**Chapter 7. You’re a Role Model Now (Sorry)**

“Don’t think out loud,” my friend Carla Geisser warned me when I became a staff engineer. “You’ll find out a month later that people are talking about your half-baked idea like it’s already a project.” My colleague Ross Donaldson described his own role even more starkly: “Being staff doesn’t absolve you of being wrong, but it does mean you need to be careful when you open your dang mouth.”

This is the blessing and the curse of a staff engineer title: people will assume you know what you’re talking about—so you’d better know what you’re talking about! Your work will be a little less checked and your ideas considered more credible. Rather than guiding you, people will look to you for guidance.

Most of all, you’ll be a role model. How you behave is how others will behave. You’ll be the voice of reason, the “adult in the room.” There will be times when you’ll think “This is a problem and someone should say something”…and realize with a sinking feeling that that someone is *you*. The behavior you model will show your less experienced colleagues how to be a good engineer. Later, in [Chapter 8](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#good_influence_at_scale), we’ll look at how to *actively*, deliberately influence your organization and colleagues for the better. But this chapter is about *passive* influence, the kind that you have just by the way you act as an engineer and as a person.

**What Does It Mean to Do a Good Job?**

Your company might have a written definition of what good engineering means: written values, perhaps, or engineering principles. But values are what you do: the clearest indicator of what the company values is what gets people promoted. No matter how much your organization claims to encourage collaboration and teamwork, that message will be undermined if any staff engineers get to that level through “heroic” solitary efforts. If your engineering principles describe a culture of thorough code reviews, but senior engineers approve PRs without reading them, everyone else will rubber-stamp code reviews too. The work that you do is implicitly the type and standard of work that others will see as correct and emulate.

Engineering goes beyond what you do when you’re talking to computer systems; it’s also about how you talk to humans. So sometimes being a good engineer boils down to being a good colleague. If you’re mature, constructive, and accountable, you’re telling your new grads that’s what a senior engineer does. If you’re condescending, impossible to please, or never available, *that’s* what a senior engineer does, too. You shape your company every day, just by how you behave.

**But I Don’t Want to Be a Role Model!**

Being a role model is not always comfortable. But as you become more senior, it’s one of the biggest ways you’ll affect your organization. Like it or not, you’re setting your engineering culture. Take that power seriously. Being a role model doesn’t mean you have to become a public figure, be louder than you’re comfortable with, or throw your weight around. Many of the best leaders are quiet and thoughtful, influencing through good decisions and effective collaboration (and showing fellow quiet people that there’s space for them to lead).

If the idea of being a leader is terrifying, you may need to build up to it. Start small. Maybe compliment someone’s success on a public channel, or offer to help onboard a new person. Think of leadership as a skill to build, just like you would learn a new language or technology. The more you practice, the easier it will be.

Be the best engineer and the best colleague that you can be. Do a good job and let others see it. (And help others do the same! We’ll discuss how in [Chapter 8](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#good_influence_at_scale).) That’s what being a role model is.

**What Does It Mean to Do a Good Job as a Senior Engineer?**

In this chapter, I’m going to spell out the four big attributes that I think you should be modeling. Let’s be clear: these are aspirational qualities, skills you should strive to learn and keep learning. I’m explicitly *not* saying that you need to score 100% on each of these attributes to be a “real engineer,” or any form of gatekeeping like that; these are ideals. This is how you should *try to be*. We’re all works in progress.

One more caveat: the tech industry is awash in advice, most of it subjective. This list is too! Best practices all depend on the situation. There will be edge cases and special circumstances; if my advice contradicts your own judgment, trust yourself. A staff engineer has the good sense to know when the conventional wisdom is wrong.

The four attributes we’re going to look at for the rest of the chapter are being competent, being a responsible adult, remembering the goal, and looking ahead.

First up, competence.

**Be Competent**

As a staff+ engineer, or any senior person, a big part of your role is to take on things that need to be done and to reliably do them well. Competence includes building (and keeping) knowledge and skills, being self-aware, and having high standards.

**Know Things**

No matter how good your leadership, you can’t be a *technical* leader without the “technical” part. Your big-picture thinking, project execution, credibility, and influence are underpinned by knowledge and experience. A big part of the value proposition of hiring you is your knowledge: you have *seen some things*.

**Build experience**

Stephanie Van Dyk, staff engineer at Google, draws parallels between the foundational skills needed for her job and those she uses in her longtime hobby as a weaver. “Technical skills come from study and practice,” she says. “They aren’t inherent. No one is born a skilled weaver; no one is born a skilled computer engineer.”

