**Chapter 5. Leading Big Projects**

What makes a great project lead? It’s rarely genius: it’s perseverance, courage, and a willingness to talk to other people. Sure, there might be times when you need to come up with a brilliant and inspired solution. But, usually, the reason a project is difficult isn’t that you’re pushing the boundaries of technology, it’s that you’re dealing with ambiguity: unclear direction; messy, complicated humans; or legacy systems whose behavior you can’t predict. When the project involves a lot of teams; has big, risky decisions along the way; or is just messy and confusing, it needs a technical lead who will stick with it and trust that the problems can be solved, and who can handle the complexity. That’s often a staff engineer.

**The Life of a Project**

In this chapter we’re going to look at the life of a big, difficult project. While projects come in many shapes, as we saw in [Chapter 4](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch04.html#finite_time-id00004), I’m going to focus on the kind that lasts for at least several months and needs work from multiple teams. For the purposes of this chapter, I’ll assume that you’re the named technical lead of the project, perhaps delegating to some subleads of smaller parts. I’ll assume that nobody who is working on the project is reporting to you, but that you’re nonetheless expected to get results. There are probably other leaders involved: you might have project manager or product manager counterparts, and there could be some engineering managers, each of whom has a team working on their own areas.[**1**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn60) But, as the lead, *you’re* responsible for the result. That means you’re thinking about the *whole* problem, including the parts of it that lie in the fissures between the teams and the parts that aren’t really anyone’s job.[**2**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn61)

We’ll begin before the project even officially starts, when you’re looking at a vast, unmapped, and quite possibly overwhelming set of things to do. We’ll work through some techniques for getting the lay of the land and making sense of it all. And we’ll talk about creating the kinds of relationships where you’re sharing information and helping each other rather than competing.

Then we’ll set this thing up for success the way a project manager would: thinking through deliverables and milestones, setting expectations (including your own), defining your goals, adding accountability and structure, defining roles, and—the number one tool for success—*writing things down*.

After the project is set up and humming along, we’ll look at *driving* it. You’ve got a destination, and you’ll need to make some turns and course corrections to get there. I’ll talk about exploring the solution space, including framing the work, breaking the problem down, and building mental models around it. When a project is too big for any one person to track all of the details, narrative is vital. We’ll look at some common pitfalls you might meet during design, coding, and making big decisions. The chapter ends on spotting the obstacles in your path—the conflict, misalignment, or changes in destination, and how to communicate clearly while you navigate around them.

We won’t look at the end of the project until [Chapter 6](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#why_have_we_stoppedquestion_mark), but for now, let’s start at the very beginning.

**The Start of a Project**

The beginning of a project can be chaotic, as people mill around trying to figure out what they’re all doing and who’s in charge. Congratulations: as the technical lead, you’re in charge. Sort of.

The other people on the project aren’t your direct reports, and they’re still getting instructions from their managers. You have…maybe?…a mandate to get something done. But it’s possible that not everyone agrees yet on what that mandate *is* or whether they’re supposed to be helping you with it. If several different managers or directors are involved, it might not be clear what you’re responsible for and what they’re expecting to own. There might be other senior engineers on the project, maybe some more senior than you. Are they supposed to follow your lead? Do you have to take their advice?

**If You’re Feeling Overwhelmed…**

Maybe you’re joining an existing project, with all of its history and decisions and personality dynamics and documentation. Maybe the project is new, but there are already detailed requirements, a project spec, milestones, and a documented list of eager stakeholders. Or maybe there’s just a whiteboard scrawl, or—frustratingly often—a bunch of long email threads (some of which you weren’t cc’d on) that culminated in a director deciding to fund a project to solve a poorly articulated, unscoped problem. Almost certainly there are other people who want to give you their opinions on that problem, and there might be immediate deadlines you want to get ahead of. All of this comes up before you really have a handle on what the project is for, what your role is, and whether you all agree on what you’re trying to achieve. It’s a lot to think about.

What do you do? Where do you even *start?* Start with the overwhelm.

It’s normal to feel overwhelmed when you’re beginning a project. It takes time and energy to build the mental maps that let you navigate it all, and at the start of the project it might feel like more than you can handle. But, in the words of my friend Polina Giralt, [“that feeling of discomfort is called learning”](https://oreil.ly/2qXQH). Managing the discomfort is a skill you can learn.

You might even find yourself feeling that you’ve been put in this position by mistake or that the project is too hard for you, struggling with fear that you’ll let others down or fail publicly: a common phenomenon known as *imposter syndrome*. Emotional overwhelm can get in the way of absorbing knowledge and even affect your performance, making impostor syndrome almost self-fulfilling.

These feelings might be a signal that you’re low on one of the resources from [Chapter 4](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch04.html#finite_time-id00004). If you’ve exhausted all of your energy, you’re low on time, or you don’t feel like you have the skills to do what you need to do, that may manifest as stress and anxiety. Check in with yourself and ask whether any of your resources are at worrying levels.[**3**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn62) Is there anything you can do to get more time or energy or build more skills? Are there people who could help?

You might also think about how this work would feel if someone else was doing it. George Mauer, director of engineering at [findhelp.org](http://findhelp.org/), told me that he used to feel imposter syndrome, until he realized “99% of people don’t know better than I what to do.” Maybe you’re figuring out what you’re doing as you go along, but hey, everyone else is too! Is it just me, or is that *really* reassuring? No matter who was doing this project, they’d find it difficult too.

The difficulty is the point. I find that I can handle ambiguity when I internalize that this is the *nature of the work*. If it wasn’t messy and difficult, they wouldn’t need you. So, yes, you’re doing something hard here and you might make mistakes, but someone has to. The job here is to be the person brave enough to make—and own—the mistakes. You wouldn’t have gotten to this point in your career without credibility and social capital. A mistake will not destroy you. Ten mistakes will not destroy you. In fact, mistakes are how we learn. This is going to be OK.

Here are five things you can do to make a new project a little less overwhelming:

**Create an anchor for yourself**

Here’s how I start, no matter the size of the project: I create a document, just for me, that’s going to act as an external part of my brain for the duration of the project. It’s going to be full of uncertainty and rumors, leads to follow, reminders, bullet points, to-dos, and lists. When I’m not sure what to do next, I’ll return to that document and look at what Past Me thought was important. Putting absolutely everything in one place at least removes the “Where did I write that down?” problem.

**Talk to your project sponsor**

Understand who’s sponsoring this project and what they’ll want you to do for them. Then get some time with them. Go in prepared with a clear (ideally, written) description of what *you* think they’re hoping to achieve from the project and what success looks like. Ask them if they agree. If they don’t, or if there’s any ambiguity at all, write down what they’re telling you and double-check that you got it right. It’s surprisingly easy to misunderstand the mission, especially at the start of a project, and a conversation with your project sponsor can confirm that you’re on the right path (which is always reassuring). This is also a good time to clear up any confusion about what your role will be and who you should bring project updates to.

Depending on the project sponsor, you might have regular access to them, or you might get a single conversation and then nothing more for months (a horrible way to work, but it does happen). The less often you’re going to talk with them, the more vital it is that you get all of the information up front.

**Decide who gets your uncertainty**

Think about who you’re going to talk with when the project is difficult and you’re feeling out of your depth. Your junior engineers are not the right people! While you can and should be open with them about some of the difficulties ahead, they’re looking to you for safety and stability. Yes, you should show your less seasoned colleagues that senior people are learning too, but don’t let your fears spill onto them. Part of your job will be to remove stress for them, making this a project that will give them quality of life, skills, energy, credibility, and social capital.

That doesn’t mean you should carry your worries alone. Try to find at least one person who you can be open and unsure with. This might be your manager, a mentor, or a peer: the staff engineer peers I discussed in [Chapter 2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch02.html#three_maps) can be perfect here. Choose a sounding board who will listen, validate, and say “Yes, this stuff is hard for me too” rather than refusing to ever admit weakness or just trying to solve your problems for you. And, of course, be that person for them or others too.

**Give yourself a win**

If the problem is still too big, aim to take a step, any step, that helps you exert some control over it. Talk to someone. Draw a picture. Create a document. Describe the problem to someone else. In some ways, the start of the project is when it’s easiest to not know things. You can preface any statement with “I’m new to this, so tell me if I have this wrong, but here’s what I think we’re doing” and learn a lot. Later on, it becomes a little more cognitively expensive or may even feel a little embarrassing not to know things. (It’s not, though! Learning is great!) Don’t waste the brief period where it’s easy to not know.

