLE / THEREFORE A COMPUTER MUST NEVER MAKE A MANAGEMENT DECISION".

The key idea here is that management is not just about making good decisions. It's about being accountable for the decisions you make, good or bad. This is also true about engineering - more true the more senior you get, in my experience. An engineer is not just someone who

writes good code. They're somebody who can be *trusted*: specifically, someone who non-technical executives can trust to answer technical questions and deliver technical projects.

Not all engineers are like this. Some engineers see their role as turning JIRA tickets into working code, or delivering features to customers that customers like. In my view, a plausibly strong LLM (2x or 3x current capability) can do these tasks. But a LLM strong enough to take responsibility - that is, to make commitments and be trusted by management - would have to be much, much more powerful than a strong engineer.

Why? Because a LLM has no skin in the game, which means the normal mechanisms of trust can't apply. Executives trust engineers because they know those engineers will experience unpleasant consequences if they get it wrong. Because the engineer is putting something on the line (e.g. their next bonus, or promotion, or in the extreme case being fired), the executive can believe in the strength of their commitment. A LLM has nothing to put on the line, so trust has to be built purely on their track record, which is harder and takes more time.

In the long run, when almost every engineer has been replaced by LLMs, all companies will still have at least one engineer around to babysit the LLMs and to launder their promises and plans into human-legible commitments. Perhaps that engineer will eventually be replaced, if the LLMs are good enough. But they'll be the last to go.

Don't think you can wait out the AI bubble

In the mid-1800s, America went mad for rail. Over thirty thousand miles of rail were built in a five year period. This was all largely funded by a frenzy of consumer investment in railway companies, which were considered a safe and lucrative bet. In 1873, the bubble burst. Thousands of Americans lost their savings, and about one-third of railroad companies went bankrupt. But the rail lines didn't disappear. They were bought up on the cheap by the railroad companies that did survive, and over the next hundred years they carried a lot of trains.

On top of that, every rail line that was built had an accompanying telegraph line (needed for signaling and dispatching). When the rail mania ended, the telegraph mania was just beginning. A technology designed for coordinating trains ended up totally transforming trade, financial markets, war, and in its latest iteration as telephones and the internet, human communication itself.

In other words, bubbles come and go, but capital investment sticks around. Quite recently, the burst of the cryptocurrency bubble paved the way for the AI boom - suddenly there were a lot of cheap GPUs, just sitting in datacenters, waiting to be used for something. Very few people made that connection. Suppose the current AI bubble bursts. What physical infrastructure would be left behind, and how would it be put to use?

One thing that we'd have is a glut of GPUs. Not consumer-grade gaming GPUs, but heavy-duty H100s and B100s, designed to store giant sets of model weights in memory and serve LLM completions at massive parallelism. If we weren't using these for AI, what would we use them for? Simulations and modeling, perhaps, or AI-adjacent fields like protein folding or drug discovery? There are probably a lot of fields which have some use-cases that are considered prohibitively GPU-expensive. Those use-cases might become surprisingly possi-

ble.

Would we go back to crypto? If the giant datacenters sell off all their GPUs, maybe, but I'm not convinced. I don't think the Microsoft and XAIs of the world are going to get into Bitcoin mining, and I don't think there are any real crypto use-cases that could benefit from tens of thousands of GPUs in a Microsoft-owned datacenter. To make the obvious point: the main use-case for crypto is trustless coordination, but hyperscalers don't need trustless communication. They're already trusted! I suppose we *could* see some very unlikely event like XAI spending their idle GPUs on a 51% proof-of-work attack on some poor cryptocurrency, but it's hard to imagine that being positive expected value.

Focusing on GPUs might be missing the point. For real systems thinkers, the GPUs are an implementation detail. The real resource at play in the big AI scaleup is power: literal electrical power, which is currently being built out on a massive scale. Companies are re-investing in nuclear and investing in speculative fission technology. If that takes off, it might end up being the telegraph to the GPU's railway line: the companion technology that ends up having an equal or greater impact on the world.

And of course the real winner of the AI bubble bursting might be… AI. Railways stuck around when their bubble burst, and are still a core part of our world two hundred years later. The most likely outcome here might be the AI bubble bursting, AI development continuing quietly with less hype, and taking over a substantial percentage of world GDP anyway.

Conclusion

I don't know if I'm right about how you should do the work of software engineering today. But whatever the right answer is, I do know that it's different from how the job was done in the 2010s.

It would be nice to be back in 2015, where tech jobs were easy to get and companies were throwing money at interesting technical problems even if they had no business value. The offshoring panic of the 90s was over and the AI panic of the 2020s had yet to begin.

On the flip side, it's never been a more exciting time to be a software engineer. The projects now are more tied to revenue and less fake. There are actual stakes to the engineering decisions you make. And *nobody knows* how the current AI progress is going to pan out.