Experience comes through time, exposure, and study—not innate aptitude. It takes work to develop technical skills. You can learn a lot from books, but there’s no substitute for working through problems yourself, learning your own techniques for solving them, and seeing what works and what doesn’t. Paula Muldoon, a senior software engineer and violinist, describes playing in an orchestra in a way that resonates for me: “You want to get so good at your craft that your focus can be almost entirely on what other people are doing and you don’t have to worry about your own execution.” Invest time—lots of time—in honing your technical skills so that they become second nature to you.

How much time? The American Society of Civil Engineers [publishes its engineering grades](https://www.asce.org/engineergrades), and its requirements include number of years of experience:

* Grade V (Typical titles: Senior engineer, program manager): 8+ years
* Grade VI (Typical: Principal engineer, district engineer, engineering manager): 10+ years
* Grade VII (Typical: Director, city engineer, division engineer): 15+ years
* Grade VIII (Typical: Bureau engineer, director of public works): 20+ years

We’re less rigorous in software engineering. Job levels vary a lot across companies. Most places don’t consider years of experience when allocating grades or titles, but staff engineers typically have at least 10 and principal engineers have at least 15.

Don’t rush past your prime learning years. Some organizations encourage their best talent by offering them senior roles, like management or staff engineering, relatively early in their careers. The push for career progression may entice you to accept. But, as [Charity Majors warns](https://oreil.ly/vmIuH):

*Never, ever accept a managerial role until you are already solidly senior as an engineer. To me this means at least seven years or more writing and shipping code; definitely, absolutely no less than five. It may feel like a compliment when someone offers you the job of manager—hell, take the compliment —but they are not doing you any favors when it comes to your career or your ability to be effective.*

It’s the same for staff+ roles. Do a whole lot of soul-searching before taking a role that takes you further from the tech. You could be cheating yourself out of your prime fully immersive hands-on experience-building years.

**ARE YOU “TECHNICAL ENOUGH”?**

I’ve sometimes heard engineers describe other people as “not technical enough,” and it’s a framing I always push back on. Apart from being a little dismissive, it claims to describe *who someone is* instead of *the skills they have*.[**1**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn87) It leaves no actionable path to *becoming* “technical enough.” If you’re inclined to describe someone in this way, be more precise. Do you mean:

* They haven’t yet spent enough years doing hands-on technical work?
* They don’t yet know the domain you’re working in?
* They’re missing specific skills? Which skills?

Being competent and knowledgeable doesn’t mean you have to know the most about every topic. Sometimes, when you come into a new domain, you will know the least, and that’s OK!

**Build domain knowledge**

Software has an extraordinary number of technology areas, each with its own specialized knowledge and vocabulary. Knowing mobile development, algorithmic computer science, or networking doesn’t prepare you for a frontend UX project. Years of experience in fintech won’t prepare you for a health care startup. But if you’re interested in a new technology area or domain, you can still go try it out. That means that even very experienced engineers can find themselves being beginners.

When you move into a new role, there will inevitably be skills that you don’t have yet, or domain knowledge that will be new to you. You may find that you’re learning from the junior engineers you work with. (This is a good thing!) Here is where your technical knowledge provides a foundation. While you might not recognize the specifics of the problems in this new domain, your general experience should help you recognize their *shapes*. You should be able to pattern-match what you’re encountering to something else you recognize, so you’re not completely at sea.[**2**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn88) The broader your scope of experience, the more “hooks” you’ll have to hang new knowledge off, and the faster you can learn new things.

When you move into a new technology area or business domain, be deliberate about learning quickly. Learn the technology in whatever way is most effective for you. Get a sense of the appropriate trade-offs; the resource constraints; the common arguments, biases, and in-jokes. Know which technologies are on the market and how they might be used. Read the core literature. Know who the “famous people” in the domain are and what they advocate for. (Twitter is handy for this.) Then take on some projects that will let you build instincts and experience, so you can become as competent in this new area or domain as you were in your last.

**Stay up to date**

Being a senior engineer means having a growth mindset and a drive to improve. It’s embarrassing for everyone when a technical leader insists on a “best practice” that has been debunked for a decade or a technology that everyone else has moved on from. Stay engaged with what’s happening in your part of the industry. Even if you aren’t deep in the code any more, your spidey-sense should stay sharp for “code smells” or problems waiting to happen. Even if you don’t know the latest, hottest tool or practice, know how to find out.

Notice if your role is preventing you from continuing to learn. In particular, be wary of drifting so far from the technology that you’re only learning how *your company* operates. While you should *also* learn enough about the business to make good choices, keep yourself anchored in the tech. I’ll talk more about learning in [Chapter 9](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch09.html#whatapostrophes_nextquestion_mark).