**Use your strengths**

Remember how, when we talked about strategy in [Chapter 3](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch03.html#creating_the_big_picture), I said you should build a strategy around your advantages? That’s true here, too. You’re going to want to pour a lot of information into your brain as efficiently as possible, so use your core muscles. If you’re most comfortable with code, jump in. If you tend to go first to relationships, talk to people. If you’re a reader, go get the documents. Probably your preferred place to start won’t give you all of the information you need, but it’ll be a good place to start convincing your brain that this is just another project. Seriously, you’ve got this.

**Building Context**

The start of a project will be full of ambiguity. You can create perspective, for yourself and others, by taking on a mapping exercise like we did in [Chapter 2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch02.html#three_maps). That means building your *locator map*: putting the work in perspective; understanding the goals, constraints, and history of the project; and being clear about how it ties back to business goals. It means filling out your *topographical map*: identifying the terrain you’re crossing and the local politics there, how the people on the project like to work, and how decisions will get made. And of course you’ll need a *treasure map* that shows where you’re all going and what milestones you’ll be stopping at along the way.

Here are some points of context you’ll need to clarify for yourself and for everyone else:

**Goals**

Why are you doing this project? Out of all of the possible business goals, the technical investments, and the pending tasks, why is *this* the one that’s happening? The “why” is going to be a motivator and a guide throughout the project. If you’re setting off to do something and you don’t know *why*, chances are you’ll do the wrong thing. You might complete the work without solving the real problem you were intended to solve. I’ll talk about this phenomenon more in [Chapter 6](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#why_have_we_stoppedquestion_mark).

Understanding the “why” might even make you reject the premise of the project: if the project you’ve been asked to lead won’t actually achieve the goal, completing it would be a waste of everyone’s time. Better to find out early.

**Customer needs**

A story I tell a lot is about my first week in a new infrastructure team. A member of the team described a project they were working on, upgrading some system to make a new feature available. Another team, he said, needed the feature. “Why do they need it?” I asked, glad of an opportunity to get the lay of the land. “Maybe they don’t,” he said. “We think they do, but we have no way of knowing.” These two teams sat in the same building, *on the same floor*.

Even on the most internal project, you have “customers”: someone is going to use the things you’re creating. Sometimes you’ll be your own customer. Most of the time it’s going to be other people. If you don’t understand what your customers need, you’re not going to build the right thing. And if you don’t have a product manager, you’re probably on the hook for figuring out what those needs are. That means talking to your customers and listening to what they say in response.

Product management is a huge and difficult discipline, and it’s not easy to understand what your users actually want—as opposed to what they’re telling you.[**4**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn63) It takes time, so budget that time. Ask a user to let you shadow them using the software you’re replacing. Ask internal users to describe the API they wish they had, or show them a sketch of the interface you think they want and see how they interact with it. Don’t mentally fill in what you wish they’d said; listen to their actual responses. Try not to use jargon, because people can get intimidated and not want to tell you that they didn’t understand. If you’re lucky enough to have UX researchers on your team to study the customer experience, make sure to read their work, talk to them, and try to observe some user interviews.

Even if you *do* have a product manager, that doesn’t mean you get to ignore your customers! I love the conversations with product managers that Gergely Orosz, author of the newsletter The Pragmatic Engineer, aggregates in his article [“Working with Product Managers: Advice from PMs”](https://oreil.ly/l9ofc), especially Ebi Atawodi’s comment that “You are also ‘product.’” Atawodi points out that engineering teams should be just as customer-obsessed as product teams, caring about business context, key metrics, and the customer experience.

**Success metrics**

Describe how you’ll measure your success. If you’re creating a new feature, maybe there’s already a proposed way to measure success, like a [product requirements document](https://oreil.ly/YBaZO) (PRD). If not, you might be proposing your own metrics. Either way, you’ll need to make sure that your sponsor and any other leads on the project agree on them.

Success metrics aren’t always obvious. Software projects sometimes implicitly measure progress by how much of the code is written, but the existence of code tells you nothing about whether any problem has actually been solved. In some cases, the real success will come from *deleting* lines of code. Think about what success will really look like for your project. Will it mean more revenue from users, fewer outages, a process that takes less time? Is there an objective metric you can set up now that will let you compare before and after? In her Kubecon keynote, [“The Challenges of Migrating 150+ Microservices to Kubernetes”](https://oreil.ly/OTILj), microservices expert Sarah Wells spoke about judging the success of a migration in two measurable ways: the amount of time spent keeping the cluster healthy, and the number of snarky messages from team members on Slack about functionality that didn’t work as expected.

If you initiated the project, be even more disciplined about defining success metrics. If your credibility and social capital are strong, you can sometimes convince other people to get behind a project based on their belief in you or by means of a compelling document or an inspirational speech. But you can’t be certain that you’re right! Treat your own ideas with the most skepticism and get real, measurable goals in place quickly, so you can see how the project is trending. As [Chapter 4](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch04.html#finite_time-id00004) explained, your credibility and social capital can go down as well as up. Don’t rely on them as the only motivator to keep the project going.

**Sponsors, stakeholders, and customers**

Who wants this project and who’s paying for it? Who are the main customers of the project? Are they internal or external? What do they want? Is there an intermediate person between you and the original project sponsor? If there’s a product requirements document, this may all be spelled out, but you might have to clarify for yourself who your first customer or main stakeholder is, what they’re hoping to see from you, and when. If the impetus for the work has come from you, then you might be on the hook to continually justify the project and make sure it stays funded. It will be easier to sell the value of the work if you can find other people who want it too.

**Fixed constraints**

Maybe there are some senior technical roles where you can walk in and start solving big problems, unconstrained by budget, time, difficult people, or other annoying aspects of reality. I’ve never seen a role like that, though. Usually you’re going to be constrained in some ways: understand what those constraints are. Are there deadlines that absolutely can’t move? Do you have a budget? Are there teams you depend on that might be too busy to help you, or system components you won’t be able to use? Will you have to work with difficult people?

Understanding your constraints will set your own expectations and other people’s too. There’s a big difference between “ship a feature” and “ship a feature without enough engineers and with two stakeholders who disagree on the direction.” Similarly, creating an internal platform for teams that are eager to beta test it is a different project from trying to convince a hundred engineers to migrate to a new system that they hate. Describe the reality of the situation you’re in, so you won’t spend all your time being mad at reality for not being as you wish it to be.[**5**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn64)

**Risks**

Is this a “moonshot” or a “roofshot” project? Does it feel huge and aspirational, or a fairly straightforward step in the right direction? In an ideal world, everyone on the project would deliver their own part in perfect synchronization, with predictable availability of time and energy (and ideally a boost to credibility, skills, quality of life, and social capital along the way!). The reality is that some things *will* go wrong, and the more ambitious the project, the riskier it will be. Try to predict some of the risks. What could happen that could prevent you from reaching your goals on deadline? Are there unknowns, dependencies, key people who will doom the project if they quit? You can mitigate risk by being clear about your areas of uncertainty. What don’t you know, and what approaches can you take that will make those less ambiguous? Is this the sort of thing you can prototype? Has someone done this kind of project before?

One of the most common risks is the fear of wasted effort, of creating something that ends up never getting used. If you make frequent iterative changes, you have a better chance of getting user feedback and course-correcting (or even canceling the project early; we’ll look at that more in [Chapter 6](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#why_have_we_stoppedquestion_mark)) than if you have a single win-or-lose release at the end.

**History**

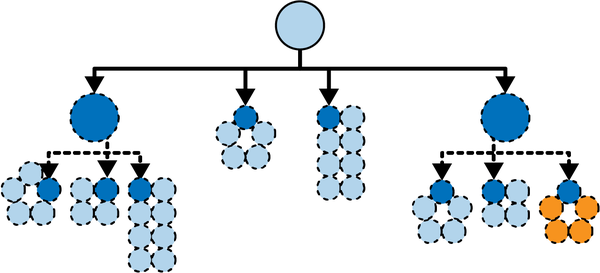
Even if this is a brand-new project, there’s going to be some historical context you need to know. Where did the idea for the project come from? Has it been announced in an all-hands meeting or email that has set expectations? If the project is not brand new, its history may be murky and fraught. When teams have already tried and failed to solve a problem, there might be leftover components you’ll be expected to use or build on, or existing users with odd use cases who will want you to keep supporting them. You might also face resentment and irritation from the people who tried and failed, and you’ll need to proceed very carefully if you want to engage their enthusiasm again.

If you’re new to an existing project, don’t just jump in. Have a lot of conversations. Find out what half-built systems you’re going to have to use, work around, or clean up before you can start creating a new solution. Understand people’s feelings and expectations, and learn from their experiences. Remember that tenet of Amazon’s principal engineer community I mentioned in [Chapter 2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch02.html#three_maps): “Respect what came before.”

**Team**

Depending on the size of the project, you might have a few key people to get to know or a massive cast of team members, leads, stakeholders, customers, and people in nearby roles, some of whom influence your direction, some who’ll make decisions you have to react to, and some of whom you’ll never speak with directly. There’ll also be other people in leadership roles.