**SHOW THAT YOU’RE LEARNING**

Junior engineers need to see that lifelong learning is part of being a senior engineer, and that they’ll never reach the end of their journey. They also need to see that your skills and knowledge didn’t magically come to you—it’s all learnable. Be open about what you’re learning, and show how you’re doing it. When you’re making a statement that involves obscure knowledge or a logical leap, fill in the gaps for the folks around you: tell them why you came to the conclusion you did, what information you used, and how you came by that information. Be clear that you’re learning so it’s safe for them to learn too.

**Be Self-Aware**

Competence is built on knowledge and experience, but you also need to be able to apply those abilities. That starts with having the self-awareness to know what you can do, how long it will take, and what you *don’t* know*.* It means being able to say “I’ve got this” and knowing that you do, in fact, have this. Competence means having well-founded confidence that you’ll be able to solve the problem. You don’t need to be arrogant, speak in incomprehensible jargon, or show off. True confidence comes from having done the work for long enough that you’ve learned to trust yourself.

**Admit what you know**

Some people are brought up to brag about their accomplishments. Others are brought up to minimize them. Whichever you innately are, aim to get to a level where you’re confident and honest with yourself about what you know and what you don’t. There are going to be some areas where you know a lot and are unusually skilled. Be confident about applying those skills to solve the problems that need them.

Being competent doesn’t mean you need to be *the best.* I’ve sometimes seen tech people be shy about claiming to be an expert, because they can always think of *someone* in the industry who is better than they are. Don’t set your bar at “best in the industry.” It won’t help anyone if you hold back out of modesty. When the skill is needed, don’t wait for someone else to say, “Hey, aren’t you [great at regular expressions”](https://xkcd.com/208)? Volunteer that you are—it’s not a brag, just a statement of fact.

If you know what skills you bring, then you know where you can step up and help, where you’ll be a good mentor, and what you still need to learn.

**Admit what you don’t know**

You won’t know everything, and it’s vital that you don’t pretend you do. If you bluff, you’ll lose opportunities to learn—and you may make bad decisions. You’ll also waste the opportunity to set an example. Every time you admit you don’t know everything and let people see you learning, you show your junior engineers that it’s normal to continuously learn.

Admitting ignorance is one of the most important things we can do as tech leads, senior engineers, mentors, managers, and other influencers of team culture. I love asking for an “ELI5,” a term that comes from Reddit and means “Explain it like I’m five years old.” It’s a helpful shortcut to mean “Look, rather than guessing my level of understanding, just spell it out for me. I promise not to be offended if you tell me things I already know.” (The social contract here is that you can’t get offended if they start with the very basics of the topic.)

We spend a huge amount of our work lives communicating, trying to get a shared model of the world into our brains so that we can collaborate or make decisions. It slows everything down when people in a conversation bluff or disengage a bit because they don’t want to be called out on not knowing something. If senior people can admit they don’t know things, everyone else will do it too.

**Understand your own context**

A huge part of self-awareness is understanding that you have a perspective, that your context is not the universal context, and that your opinions and knowledge are specific to *you*. You’ll need to escape your echo chamber every time you talk to teams in other areas or explain technical topics to nonengineers. You’ll know what information you have that they might not, so you can bridge that gap. (You can read more about building this kind of perspective in [Chapter 2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch02.html#three_maps).)

It’s much harder to explain something simply! It requires more understanding of the topic and more self-awareness about your own context. But it’s a real indicator of expertise. If you can explain a topic in plain language, so that nonexperts can hook it onto something they already understand, you really understand it.

**Have High Standards**

Your standards will serve as a model for how other people work. Know what high-quality work looks like and aim for that standard in everything you do, not just the parts you enjoy most. Write the clearest documentation you can. Be the first person to know if your software breaks. There are always trade-offs, of course: sometimes the right move is to slap a solution together with duct tape as quickly as possible. But make that determination based on the problem you’re trying to solve, not how interesting the work is to you.

**Seek out constructive criticism**

Having high standards means making your work as good as it can be. Look for opportunities to put aside your ego and ask someone else to help make your work better. Ask for code review, design review, and peer evaluations. When you’ve got an idea you love, invite your colleagues to poke holes in it. When you “request comments,” don’t secretly resent them; each one is an opportunity to make your solution better, so take them seriously even if you don’t use them all. Your solutions are not you and they don’t define you. Criticism of your work isn’t criticism of you. (You’ll *give* constructive criticism too, of course. We’ll explore that in [Chapter 8](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#good_influence_at_scale).)

**Own your mistakes**

At some point you *will* make a mistake, and it might be a big one. Maybe you reviewed code and didn’t notice a bug that cost the company a ton of money. Maybe you wrote that code! Maybe you said something in a meeting that you later found out made someone cry (or quit).

Mistakes are normal.[**3**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn89) Humans aren’t perfect, and mistakes are how we learn. What matters most is how you respond to your mistakes. It’s easy to get defensive, deflect blame, or fall to pieces (that someone else needs to pick up). To be competent, you need to own your mistakes. Don’t beat yourself up, but don’t deny the impact or insist that the mistake wasn’t *really* your fault. Admit what happened, then set out to fix it. Communicate quickly and clearly and make sure everyone has the information they need. If there’s any risk that someone else will get blamed, clarify that they didn’t do anything wrong. If you hurt someone else’s feelings, acknowledge the hurt and apologize. (Even if *you* wouldn’t have felt bad in the same situation, their feelings are real.)

If you caused an outage or took an expensive wrong path, consider having a retrospective afterward where you talk through what happened, how you recovered, and what you learned. Be open and matter-of-fact about the part you played. It’s much easier to understand what happened when nobody’s trying to downplay their missteps.

Making a mistake just *stings*. Solving the problem you caused may be the last thing you want to do in that moment. But it’s the best thing you can do to retain the goodwill and social capital of your team. If you react well and fix the problem you caused, you could even end up with *more* esteem from your colleagues. And a leader being open about their mistakes will make it easier for everyone else to do the same: it’s a big boost to the team’s psychological safety.

**Be reliable**

My final thought on competence is this: be reliable. One of the biggest compliments I give is, “Alex is going to be in that meeting, so I don’t need to go.” When I say that, I’m not just saying that any information I have will be represented in the meeting. I’m also saying that I think the right thing will happen. The situation will be managed. I don’t need to be there. I’m saying that I find Alex reliable.

A reputation for reliability is like the credibility and social capital we talked about in [Chapter 4](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch04.html#finite_time-id00004): it builds up as people see you do the work and get things under control. Be the sort of person who is trusted to get it done well.

Part of reliability is also finishing what you start. Use the techniques in [Chapter 6](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#why_have_we_stoppedquestion_mark) to make sure you’re not blocked or stopping too early. Stick with it even after it gets boring or difficult. And if you stop deliberately because the project isn’t the right use of resources, own and communicate that decision. You accepted responsibility for the work, so take it to the finish line.

That brings us to the second attribute senior engineers should strive for: being the responsible person in the room.

**Be Responsible**

Like it or not, a senior or staff title turns you into an authority figure—and, as the philosopher Uncle Ben once told Spider-Man, with great power comes great responsibility. The more senior you get, the more you have to internalize that nobody else is coming to be the “grown-up in the room.” *You* are the “someone” in “someone should do something.”

In this section, we’ll look at three aspects of responsibility: taking ownership, taking charge, and creating calm.

**Take Ownership**

Senior people own the whole problem, not just the parts that go as planned. You’re not running someone else’s project for them: it’s *yours* and you don’t passively let it sink or swim. When something goes wrong, you don’t shrug and decide the work is impossible. You navigate the problem and you’re accountable for the result. ([Chapter 6](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#why_have_we_stoppedquestion_mark) has techniques for getting unblocked that can help here!)

Avoid what [John Allspaw calls](https://oreil.ly/lfRcs) “Cover Your Ass Engineering” (CYAE):

*Mature engineers stand up and accept the responsibility given to them. If they find they don’t have the requisite authority to be held accountable for their work, they seek out ways to rectify that. An example of CYAE is “It’s not my fault. They broke it, they used it wrong. I built it to spec, I can’t be held responsible for their mistakes or improper specification.”*

Ownership also means using your own good judgment: you don’t need to constantly ask for permission or check whether you’re doing the right thing. But that doesn’t mean you should operate in private. While the classic advice is to seek forgiveness rather than ask permission, Elizabeth Ayer, a product and delivery advisor, [offers a more open and predictable approach: “radiating intent”](https://oreil.ly/fXxG4), the idea of signaling what you’re about to do before you do it. You’re giving everyone else context about your actions—and you’re creating an opportunity to intervene if you’re about to do something dangerous.

Ayer calls out another important advantage of radiating intent: “The ‘radiator’ keeps responsibility if things go sour. It doesn’t transfer the blame the way seeking permission does.” That’s key to ownership too.

**Make decisions**

Professional engineers in some disciplines are responsible for putting their seal on documents: an engineer might sign off on the structural integrity of a building, for example. By doing so, they’re attesting that the document is structurally safe and taking on legal liability for any mistakes they’ve made. They’re personally on the hook for that decision.

While software engineers don’t currently have this kind of professional responsibility, as technical leaders, we must be prepared to make the final call and own the outcome. In particular, when a decision is needed, avoid staying on the fence: weigh the options, choose decisively, and explain your reasoning. Be honest with yourself as you consider the trade-offs: you should be able to vote against your own preferences when you know it’s the best move.