If you’re the lead of a project that only includes one team, you’ll probably talk regularly with everyone on that team. On a bigger project with many teams involved, you’ll need a contact person on each team. For even bigger projects, you might have a sublead in each area (see [Figure 5-1](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#as_the_project_lead_left_parenthesisthe)). Or maybe your project is just one part of a broader project and *you’ll* be a sublead.



**Figure 5-1. As the project lead (the root node of this tree) you may have contacts or subleads on each of several other teams. Some of those subleads may be directing work for their own subleads.**

It’s vital to build good working relationships with all of the other leaders and help each other out. Don’t waste your time in power struggles. You’ll be more likely to achieve your shared goals if you work well together, and of course work is a much more pleasant endeavor (higher quality of life, higher energy!) when you’re harmonious with the people around you. Unfortunately, having multiple leaders often means unclear expectations about who’s doing what—a common source of conflict. Understand who the leaders are, how they’re involved in the project, and what role they expect to play.

**Giving Your Project Structure**

With all of that context in mind, you can start setting up the formal structures that will help you run the project. Setting expectations and structure and following a plan can be time-consuming, but they really do increase the likelihood that the thing you’re eager to start working on will actually succeed. The more people involved, the more you’ll want to make sure you’re all aligned on your expectations. These structures will also act as tools to help you feel in control of what’s going on—so if you’re still a little overwhelmed, don’t worry, this will make it easier.

Here are some of the things you’ll do to set up a project.

**Defining roles**

I mentioned the risk of conflict when there are multiple leaders, so let’s start there. At senior levels, engineering roles start to blur into each other: the difference between, say, a very senior engineer, an engineering manager, and a technical program manager might not be immediately clear. At a baseline, all of them have some responsibility to be the “grown-up in the room,” identify risks, remove blockers, and solve problems. By definition, the manager has direct reports, but the engineer might too. The program manager sees the gaps, communicates about the project status, and removes blockers, but everyone else should step up and do those things if they aren’t happening. We might argue that the engineer should have deeper technical skills, but some program managers are in deeply technical roles and many have extensive software engineering experience. This is even more complicated when a manager or TPM comes from an engineering background and is still involved in big technical decisions, or when there’s more than one staff engineer. Who does what?

The beginning of a project is the best time to lay out each leader’s responsibilities. Rather than waiting until two people discover they’re doing the same job, or work is slipping through the cracks because nobody thinks it’s theirs, you can describe what kinds of things will need to be done and who will do them. The simplest approach is to create a table of leadership responsibilities and lay out who should take on each one. [Table 5-1](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#example_table_of_leadership_responsibil) gives an example.

|  |  |
| --- | --- |
| **Product Manager** | Olayemi |
| **Technical Lead** | Jaya |
| **Engineering Manager** | Kai |
| **Technical Program Manager** | Nana |
| **Engineering Team** | Adel, Sam, Kravann |
| **Understanding customer needs and providing initial requirements** | Product Manager |
| **Providing KPIs for product success** | Product Manager |
| **Setting timelines** | Technical Program Manager |
| **Setting scope and milestones** | Product Manager, Engineering Manager |
| **Recruiting new team members** | Engineering Manager |
| **Monitoring and ensuring team health** | Engineering Manager |
| **Managing team members’ performance and growth** | Engineering Manager |
| **Mentoring and coaching on technical topics** | Technical Lead |
| **Designing high-level architecture** | Technical Lead (with support from engineering team) |
| **Designing individual components** | Technical Lead, Engineering Team |
| **Coding** | Engineering Team (with support from Technical Lead) |
| **Testing** | Engineering Team (with support from Technical Lead) |
| **Operating, deploying, and monitoring systems** | Engineering Team, Technical Lead |
| **Communicating status to stakeholders** | Technical Program Manager |
| **Devising A/B experiments** | Product Manager |
| **Making final decisions on technical approach** | Technical Lead |
| **Making final decisions on user-visible behavior** | Product Manager |
| Table 5-1. Example table of leadership responsibilities | |

I really want to emphasize that this is just an example! Some projects will have many more leaders than this. Some will have fewer. Internal-facing projects, like on infrastructure teams, usually won’t have product managers.[**6**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn65) If you would have put different names on different tasks here, that’s *fine.* If you want to make all of this more sophisticated, a popular tool is RACI, also known as a [responsibility assignment matrix](https://oreil.ly/eebGs). Its name comes from the four key responsibilities most typically used:

*Responsible*

The person actually doing the work.

*Accountable*

The person ultimately delivering the work and responsible for signing off that it’s done. There’s supposed to be only one accountable person per task, and it will often be the same person as whoever is “Responsible.”

*Consulted*

People who are asked for their opinion.

*Informed*

People who will be kept up to date on progress.

If you’re really into project management, you’ll probably enjoy reading about the many, many variants of RACI, but I’m not going into that here. I’ll just note that RACI turns the preceding list into a matrix, so you can set everyone’s expectations even more clearly. It can be overkill for some situations, but when you need it, you *really* need it. A staff engineer friend at Google told me about using RACI for a chaotic project:

*We needed some kind of formal framework to explicitly define who would be making decisions. This helped us break out of two bad patterns: never making a decision because we didn’t know who the decider was, so we’d just discuss forever, and relitigating every decision over and over because we didn’t have a process for making decisions. RACI didn’t solve either of those problems entirely, but it at least provided some (fairly uncontroversial) structure for people.*

The uncontroversial structure is the real superpower here. It gives you a way to broach the conversation without it being weird.

Lara Hogan offers an alternative tool for product engineering projects, the [team leader Venn diagram](https://oreil.ly/Eoi2I), which has overlapping circles for the stories of “what,” “how,” and “why,” which are then assigned to the engineering manager, the engineering lead, and the product manager. I’ve heard suggestions that a fourth circle, the story of “when,” if you could squeeze it into that diagram, might be assigned to a project or program manager.

However you approach it, try to get every leader aligned on what your roles are and who’s doing what. If you aren’t sure that everyone knows you’re the lead (or even that you are), then the stress of a new project gets even worse. As a sublead for a project more than a decade ago, I found myself showing up at my weekly meetings with the overall lead a little nervous about whether I’d done what was expected of me. I could have saved myself a whole lot of anxiety by having a direct conversation about it: “Here’s what I think I’m responsible for. Do you agree? Am I taking the right amount of ownership?”

Last thought on roles: if you’re the project lead, you are ultimately responsible for the project. That means you’re implicitly filling any roles that don’t already have someone in them, or at least making sure the work gets done. If your teammates have no manager, you’re going to be helping them grow. If there’s nobody tracking user requirements, that’s you. If nobody is project managing, that’s you as well. It can add up to a lot. For the rest of this section, I’m going to talk about some tasks that may be assigned to you in these roles.

**Recruiting people**

If there are unfilled roles that you don’t want to do or don’t have time to do, you may have to find someone else to do them. That might mean recruiting someone internally or externally, or picking subleads to be responsible for parts of your project. Sometimes that will mean you’re looking for specific technical skills that your team doesn’t have enough of or experience you don’t have.[**7**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn66) Look also for the people who complement or fill the gaps in your own skill set. If you’re a big-picture person, look for someone who loves getting into the details, and vice versa. For bonus points, see if you can find people who absolutely *love* doing the kind of work that you hate to do. That’s the best kind of partnership!

I had an opportunity to lead a panel for LeadDev in October 2020, [“Sustaining and Growing Motivation Across Projects”](https://oreil.ly/o6Rj1). In it, Mohit Cheppudira, principal engineer at Google, talked about what he looks for when he recruits people:

*When you’re responsible for a really big project, you’re kind of building an organization and you’re steering an organization. It’s important to get the needs of the organization right. I spent a lot of time trying to make sure that I had the best leads in all the different domains that were involved in that one specific project. And, when you’re looking for leads, you’re looking for people that don’t just have good technical judgment, but also have the attitudes: they are optimistic, good at conflict resolution, good at communication. You want people that you can rely on to actually drive this project forward.*

Recruiting decisions are some of the most important you will make. The people you bring onto the project will make a huge difference in whether you meet your deadlines, complete visible tasks, and achieve your goals. Their success is your success, and their failure is very much your failure. Recruit people who will work together, push through friction, and get the job done—people you can rely on.

**Agreeing on scope**

Project managers sometimes use a model called the [project management triangle](https://oreil.ly/4aboG), which balances a project’s time, budget, and scope. You’ll sometimes also hear this framed as “Fast, cheap, good: Pick two.” It feels obvious to say, but it’s somehow easy to forget: if you have fewer people, you can’t do as much. Agree on what you’re going to try to do.