Owning decisions includes accepting that you might be wrong. Make the cost of a wrong decision as low as possible, and if it turns out that you *are* wrong, own that, too.

**Ask “obvious” questions**

One of the best things about being senior is that you can ask questions that are so obvious, nobody else is willing to ask them. Here are a few examples:

* It sounds like you’re planning to run a mission-critical microservice in a team with only two engineers. How do you intend to handle on-call for it?
* I assume you’ve evaluated what it would take to move off this old system instead of working so hard to keep it alive?
* What will happen if users start to depend on that incrementing field in your API that you’re telling them to ignore?
* You’ve run this odd-sounding proposal by security, right?
* What would it take to support this use case that we keep asking people not to do on our platform?

As a leader, you have a responsibility to make the implicit explicit. It’s not fair, but if a junior person asks these questions, the team may sigh and say, yes, *obviously* we thought of that. If an expert asks, team members learn that they should include explicit answers to these questions in their design documentation. (Or they genuinely consider the question for the first time!)

**Don’t delegate through neglect**

A few years ago I wrote a conference talk that went a bit viral. OK, we’re not talking *otters holding hands* viral, but it swept across tech Twitter, hit the front page of Hacker News, that sort of thing. The talk was about the leadership and administrative tasks that aren’t on anyone’s job ladder but are needed to make a team successful: all the unblocking, onboarding, reminders, mentoring, and scheduling. I called this kind of work [“glue work”](https://noidea.dog/glue).

Why did the talk hit such a nerve? Because although projects can’t succeed without it, this kind of work is rarely rewarded or allocated fairly. It falls to whomever on the team can’t look away from the problem, often a junior person with a strong sense of ownership.

The problem is, when junior people do too much administrative or leadership work and not enough technical work, they’re spending their prime technical learning years in a way that doesn’t teach them technical skills. That can stunt their careers in the long run. But, often, leaders don’t step in: the glue work is needed for the project to succeed, and they’re just glad it’s getting done.

If glue work is needed for your organization or your project, recognize it and understand who is doing it. Be aware that managers, promotion committees, and future employers might consider this work to be *leadership* when a staff engineer does it, but dismiss it when a more junior engineer does. So take ownership and do a lot of the work that’s not anybody’s job but that furthers your goals. Redirect your junior colleagues toward tasks that will develop their careers instead.

**Take Charge**

Gently redirecting your colleagues towards more valuable work is an example of what I’m going to talk about next: taking charge of the situation. Note that *taking charge* doesn’t necessarily mean you have prior authority. It means that you see a gap and you’re stepping up to fill that gap.

**Step up in an emergency**

Being able to take control of a mess is a key aspect of technical leadership. If security detects a breach, a database gets dropped, or a meteor hits US-East-1, there are likely to be many responders. Unless they’re working together, the ensuing chaos can make the problem worse. Everything goes better if someone is coordinating.

Unfortunately, coordinating only works if everyone *knows* you’re coordinating. If you don’t take charge explicitly*,* you’ll just be one more person making noise. Ideally, you’ll have emergency plans in place before the disaster hits, so that the role of the coordinator is well understood: the classic Incident Command System is a popular choice.[**4**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn90) If not, you’re going to have to find a way to announce that you’re coordinating and set expectations about what you’re going to do. Then make sure your coordination is valuable. A few ways to do this are to take clear notes, make sure that everyone involved in the emergency has the same context, and ask everyone to *radiate intent* about what they’re doing and when.

**Ask for more information when everyone is confused**

Earlier in this chapter, I talked about admitting what you don’t know and asking obvious questions. During an emergency, you’ll often need to do both at once. When teams are sharing information to resolve the issue, they often don’t all have the context to interpret it. Objective facts like “the FooService has 1% 401 errors” aren’t helpful to anyone who doesn’t know what’s typical for the service. Is that bad? Is there a theory for what’s happening? How does FooService fit into this outage?

Someone needs to be brave enough to say, “I don’t know what to do with the information you just gave me!” Take charge and ask. Tech can be fraught with egos and insecurity, and it’s sometimes scary (or legitimately risky!) for junior people to admit that they don’t know something. It’s safer for senior people to ask.

**FEIGNED SURPRISE**

Feigned surprise used to be a standard part of conversation for sysadmins and software engineers: “You’ve *seriously* never used Linux?” It was part of the [BOFH](https://en.wikipedia.org/wiki/Bastard_Operator_From_Hell) toolbox, designed to undermine those who were still learning (the noobs and the lusers) and let the more experienced tech folks feel superior.

How do you ask questions in that environment? Mostly, you don’t. You make a poker face, try to keep up, and hope the topic changes to something you know. When you absolutely can’t avoid it, you ask for help in private. It takes much longer to learn anything.

But then the Recurse Center (then called Hacker School) [called out the phenomenon in its social rules](https://oreil.ly/WLk0r). Naming the behavior gave people power over it: it was something they could recognize and ask others not to do. The Recurse Center built in a mechanism to keep the rules low-stakes too: “The social rules are lightweight. You should not be afraid of breaking a social rule. These are things that everyone does, and breaking one doesn’t make you a bad person. If someone says, ‘Hey, you just feigned surprise,’ don’t worry. Just apologize, reflect for a second, and move on.”

Don’t feign surprise, but go even further. Every time someone apologizes for asking a basic question and there’s a chorus of reassurance from others who insist that it’s actually a *good* question, culture is built. That’s an environment where it’s easy to learn.

**Drive meetings**

Meetings are another place where it really helps to have someone step up and take charge. If the group is passive, distracted, or inclined to turn a work meeting into a social conversation, any one of the attendees can (in theory) say, “OK, let’s get started on our agenda.” But most meeting attendees are hesitant to play that role. Step up when it’s needed. Make sure there *is* an agenda: collect items to discuss at the start of the meeting, or set the example of sending around the agenda in advance. Remember what you’re hoping to get out of the meeting, and drag it back to that topic if it goes too far astray.

If the meeting doesn’t have notes, was it really worth getting together? Meeting notes are a great example of glue work. If a junior person is taking notes, they’re unable to participate, and it’s considered low-status administrative work. If a senior person takes notes, they’re making sure the meeting is effective, and everyone’s very impressed!

Meeting notes are a great lever for making progress on your projects, so don’t hesitate to volunteer to take them. You can record the facts you think are most important, document decisions made, and be the first to frame the decision. Then you can invite everyone to confirm what you wrote. As a moderator, if you need to give everyone a moment to think and reflect, you can also say, “Wait a moment, I need to catch up with the notes.” They’re a useful flow control for the meeting.

**If you see something, say something**

Another common situation where you might need to take charge is an awkward one: when someone just said something disrespectful or offensive in a public channel. Other people in the room might want to say something but feel like they lack the social capital. Use your position as a leader and speak up.

Like many engineers, I find these situations uncomfortable, so I asked Sarah Milstein, engineering VP at Daily, for advice. She always seems fearless when confronted with “advanced humaning” problems, so I was disappointed to learn that speaking up isn’t magically easier when you’re a manager. The adrenaline awfulness, she told me, doesn’t go away. You just accept the discomfort. Go in knowing that it’s going to be awkward, but that it will be better to have said something than not. You don’t have to say the perfect thing—there often *isn’t* a perfect thing—but you do need to speak up.

While the conventional wisdom for feedback is to praise in public and criticize in private, this is a time when it’s vital that you say something public: if it looks like the original message wasn’t addressed, it can create an environment where that kind of message is seen to be acceptable. If someone is attacked in front of a group, you need to support them in front of the same group. If nobody addresses the problem, your group dynamics will become weird and uncomfortable.

It’s best if you can deal with this kind of situation quickly, but it’s OK to return to it a little later: “Look, I feel this hanging in the air and I wish I’d addressed it at the time, but I want to go back to it.” By addressing it, you can “give the energy a place to travel,” Milstein says. She adds, “Almost always, somebody thanks me later for having spoken up in a hard situation.”

Here are some more techniques from Milstein:

* Describe the culture that you’re aiming to build, and use that as a reference. For example, “You all know that respect for each other is a big value here. It’s part of how we get things done. That message violated those norms.”
* Give the person a path to being on the same side as you. For example, if they made a hurtful joke about a tense news story, show that you understand why someone would joke just then. “I get using humor in hard situations, but let’s be mindful that people in this meeting might be affected by what’s going on.”
* If it’s a private conversation, you can appeal to the person’s values: “I know you really care about fairness, so I want to flag something you said that you might not have realized the implications of.”

Finally, this isn’t a thing that should end with you. While you have the power and responsibility to address culture issues, this situation is a behavior problem too. You can also tell a relevant manager so that, if there’s a pattern, they can help address it. That’s their role, not yours.

**Create Calm**

The final factor in being the responsible grown-up in the room: stay calm. Tired, stressed people often disagree about the right way to proceed. If you can stay calm and constructive and avoid casting blame, other people will too.