Probably you’re not going to deliver the whole project in one chunk. If you have multiple use cases or features, you’ll want to deliver incremental value along the way. So decide what you’re doing first, set a milestone, and put a date beside it. Describe what that milestone looks like: what features are included? What can a user do?

Jackie Benowitz, an engineering manager who has led several huge cross-organization projects, told me that she thinks about milestones as beta tests: every milestone is usable or demonstrable in some way, and gives the users or stakeholders an extra opportunity to give feedback. That means you have to be prepared for each incremental change to potentially change the user requirements for the next one, because changing what your users *can* do will help them realize what else they *want* to do. They might also tell you that you’re on the wrong track, giving you an early opportunity to change your direction.

To maintain this kind of flexibility, some projects won’t plan much further than the next milestone, considering each one to be a destination in itself. Others will roughly map out the entire project, updating the map when a change of direction is needed. Whichever you prefer, make the increments small enough that there’s always a milestone in sight: it’s motivational to have a goal that feels reachable. I’ve also found again and again that people don’t act with a sense of urgency until there’s a deadline that they can’t avoid thinking about. Regular deliverables will discourage people from leaving everything until the end. Set clear expectations about what you expect to happen when.

If the project is big enough, you might split the work into *workstreams*, chunks of functionality that can be created in parallel (perhaps with different subteams), each with its own set of milestones. They may depend on each other at key junctures, and you may have streams that can’t start until others are completely finished, but usually you can talk about any one of them independently from the others. You might also describe different *phases,* where you complete a huge piece of work, reorient, and then kick off the next stage of the project. Splitting the work up like this makes it a little more manageable to think about. It lets you add an abstraction and think at a higher altitude. If you can say a particular workstream is on track, for example, then you don’t need to get into the weeds of each task on that stream.

If your company is using product management or roadmapping software, it will probably have functionality to organize your project into phases, workstreams, or milestones. If everyone who needs to participate is in one physical place, you can do the same thing with sticky notes on a whiteboard. What matters is that you all get the same clear picture of what you’ve decided you’re doing and when.

**Estimating time**

I have met almost nobody who is good at time estimation. This may be the nature of software engineering: every project is different, and the only way we can tell how long a project will take is if we’ve done exactly the same thing before. The most common advice I’ve read is to break the work up into the smallest tasks you can, since those are easiest to estimate. The second most common is to assume you’re wrong and multiply everything by three. Neither approach is very satisfying!

I prefer the advice Andy Hunt and Dave Thomas give in [*The Pragmatic Programmer*](https://oreil.ly/KCF9o) (O’Reilly): “We find that often the only way to determine the timetable for a project is by gaining experience on that same project.” As you deliver small slices of functionality, they explain, you gain experience in how long it will take your team to do something—so you update your schedules every time. They also recommend that you practice estimating and keep a log of how that’s going. Like every other skill, the more you do it, the better you’ll get at it, so practice estimating even when it doesn’t matter and see if more of your estimates are right over time.

Estimating time needs to include thinking about teams you depend on. Some of these teams might be fully invested in the project, perhaps considering it their main priority for the quarter or year. Others may see it as just one of many requests competing for their attention. Talk with the teams you’ll need as early as possible, and understand their availability.

Engineers in platform teams in particular have told me about the frustration of receiving a last-minute request to add functionality that’s needed immediately for launch, functionality they could have easily provided if they’d known about it a few months earlier, when they could have incorporated it into their planning for the quarter. The later you tell other teams you’ll need something from them, the less likely you are to get what you need. If they do agree to scramble to accommodate you, bear in mind that you’re interrupting their previous work: you’re disrupting the time estimation for their other projects.

**Agreeing on logistics**

There are a lot of small decisions that can help set your project up to run smoothly, and you’ll probably want to discuss them as a team. Here are some examples:

*When, where, and how you’ll meet*

How often are you going to have meetings? If you’re a single team, will you have daily standups? If you’re working across multiple teams, how often will the leads get together? Will you have regular demos, agile ceremonies, retrospectives, or some other way to reflect?

*How you’ll encourage informal communication*

Meetings can be a fairly formal way to exchange information, and they probably don’t happen every day. How can you make it easy for people to chat with each other and ask questions in the meantime? If you all sit together, this can be pretty easy, but in an increasingly remote workforce, that’s starting to become unusual.[**8**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn67) In a new project where people don’t know each other, they may be hesitant to send DMs, so you can encourage conversation with a social channel, informal icebreaker meetings, or (if people’s lives allow it) getting everyone in the same place for a couple of days. Even silly things like meme threads can give people a connection that will make them quicker to ask each other questions and offer help.

*How you’ll all share status*

How would your sponsor like to find out what’s going on with the project? What about the rest of the company: where do you want them to go to find out more? If you’re planning to send regular update emails, who will send them and at what cadence?

*Where the documentation home will be*

Does the project have an official home on the company wiki or documentation platform? If not, create one and make it easy to find with a memorable URL or prominent link. This documentation space will be the center of the project’s universe and should link out to everything else. It will give you a single place to start when you’re looking for a meeting recap, a description of the next milestone, or the wording of a related OKR. You want everyone to be looking at the same up-to-date information. It’s a single fixed point in a chaotic universe!

*What your development practices will be*

In what languages are you going to work? How are you going to deploy whatever you create? What are your standards for code review? How tested should everything be? Are you releasing behind feature flags? If you’re adding a new project in a company that has been around for a while, maybe there are standard answers for all of these questions. Others may depend on technical decisions you’ll make as you work through the project. Begin the discussions, though, and get everyone aligned.

**Having a kickoff meeting**

The last thing you might do as part of setting up a project is to have a kickoff meeting. If all of the important information is written down already, this might feel unnecessary, but there’s something about seeing each other’s faces that starts a project with momentum. It gives everyone an opportunity to sync up and feel like part of a team.

Here are some topics you might cover at your kickoff:

* Who everyone is
* What the goals of the project are
* What’s happened so far
* What structures you’ve all set up
* What’s happening next
* What you want people to do[**9**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn68)
* How people can ask questions and find out more

**Driving the Project**

My favorite talk about managing projects is [“Avoid the Lake!”](https://oreil.ly/NHWFj), by Kripa Krishnan, VP of Google Cloud Platform. I’d often heard the term “driving a project” without really thinking about what that means, but Krishnan makes the analogy clear when she says, “Driving doesn’t mean you put your foot on the gas and you just go straight.” Driving, in other words, can’t be passive: it’s an active, deliberate, mindful role. It means choosing your route, making decisions, and reacting to hazards on the road ahead. If you’re the project lead, you’re in the driver’s seat. You’re responsible for getting everyone safely to the destination.

In [Chapter 3](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch03.html#creating_the_big_picture) we looked at one of the responsibilities of a project lead: making sure decisions get made. We’re going to look now at some of the other challenges you might encounter on the road as you drive your project toward its destination.

**Exploring**

I’m always suspicious when a brand-new project already has a design document or plan, and even more when those include implementation details: “Build a GraphQL server with Node.js to…” and so forth. Unless the problem is really straightforward (in which case, are you sure it needs a staff engineer?), you won’t have enough information about it on day one to make these kinds of granular decisions. It will take some research and exploration to understand the project’s true needs and evaluate the approaches you might take to achieve them. If you’re creating a design where it’s difficult to articulate the goals (or if the goals are just a description of your implementation!), that’s a sign that you haven’t spent enough time in this exploration stage.

**What are the important aspects of the project?**

What are you all setting out to achieve? The bigger the project, the more likely it is that different teams have different mental models of what you’re trying to achieve, what will be different once you’ve achieved it, and what approach you’re all taking. Some teams might have constraints that you don’t know about, or unspoken assumptions about the direction the project will take: they might have only agreed to help you because they think your project will also achieve some other goal they care about—and they might be wrong! Team members may fixate on smaller, less important aspects of the project or niche use cases, or expect a different scope than you do. They may be using different vocabulary to describe the same thing, or using the same words but meaning something different. Get to the point where you can concisely explain what different teams in the project want in a way that they’ll agree is accurate.

Aligning and framing the problem can take time and effort. It will involve talking to your users and stakeholders—and actually listening to what they say and exactly what words they use. It may involve researching other teams’ work to understand if they’re doing the same thing as you, just described differently. If you’re going into this project with well-formed mental models, it can be difficult to set those preconceptions aside and explore how other people think about the work. But it will be even more painful to try to drive a project where everyone’s using different words, or is aiming for a different destination.

As you explore, and uncover expectations, you’ll start building up a crisp definition of what you’re doing. Exploring helps you form an elevator pitch about the project, a way to sum it up and reduce it to its most important aspects. You’ll also start building up a clear description of what you’re not doing. Where projects are related to yours, you’ll begin to show how one is a subset of the other, or how they overlap. It’s clarifying to describe work that seems similar but isn’t actually related, or work that seems entirely unrelated but has unexpected connections. I’ll talk a little later in this chapter about building mental models to help you and others think about a problem in the same way. The better you understand the problem, the easier it will be to frame it for other people.

**What possible approaches can you take?**

Once you have a clear story for what you’re trying to do, only then figure out how to do it. If you’ve gone into the project with an architecture or a solution in mind, it can be jarring to realize that it might not actually solve the real problem that you’ve framed as part of your exploration. This is such a difficult mental adjustment that I’ve seen project leads cling tightly to their original ideas about what problem they’re solving, resisting all information that contradicts that worldview. That doesn’t make for a good solution. So really try to keep an open mind about how you’re solving the problem until you have agreed on what you need to solve.

Be open to existing solutions too, even if they’re less interesting or convenient than creating something new. In [Chapter 2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch02.html#three_maps), I talked about building perspective by studying and learning from other teams (both in your company and outside) before diving into creating some new thing. The existing work might not be exactly the shape of whatever you’ve been envisioning, but be receptive to the idea that it might be a better shape, or at least a workable one. Learn from history: understand whether similar projects have succeeded or failed, and where they struggled. Remember that creating code is just one of the stages of software engineering: running code needs to be maintained, operated, deployed, monitored, and someday deleted. If there’s a solution that means your organization has fewer things to maintain after your project, weigh that up when you’re choosing your approach.

**Clarifying**

A big part of starting the project will be giving everyone mental models for what you’re all doing. When there are a lot of moving parts, opinions, and semirelated projects, it’s a strain to keep track of them all. As the project lead, you have an incentive to spend time understanding the tricky concepts if it helps you achieve your project. But the people you ask for help have a different focus and may not try as hard. Unless you take the time to reduce the complexity for them, they could end up thinking about the project in a way that leads them to optimize for the wrong outcome or muddy a clear story you’re trying to tell your organization.

In *The Art of Travel (Vintage),* Alain de Botton talks about the frustration of learning new information that doesn’t connect to anything you already know—like the sorts of facts you might pick up while visiting a historic building in a foreign land. He writes about visiting Madrid’s Iglesia de San Francisco el Grande and learning that “the sixteenth-century stalls in the sacristy and chapter house come from the Cartuja de El Paular, the Carthusian monastery near Segovia.” Without a connection back to something he was already familiar with, the description couldn’t spark his excitement or curiosity. The new facts, he wrote, were “as useless and fugitive as necklace beads without a connecting chain.”

I love that quote and think about it a lot while trying to help other people understand something. How can I hook this concept onto their existing knowledge? How can I make it relevant and spark their curiosity about it? Maybe I can build a necklace chain back via connecting concepts, or use an analogy to give them an idea that’s close enough to be useful, even if it’s not exactly correct.

Let’s look at a few ways you can reduce the complexity of big messy projects by building shared understanding.

**Mental models**

When you start learning about Kubernetes, you’re deluged with new terms: Pods, Services, Namespaces, Deployments, Kubelets, ReplicaSets, Controllers, Jobs, and so forth. Most documentation explains each of these concepts through its relationship to *other* new terms, or describes them in abstract ways that make perfect sense *if* you already understand the whole domain. If you’re coming in cold, it can be overwhelming—until a friend frames it in relation to something familiar. They might use an analogy that lets you imagine the behavior of something you already know: “Think of this part like a UNIX process.” They might use an example instead, to give you a hint to the shape of the concept being described: “This is likely to be a Docker container.” Neither of these models is perfect, but they don’t have to be: they have to be *close enough* to make a chain back to some other thing you already understand, to give you something to hook the knowledge onto.

I’ve deployed these sorts of rhetorical devices throughout this book, using video game analogies and geographical metaphors to describe concepts. Connecting an abstract idea back to something I understand well removes some of the cognitive cost of retaining and describing the idea. It’s like I’m putting the idea into a well-named function that I can call again later without needing to think about its internals. (See, I just did it again.)

Just like we build APIs and interfaces to let us work with components without having to deal with their messy details, we can build abstractions to let us work with ideas. “Leader election” is something we can understand and explain more easily than “distributed consensus algorithm.” As you describe the project you want to complete, you’ll likely have a bunch of abstract concepts that aren’t easy to understand without a whole lot of knowledge in the domain you’re working in. Give people a head start by providing a convenient, memorable name for the concept, using an analogy, or connecting it back to something they already understand. That way they’ll be able to quickly build their own mental models of what you’re talking about.

**Naming**

Two people can use the same words and mean quite different things. I joke that conversations with one of my favorite colleagues always devolve into us arguing about the meanings of words. But once we understand each other, we can speak in a very nuanced, high-bandwidth way and have a much more powerful conversation about where we actually agree or disagree.

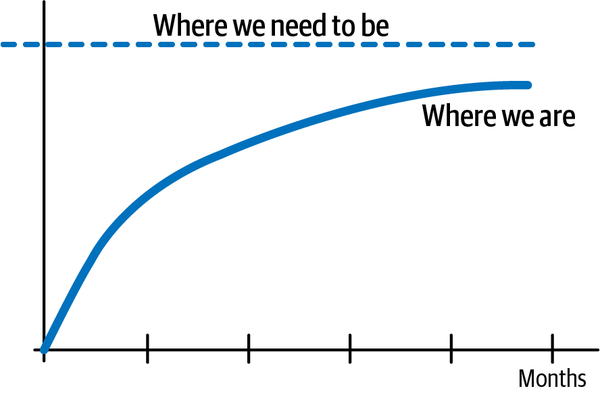
In 2003, Eric Evans wrote *Domain-Driven Design* (Addison Wesley) and gave us the concept of deliberately building what he called a [“ubiquitous language”](https://oreil.ly/CEQmR): a language shared by the developers of a system and the real-world domain experts who are its stakeholders. Inside a company, even very common words like *user*, *customer,* and *account* may have specific meanings, and those can even change depending on whether you’re talking to someone in finance, marketing, or engineering. Take the time to understand what words are meaningful to the people you intend to communicate with, and use their words when you can. If you’re trying to talk with multiple groups at once, provide a glossary, or at least be deliberate about describing what *you* mean by the terms you’re using.

**Pictures and graphs**

If you really want to reduce complexity, use pictures. There’s no easier way to help people visualize what you’re talking about. If something’s changing, a set of “before” and “after” pictures can be clearer than an entire essay. If one idea fits within another, you can draw them as nested; if they’re parallel concepts, they can be parallel shapes. If there’s a hierarchy, you might depict it as a ladder, a tree, or a pyramid. If you’re representing a human, using a stick figure or smiley-face emoji is clearer than just drawing a box.

Be aware of existing associations: don’t use a cylinder on your diagram unless you’re OK with many readers thinking of it as a datastore. If you use colors, some of your audience will try to interpret their meaning, for example assuming that green components are intended to be encouraged and red ones should be stopped.

Pictures can also take the form of graphs or charts. If you can show a goal and a line trending toward that goal (like in [Figure 5-2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#a_graph_can_show_progress_toward_a_goal)), it’s easy to see what success will look like. Similarly, if your line is trending toward some disaster point you’re highlighting, the need for the project can become viscerally clear.



**Figure 5-2. A graph can show progress toward a goal.**

**Designing**

Once the exploration is done and the work is clarified, you’ll probably have a lot of ideas for what happens next: what you’re going to build or change, and what approach you’re going to take. Don’t assume everyone you work with understands or agrees with those ideas. Even if you’re not hearing objections when you talk about them, your colleagues may not have internalized the plan and their implicit agreement may not mean anything. You’ll need to work to make sure everyone is aligned. The most efficient way to do that is to write things down.

**Why share designs?**

In [Chapter 2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch02.html#three_maps) I discussed oral versus written company cultures. The bigger the company, the more likely that your culture has shifted toward the latter, and that there is some expectation that you will write and review design documentation. That’s because it’s very difficult to have many people achieve something together without shared understanding, and it’s hard to be sure you have that shared understanding without a written plan. Whether you’re creating features, product plans, APIs, architecture, processes, configuration, or really anything else where multiple people need to have the same understanding, you won’t truly know if people understand and agree until you write it down.

Writing it down doesn’t mean you need a 20-page technology deep dive for every tiny change. A short, snappy, easy read can be perfect for getting a group on (literally) the same page. But you should at least include the important aspects of the plan, and let other people get in touch with you if they see hazards in your path. Asking for review on a design doesn’t just mean asking about the feasibility of an architecture or a series of steps; it includes agreeing on whether you’re solving the right problem at all and whether your assumptions about other teams and existing systems are correct. An idea that seems like an obvious path for one team may cause work for or break the workflows of another org. As my friend Cian Synnott says, [a written design is a very cheap iteration](https://oreil.ly/gp5f2).