**Defuse, don’t amplify**

If you’re dealing with a big problem, try to make it smaller. If you’re dealing with a small problem, *keep it small.* When someone brings you a fraught situation, stay calm. Ask questions. Understand why they’re telling you. Do they just need to vent? Are they hoping you’ll take action? Be curious, even about topics you think you understand. If there’s a problem, acknowledge it. Even just seeing that you have the same information and don’t seem to be panicking can be enough to reduce a colleague’s anxiety.

Even if you can see something you can do, don’t react reflexively. A senior person making a fuss can blow up a minor thing into a big, loud issue, so be certain that you have all of the facts and that you’re genuinely helping by joining in. If your actions will amplify rather than calming the situation down, consider staying out of it. Also, remember the warning at the start of [Chapter 6](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#why_have_we_stoppedquestion_mark): make sure this side quest is the right use of your time.

Finally, be cautious about where you share your anxieties or frustrations. While you can acknowledge that there are problems, don’t let your worries about them spill out on more junior people: it’s not fair to ask them to carry your concerns, and you’re amplifying the problem if you upset them. That’s not to say you have to keep your worries to yourself: you can vent to your manager, close peers, or the project sounding board you chose in [Chapter 5](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#leading_big_projects). But be clear about whether you’re complaining socially or you want the other person to do something. Be especially clear in one-on-ones with other leaders about whether you’re asking for action, unpacking something for yourself, or sharing context. *They* might reflexively react and amplify something that you’d just intended to blow off steam about.

**Avoid blame**

I still remember one of the first mistakes I made in production. While updating a customer record, I’d somehow deleted their entire account. I was 22, new to the team (and the industry), and absolutely petrified that taking the blame would mean the end of my short career. My coworker Tim cleaned up my mess and I’ll never forget his reaction: “It’s always interesting to see how new people handle their first screw-up. We’ve all been there.” It was such a relief! Of course I was still upset, but the sick feeling in my stomach was gone. If every one of my coworkers had survived their first mistake, I would too. In the middle of the annoying task of recovering the customer data, Tim took the time to be kind.

A big outage is an expensive training course, and if you’re paying the cost, you’d better all learn something! If someone made a mistake or discovered an edge case by breaking something, create an environment where everyone will feel safe talking through the event. If you’re curious and avoid blame, you’ll find it easier to ask question like:

* Exactly what happened?
* What factors led them down this path?
* Was there information they didn’t have, but could have had?
* Where did their mental models diverge from reality?

**Be consistent**

Have you ever had a leader who is a complete wild card? You don’t know how to prepare for any meeting with them. One day they only care about high-level project delivery dates; the next, they’re asking you to justify the tiniest technical decisions. They tell you that one goal is the most important business need and then, just as you’ve finished adjusting your project plans, they prioritize something else. It’s chaotic. You can’t know where you stand.

Don’t be that leader. Instead, create a sense of safety and calm by being consistent and predictable. Your colleagues should know what they can expect if they ask you to help with something. During times of change or difficulty, the way you show up and express yourself at work can reassure your colleagues: yes, change can be scary, but they can rely on you to be steady while you all work through it.

It’s harder to be consistent when you’re stressed out or working beyond your capacity, so being consistent means taking care of yourself. Remember [Figure 4-3](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch04.html#how_full_is_your_work_schedulequestion): leave a little space in your life for unplanned events. Work in a way that is sustainable for you. That means taking time off, getting enough rest, and doing the things outside work that make you happy. Remember, you’re modeling sustainable work for your colleagues too.

**Remember the Goal**

On to the third attribute of being a role model senior engineer: remembering what the heck you’re all doing here. It’s not just technology! There’s a broader context: a business that’s trying to achieve something, a mission you’re setting out on. In this section, we’ll look at bringing business context (and budget context) to your decisions and solving the entirety of the bigger problems your users care about, not just your team’s tasks. And we’ll think about achieving the goals as a team, not as individuals.

**Remember There’s a Business**

As a senior engineer, you have a responsibility to the future as well as the present. You will always be responsible for creating software that stands up under stress. But you’re working for a business (or a nonprofit, government agency, or other organization) that has goals. The software is the means to those ends, not an end in itself.

**Adapt to the situation**

I took part in a hackathon once for a volunteer event, and I remember the team lead looking at my code and saying, “Wow, tests? That’s, uh, nice.” He was being polite, but it was clear that tests were unusual—and not particularly welcome. Speed was much more important than accuracy, and the code was going to be thrown away later. As far as he was concerned, I’d wasted my time.

Sometimes a faster solution is better; sometimes a more stable one is. If time to market is vital to your business’s survival, getting a shoddy first version out the door might be more important than having beautiful code and architecture.[**5**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn91) Similarly, if you’re shipping software to accompany a holiday promotion or a major sporting event, an imperfect solution is much better than a late one.