**RFC templates**

A common approach to sharing information in this way is a design document, often called a request for comment document (or RFC). Although you’ll find RFCs used at many companies, there’s not really a consistent standard for what one should look like or how they get used. Different companies will have different levels of detail and formality, you’ll share more or less broadly, comments may or may not be encouraged, and you might have an official approval step or meeting to discuss the design.

I’m not going to weigh in on which process is best—it really depends on your culture—but I’m a big fan of having templates for documents like this.[**10**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn69) No matter how amazing we are as architects, there’s a lot to remember when designing a complex system or process or change. And humans aren’t great at paying attention to all of the things. As Atul Gawande, author of *The Checklist Manifesto: How to Get Things Right* (Picador), says:

*We are not built for discipline. We are built for novelty and excitement, not for careful attention to detail. Discipline is something we have to work at. It somehow feels beneath us to use a checklist, an embarrassment. It runs counter to deeply held beliefs about how the truly great among us—those we aspire to be—handle situations of high stakes and complexity. The truly great are daring. They improvise. They do not have protocols and checklists. Maybe our idea of heroism needs updating.*[***11***](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn70)

Gawande argues that using a checklist helps us talk to each other, avoid common mistakes, and make the right decisions *intentionally* instead of implicitly. A good RFC template helps you think through the decisions and reminds you of topics you might otherwise forget. Going through the exercise of creating this kind of document and answering some (perhaps uncomfortable) questions about your plan will help ensure you haven’t missed some vital category of problem.

**What goes in an RFC?**

Your company may already have its own RFC template, and you should follow that if it exists. However, here are the headings that I put in every RFC, and that I think should be included at an absolute minimum.

**Context**

I like documents to be anchored in space and time. When someone stumbles across this document in two years, the header should give them enough context to decide whether it’s relevant to whatever they’re searching for. It should have a title, the author’s name, and at least one date; I like “created on” and “last updated on,” but either of those is better than none. Include the status of the document: whether it’s an early idea, open for detailed review, superceded by another document, being implemented, completed, on hold. I like a standard format for headers so that scanning an RFC quickly is really easy, but it’s more important that the information is available than that it’s standardized.

**Goals**

The goals section should explain why you’re doing this at all: it should show what problem you’re trying to solve or what opportunity you’re trying to take advantage of. If there’s a product brief or product requirements document, this section could be a summary of that, with a link back. If the goal just suggests the question, “OK, but why are you doing *that*?” then you should go a step further and answer that question too. Provide enough information to let your readers know whether they think you’re solving the right problem. If they disagree, that’s great—you found out now and not after you built the wrong thing.

The goal shouldn’t include implementation details. If you send me an RFC with a goal of “Create a serverless API to translate the sounds of chickens,” I can absolutely believe that that’s what you’re trying to do, and I can review the RFC and try to appraise your design. But without knowing what actual problem you’re trying to solve and for whom, I can’t evaluate whether this is really the right approach. You’ve specified in the goal that you’re setting out to make it serverless, so you’ve already made a major design decision without justifying it. The specific implementation should *serve* the goal; it should not *be* the goal. Leave the design decisions to the design section.

**Design**

The design section lays out how you intend to achieve the goal. Make sure that you include enough information for your readers to evaluate whether your solution will work. Give your audience what they need to know. If you’re writing for potential users or product managers, make sure you’re clear about the functionality and interfaces you intend to give them. If you’ll depend on systems or components, include how you’ll want to use them, so readers can point out misunderstandings about their capabilities.

Your design section could be a couple of paragraphs or it could be 10 dense pages. It could be a narrative, a set of bullet points, a bunch of subsections with headers, or any other format that will clearly convey the information.

Depending on what you’re trying to do, the design section could include:

* APIs
* Pseudocode or code snippets
* Architectural diagrams
* Data models
* Wireframes or screenshots
* Steps in a process
* Mental models of how components fit together
* Organizational charts
* Vendor costs
* Dependencies on other systems

What matters is that at the end, your readers should understand what you intend to do and should be able to tell you whether they think it will work.

**WRONG IS BETTER THAN VAGUE**

I’ve often seen people be a little hand-wavy in writing this design section—or avoid committing to a plan at all—because they don’t want people to argue with them about the details. But it’s a better use of your time to be wrong or controversial than it is to be vague. If you’re wrong, people will tell you and you’ll learn something, and you can change direction if you need to. If you’re trying out a controversial idea, you can find out early whether your colleagues will pull against your approach. Having disagreements about your design doesn’t mean that you need to change course, but it gives you information you wouldn’t have had otherwise.

Here are two tips to make your design more precise:

* Be clear about who or what is doing the action for every single verb. If you find yourself writing in the passive voice, like “The data will be encrypted in transit” or “The JSON payload will be unpacked,” then you’re obscuring information and making the reader guess. Instead, write with active verbs that have a subject who does the action: “The client will encrypt the data before it is transmitted” or “The Parse component will unpack the JSON payload.”[**12**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn71)
* Here’s a tip that was a game-changer for me: It’s fine to use a few extra words or even repeat yourself if it means avoiding ambiguity. As software engineer and writer Eva Parish recommends in her post [“What I Think About When I Edit”](https://oreil.ly/TExXb):

*Instead of saying “this” or “that,” you should add a noun to spell out exactly what you’re referring to, even if you’ve just mentioned it.*

*Example: We only have two boxes left. To solve this, we should order more.*

*Revision: We only have two boxes left. To solve this shortage, we should order more.*

Since I read Eva’s article, I notice so many examples where a bare “this” or “that” in a design document obscures information. For example, “A proposal exists to replace OldSolution, which was built to provide OriginalFunctionality, with NewSolution. TeamB needs this, so we should discuss requirements.” What does TeamB need: the proposal, the original functionality, or the new solution?

If you struggle with writing, bear in mind that it’s a learnable skill. Any learning platform your company has access to will probably have a technical writing class on offer. Or consider Google’s courses, [Technical Writing One and Technical Writing Two](https://oreil.ly/DUTL2). The [Write the Docs](https://oreil.ly/g535C) website also offers a ton of resources on how to write well.

**Security/privacy/compliance**

What do you have worth protecting, and from whom are you protecting it? Does your plan touch or collect user data in any way? Does it open up new access points to the outside world? How are you storing any keys or passwords you’re going to use? Are you protecting against insider or external threats, or both? Even if you think there are no security concerns and believe this section isn’t relevant, write down why you believe that is the case.

**Alternatives considered/prior art**

If you could solve this same problem with spreadsheets, would you still want to do it?[**13**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn72) The “alternatives considered” section is where you demonstrate (to yourself and others!) that you’re here to solve the problem and you aren’t just excited about the solution. If you find yourself omitting this section because you *didn’t* consider any alternatives, that’s a signal that you may not have thought the problem through. Why *wouldn’t* simpler solutions or off-the-shelf products work? Has anyone else at your company ever tried something similar, and why isn’t their solution a good fit? I have a policy that if a plausible-seeming option already exists inside the company and we’re not going to use it, the RFC author *has to* send the new design to the people who own that system and give them an opportunity to respond.

Those are the headings that I think absolutely have to be included, even on a tiny RFC. They’re the “keep you honest” sections! But there are some others that are often helpful if you want to get the most value out of the document you’re writing.

**Background**

What’s going on here? What information does a reader need to evaluate this design? You could include a glossary if you’re using internal project names, acronyms, or niche technology terms that reviewers might not know.

**Trade-offs**

What are the disadvantages of your design? What trade-offs are you intentionally making because you think the downsides are worth the benefits?

**Risks**

What could go wrong? What’s the worst that could happen? If you’re a bit nervous about system complexity, added latency, or the team’s lack of experience with a technology, don’t hide that concern: warn your reviewers and give them enough information to draw their own conclusions about it.

**Dependencies**

What are you expecting from other teams? If you’re going to need another team to provision infrastructure or write code, or if you need security, legal, or comms to approve your project, how much time will you need to allow them? Do they know you’re coming?

**Operations**

If you’re writing a new system, who will run it? How will you monitor it? If it will need backups or disaster recovery tests, who will be responsible for those?

**Technical pitfalls**

While this is not intended to be a technical or architectural book, I do want to call out a few pitfalls I often see in design documentation. Catch them for yourself so other people don’t have to.

**It’s a brand-new problem (but it isn’t)**

There are occasional exceptions, but your problem is almost certainly not brand new. I already talked about looking for prior and related projects, but it’s important enough to mention again here. Don’t miss the opportunity to learn from other people, and consider reusing existing solutions.

**This looks easy!**

Some projects are seductively harder than they look, and you might not realize that until you’re deep in the weeds of implementing them. Software engineers don’t always really internalize that other domains are as rich and nuanced and complex as their own. They might see, say, an accounting system and assume they can build a better, cleaner, simpler one. How could previous teams have put thousands of engineer-hours into *this*!? But building an accounting system (or a payroll system, or a recruitment system, or even something to correctly share on-call schedules) is actually a hard problem. If it seems trivial, it’s because you don’t understand it.

**Building for the present**

If you’re building for the state of the world as it is right now, will your solution still work in three years? Even if you’re designing for five times the current number of users and requests, there are other dimensions to think about. If the system needs to know about all teams or all products at your company, what happens as the company grows, or has acquisitions, or is acquired? If you have five times as many products, will this component become a bottleneck as everyone waits for one team to add custom logic for them? If your team doubles in size, will its members still be able to work in this codebase?

**Building for the distant, distant future**

If you’re designing for a few orders of magnitude more than your current usage, do you have a real reason to go that big? If it’s trivial to handle more users, that’s great, do it, but watch out for overengineered solutions that are much more complicated than they need to be. If you’re adding custom load balancing, extra caching, or automatic region failover, explain why it’s worth the extra time and effort. “We might need it later” is not a good enough justification.[**14**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn73)

**Every user just needs to…**

If you have five users, you can probably individually teach each of them all the arcane rules of your system. If you have hundreds, or more, they’re going to do it wrong; if you don’t plan for that, your design doesn’t work. Any part of your solution that involves humans changing their workflows or behavior will be difficult and needs to be part of the design.

**We’ll figure out the difficult part later**

This one is common in migrations: you spend a quarter building and deploying the system, polishing it up, and making it perfect for a couple of easy use cases—and then you have to figure out how to make it work for more difficult cases. What happens if that turns out not to be possible? Ignoring the difficult part of your project might also mean pushing complexity to someone else, like requiring every existing caller of an API to change their code rather than being backward compatible, or forcing your clients to write their own logic to interpret arcane and scattered information.

**Solving the small problem by making the big problem more difficult**

If you have lots of tiny projects with barely enough staffing to scrape by, you’ll see people working around difficult problems in hacky ways instead of engaging with them directly. These tacked-on solutions often have hidden dependencies on existing system behavior that mean it will be harder to implement a more comprehensive solution later. If your organization refuses to invest in solving the underlying problem, you may not have any choice, but at least call it out in your design. Think about how you can solve the smaller problem without making the bigger one less tractable.

**It’s not really a rewrite (but it is!)**

If you’re looking at a huge software system and envisioning it in a different shape, be honest with yourself and others about how much work that will take. You might imagine “just” taking the business logic and refactoring it, for example, or rearchitecting it for the cloud. But unless your code is already very modular and well organized (in which case, are you sure you need to rearchitect?), chances are you’ll end up rewriting a lot more than you intended. If your project is a veiled “rewrite from scratch,” be honest with yourself and admit it.

**But is it operable?**

If you struggle to remember how something works at 3 p.m., you won’t understand it at 3 a.m. And the people who join your team after you’ve moved on will find it much harder. Make sure you create something that other people can reason about. Aim to make systems observable and debuggable. Make your processes as boring and as self-documenting as possible.

Speaking of operability, if it’s going to run in production, decide who’s on call for it and put that in the RFC. If that’s your own team, make sure you have more than three people (ideally at least six) or you’ll be setting yourselves up for burnout and dropped pages.

**Discussing the smallest decisions the most**

Who doesn’t love a good bikeshed discussion! The expression “bikeshedding” came from C. Northcote Parkinson’s 1957 [“Law of Triviality”](https://oreil.ly/D8nvw), which holds that since it’s much easier to discuss a trivial issue than a difficult one, that’s where teams tend to spend their time.[**15**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn74) Parkinson’s example was a fictional committee evaluating the plans for a nuclear power plant but spending the majority of its time on the easiest topics to grasp, like what materials to use for the staff bicycle shed.[**16**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn75) Tech people are usually aware of the concept of bikeshedding, but even senior people drift into writing long paragraphs about the most trivial, reversible decisions, while not engaging at all with the ones that are harder to grasp or to find consensus on.

These are just a few of the common pitfalls. You’ve probably noticed others. Add those to your list and be really sure you’re not falling prey to them yourself.

**Coding**

Most software projects will involve writing a lot of new code or changing existing code. In this section, I’ll talk about how a project lead can engage with this kind of hands-on technical work. (If you’re not a software engineer, swap in whatever core technical work makes sense for you here.)

**Should you code on the project?**

As the project lead, how much code you contribute will vary depending on the size of the project, the size of the team, and your own preferences. If you’re on a tiny team, you might be deep in the weeds of every change. On a project with multiple teams, you might contribute occasional features, or just small fixes, or you might work at a higher level and not code at all. Many project leads find that they review a lot of code, but don’t write much themselves.

As Joy Ebertz [points out](https://oreil.ly/mPHXC), “Writing code is rarely the highest leverage thing you can spend your time on. Most of the code I write today could be written by someone much more junior.” Ebertz notes, however, that coding gives you a depth of understanding that’s hard to gain otherwise and helps you spot problems. What’s more, “If spending a day a week coding keeps you engaged and excited to come to work, you will likely do better in the rest of your job.” Finally, staying involved in the implementation ensures that you feel the cost of your own architectural decisions as much as your team does.

Notice, though, if you’re contributing code at the expense of more difficult, more important things. This is a form of the *snacking* I mentioned in [Chapter 4](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch04.html#finite_time-id00004): taking on work that you know how to do (and that has a shorter feedback loop) and avoiding the big, difficult design decisions or crucial organizational maneuvering.

**Be an exemplar, but not a bottleneck**

As the person responsible for moving the project along, your time is going to be less predictable than other people’s. You’ll probably have more meetings than everyone else does too. So if you take on the biggest, most important problems, chances are you’ll take longer to get to coding work than someone else would, which can block others and make you a bottleneck. If you’re coding, try to pick work that’s not time-sensitive or on the critical path.

Think of your code as a lever to help everyone else. Katrina Owen, coauthor of [*99 Bottles of OOP*](https://oreil.ly/kEpkE) and a staff engineer at GitHub, told me about a project where she created a standard way of writing a test for API pagination, then replaced all of the existing tests with her approach. By changing all of the current pagination tests, she was implicitly improving future tests too: anyone creating one would copy the pattern that was already there.

Aim for your solutions to empower your team, not to take over from them. Ross Donaldson, a staff engineer working on database systems, has described part of his work to me as “scouting and cartography”:

*I come back to the team and say, “I found this problem, that river, these resources,” then we can all together discuss how we want to approach this new information. Then maybe I go out and build a rough bridge over the new river, which the team will own and improve. I provide an opinion or two and remind people of some of the tools they have at their disposal, but otherwise prioritize their sense of ownership over my own sense of code aesthetic.*

Polina Giralt, senior staff engineer at Squarespace, adds,

*If there’s something only I understand, I’ll do it, but insist that someone pairs up with me. Or if it’s an emergency and I know how to fix it, I’ll do it myself and explain it later. Or I’ll write the code to establish a new pattern, but then hand it off to someone else to continue implementing it. That way it forces knowledge sharing.*

Rather than taking on every important change yourself, find opportunities for other people to grow, by chatting over the details or pair programming on the change. Pairing shares knowledge and builds other people’s skills. Pairing also means you can dip in for the key part of a change, then leave your colleague to complete the work.

If you’re reviewing code and changes, be aware of how your comments are received. Even if you think you’re relatable and friendly, it can be intimidating for early-career engineers when a staff engineer comments on their work. You want the rest of the team to think of you as a resource to learn from, not as someone who criticizes every decision and makes them feel inadequate. Sometimes it’s better to let someone set a pattern that’s *good enough* and not overrule them, even if you would have done it better. Also, be careful that you’re not doing *all* the code reviews—or you’ll be a single point of failure and the rest of your team won’t learn as much.

A staff+ engineer is an exemplar in another, more implicit way: *whatever you do will set expectations for the team*. For that reason, it’s important to produce good work. Meet or exceed your testing standards, add useful comments and documentation, and be very careful about taking shortcuts. If you’re the most senior person on the team and you’re sloppy, you’re going to have a sloppy team. (I’ll talk more about being a role model in [Chapter 7](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#youapostrophere_a_role_model_now_left_p).)

**Communicating**

Communicating well is key for delivering a project in time. If the teams aren’t talking to each other, you won’t make progress. And if you’re not talking to people outside the project, your stakeholders won’t know how it’s going. Let’s look at both of these types of communication.

**Talking to each other**

Find opportunities for your team members to talk with each other regularly and build relationships, even (or maybe especially) on fully remote teams. It should feel easy to reach out and ask questions, and they should be comfortable enough with each other that they can disagree without it getting tense. You can make relationship building easier with shared meetings, friendly Slack channels, demos, and social events. If you have a small number of teams, key people from adjacent teams can attend each other’s meetings or standups. The goal is comfortable familiarity.

Familiarity will make it feel safer to ask clarifying questions too. Engineers who don’t know each other may feel uncomfortable saying “I don’t know what that term means” or “What are the implications of the problem you just described?” It will be harder to work together, share knowledge, and uncover misunderstandings as a result. Aim to get to a place where it’s normal for your team to ask questions and admit what they don’t know.

**Sharing status**

Your project has other people who care about it: stakeholders, sponsors, customer teams who are waiting for you to be done. Make it easy for them to find out what’s going on, and set their expectations about when you’ll reach various milestones. That might mean one-on-one conversations, regular group email updates, or a project dashboard with statuses.

As you understand the progress of your project, you’ll probably pick up a lot of detail and nuance about who’s doing what, what’s intended to happen when, and how each part of the project is going. When you’re delivering status updates, you might feel inclined to share everything you know: more information is better, right? Not necessarily! Too much detail can obscure the overall message and make it harder for your audience to take away the conclusion you intended.

Instead, explain your status in terms of impact and think about what the audience will actually *want* to know. They probably don’t care that you stood up three microservices; they care about what users can do now, and when they’ll be able to do the next thing. If it’s becoming clear that you’re not going to hit a key milestone date, that’s a fact you’ll want to pass on. But if one team is delayed in a way that doesn’t change your delivery date, that delay may not be relevant to them. Calling it out may even look like you’re trying to escalate something that doesn’t need escalation.

If you think your audience really will want all the details, at least lead with the headlines. Don’t assume that they’ll sift through your update for the key facts or read between the lines to pick up nuances. If something is not clear, spell out the takeaways. Practice following facts with “That means…” or explaining that you’re doing something “so that we can…”

Be realistic and honest about the status you’re reporting. If your project is having difficulties, it may be tempting to put on a brave face, hope you’ll sort everything out, and report that the status is green. When you do this, though, you risk an unpleasant surprise at the end of the project when you have to admit that it’s not and hasn’t been for a while. Have you ever heard someone talk about a “watermelon” project? They’re all green on the outside, but the inside is red. If your project is stuck, don’t hide it: ask for help.

**Navigating**

Something will always go wrong. Maybe you realize that a technology that’s been core to your plans isn’t going to be a good fit after all: it won’t scale, it’s missing some table-stakes feature, or it’s got a licensing condition your legal team has absolutely vetoed. Maybe your organization announces a change in business direction and now needs you to solve a different problem. Maybe someone vital to the project quits. It is inevitable that you’ll meet some roadblocks and have to change direction, and you’ll have a better time if you go into the project assuming *something* is going to go wrong—you just don’t know yet what it will be. This attitude can help you be more flexible, so you’ll find it sort of interesting when the roadblock arrives, rather than being frustrated by it. Reframe these diversions as an opportunity to learn and to have an experience you wouldn’t have had otherwise.

As the person at the wheel, you’re accountable for what happens when your project meets an obstacle. You don’t get to say, “Well, the project is blocked and so there’s nothing we can do”: you are responsible for rerouting, escalating to someone who can help, or, if you really have to, breaking the news to your stakeholders that the goal is now unreachable. Avoid those “watermelon projects”: if the project status is green apart from one key problem that’s going to be impossible to solve, the project status is not really green!

Whatever the disruption, work with your team to figure out how you can navigate around it. The bigger the change, the more likely it is that you’ll need to go back to the start of this chapter, deal with the overwhelm all over again, build context, and treat the project like it’s restarting. Whatever happens, make sure you keep communicating well about it. Don’t create panic or let rumors start. Instead, give people facts and be clear about what you want them to do with the information.

When you’re having difficulties, remember that you are not the only person who wants your project to succeed.[**17**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn76) Your manager’s job is to make you successful, and your director’s job is to make your organization successful. If you’re not telling them you need help, it’s going to be harder for them to do their jobs. Some people really resist asking for help. It feels like failure, maybe. But if you’re stuck and need help, the biggest failure is not asking for it. Don’t struggle alone.

I’ll talk about navigating obstacles a lot more in the next chapter, when we look at some of the reasons that projects can get stuck and how to get them back on track.

**To Recap**

* Staff engineers can take on problems that seem intractable and make them tractable.
* It’s normal to feel overwhelmed by a huge project. The project is difficult. That’s why it needs someone like you on it.
* Set up the structures that will reduce ambiguity and make it easy to share context.
* Be clear on what success on the project will look like and how you’ll measure it.
* Leading a project means deliberately *driving* it, not just letting things happen.
* Smooth your path by building relationships and deliberately setting out to build trust.
* Write things down. Be clear and opinionated. Wrong gets corrected, vague sticks around.
* There will always be trade-offs. Be clear what you’re optimizing for when you make decisions.
* Communicate frequently with your audience in mind.
* Expect problems to arise. Make plans that assume there will be changes in direction, people quitting, and unavailable dependencies.

[**1**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn60-marker) I’m going to say “project manager” throughout this chapter, but you might work with a program manager (usually someone who’s responsible for multiple projects that have a shared goal). Sometimes you’ll see the title technical program manager (TPM). For the purposes of this chapter, just assume I’m talking about someone in any of these roles, and that they’re preternaturally organized and competent at delivering products.

[**2**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn61-marker) If you skipped [Chapter 2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch02.html#three_maps), just imagine teams and organizations as tectonic plates moving against each other, with friction and instability where the plates meet.

[**3**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn62-marker) Check in on your biological needs too. Not trying to get in your business, but lots of people in our industry are sleep-deprived. Are you? Sleep builds resilience and willpower and energy! It’s amazing. And when did you last drink a glass of water?

[**4**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn63-marker) Although it’s almost certainly apocryphal, there’s a Henry Ford quote about the Model T that illustrates how people in different domains can fail to communicate: “If I had asked people what they wanted, they would have said faster horses.”

[**5**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn64-marker) It probably goes without saying, but be diplomatic when you describe this reality. If you write down the most charitable description of the difficult person, antagonistic team, or indecisive director, you’ll feel less awkward when someone inevitably forwards the email or document to them. You’ll build empathy for them too and perhaps have some insights about how to work with them.

[**6**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn65-marker) Though think how much better our industry’s internal solutions would be if they did.

[**7**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn66-marker) Though try to avoid relying on one person’s very specific skill set. People move between companies often, and you don’t want a single point of failure.

[**8**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn67-marker) It’s likely there are at least some people who are outside the office, and not uncommon for everyone on the team to be in a different city or time zone! If everyone is distributed around the globe, it’s best if they have a good amount of overlap in their work days, especially if they will need to work together closely. It’s hard to build a trusting relationship on completely asynchronous communication.

[**9**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn68-marker) Be clear about this point throughout the project! It’s common for new leaders to sort of hint that it might be nice if everyone did something you need them to do. Think of it this way: if you’re ambiguous, you’re making *more* work for everyone else as they try to figure out how much the thing you’ve asked for matters. Be explicit about what you want people to do.

[**10**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn69-marker) I wrote a blog post for the Squarespace engineering blog that includes a sample template: [“The Power of ‘Yes, if’: Iterating on Our RFC Process”](https://oreil.ly/Pdb3l).

[**11**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn70-marker) He goes on to show that checklists save lives. Few people reading this will be responsible for life-critical systems, but if you are, please have protocols and checklists!

[**12**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn71-marker) Dr. Rebecca Johnson [offers](https://oreil.ly/zSRhh) the best test I’ve ever read for accidental use of the passive voice: “If you can insert ‘by zombies’ after the verb, you have passive voice.” It almost always works. “The data will be encrypted in transit [by zombies].” Thanks, zombies!

[**13**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn72-marker) If you love spreadsheets and this just makes the project more attractive to you, sub in whatever technology you find most mundane.

[**14**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn73-marker) One of the most expensive phrases ever uttered in software engineering.

[**15**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn74-marker) Parkinson also [coined](https://oreil.ly/AE5w5) the law that “work expands so as to fill the time available for its completion.” The dude was insightful!

[**16**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn75-marker) I wrote a talk once where someone left no comments on the entire 83-slide deck except to suggest a replacement for the bikeshed picture I’d included on one of the slides. True story! I don’t think they were being ironic.

[**17**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#ch01fn76-marker) Unless you’re really spending your social capital on a passion project that nobody else cares about, as discussed in [Chapter 4](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch04.html#finite_time-id00004)! In that case, sorry, you’re probably on your own. But I hope you’ve got enough goodwill that your colleagues and manager are still sort of enthusiastic about it on your behalf and are willing to help you get it unstuck.