Priorities sometimes change during an outage, too. Maybe you’ve got a rule that you always do a clean rolling restart of your service so that you only have a couple of instances offline at a time. When everything is broken, though, your usual principles might go right out the window: sometimes the fastest thing to do is to turn the whole thing off and then back on again. A midlevel engineer might strive for the platonic ideal of a clean, technically elegant fix, but their senior mentor will teach them that this is when you get the system back online first and clean up later.

As the business changes, your priorities will change. *Be OK with that.* It’s inevitable. Growth, acquisition, new markets, or a change in fortunes will mean that your goals may get thrown out as the company pursues a new direction or even a new culture. If you don’t like that or it doesn’t fit your values, you might no longer be in the right place. But if you just resent change, you’ll spend your time being unhappy. Expect it, and you’ll embrace it as a new challenge instead.

**Be aware that there’s a budget**

Your high engineering standards will always be in tension with the amount of money the business is willing to spend on good engineering. That tension doesn’t mean you should drop your principles and start advocating for shoddy software, but do keep the idea of a budget in the back of your mind. Remember that other people are limited in what they can spend on headcount, vendor tooling, and so on.

Don’t obsess about the budget: it’s easy to get frozen in indecision, trying to decide if one project or another is really worth the cost. But have a feeling for what kinds of expenditure, savings, or new revenue are considered big. Understand how your company makes money and know whether you’re in good times or lean times. Bear those facts in mind when you’re deciding what to suggest your organization spends time on.

**Spend resources mindfully**

“Growing up” in Google during a time of plenty, it took me a decade longer than it should have to realize that headcount is finite and staffing a project has an opportunity cost. Part of your technical judgment is “spending” that finite headcount wisely.

You’ll probably have a ton of ideas about places you can innovate, invent something new, or make one of your systems a little better. Make sure you’re choosing work that your business actually needs. Your team has finite time and energy; is this the right way to spend it? Take [Dan McKinley’s advice](http://boringtechnology.club/) too and be judicious about where you spend your “innovation tokens”: that is, your company’s “limited capacity to do something creative, or weird, or hard.” If you can only invest in a few places, is this the right place?

Build the most useful thing, not the thing that would be most fun to build. And when it’s time to stop polishing something and declare that it’s good enough, stop.

**Remember There’s a User**

I remember once sitting with a vendor in a huge cafeteria with hundreds of coworkers. My colleague Mitch and I listened as the vendor explained how a feature we’d been asking for was now ready for us. But I’d tried the feature and it didn’t work. We argued back and forth until I pulled out my laptop and showed him.

“Oh, I get it now,” the vendor said. “You’re using Chrome. It works on Firefox and Internet Explorer.” (Yeah, this was a while ago.) “But don’t worry, not many people use Chrome.”

“Look around you,” Mitch replied, gesturing around the cafeteria. “See all of these people? Everyone in this room uses Chrome.”

I’ve seen too many teams create a feature for a set of fictional, perfect users who don’t exist.[**6**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn92) Know who uses your software and know how they use it. Make sure they can use the thing you’re creating for them, and that they *want to*.

Part of getting this right is the classic solution: write it down! Be clear about the exact requirements you’re creating for, and share those requirements broadly. Get the proposed API reviewed before you start the code. Show a mockup of the user interface before you start creating it. Check in frequently and show updates. Once again, avoid CYAE: whether you built it to spec or not, if you didn’t make your user happy, you built the wrong thing.

**Remember There’s a Team**

The final thought in focusing on the mission: remember you’re not doing this work alone. While you may be the best coder on the team, the most experienced engineer, or the fastest problem solver, that doesn’t mean you should jump on all of the problems. You’re working as part of a team, not a collection of competing individuals. Don’t become a single point of failure where the team can’t get anything done when you’re not available. It’s not sustainable. It hides problems.

Just like I advised you to be self-aware, be aware of the capabilities of your team. If you can reach your goal by empowering someone else to do better work, that’s just as much a victory as if you solve it yourself. Consider your impact to be what wouldn’t have happened without you, not just what you personally did.

**Look Ahead**

While there are, as you’ve seen, times when your first priority will be to get something to market quickly, most of the time you’re planning for a longer time horizon. The code and architecture you work on are likely to still be in use in 5 or 10 years. The interconnected software systems that make up your production environment may last much longer, and each component will influence the ones that follow.[**7**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn93) As Titus Winters writes in *Software Engineering at Google* (O’Reilly)*, “*Software engineering is programming integrated over time.” Expect the impact of your software to stick around.

Your organization, codebase, and production environment probably existed before you joined them. They’ll probably exist after you move on. Don’t optimize for *now* at the cost of future velocity or engineering ability. It’s OK to plant some seeds that you won’t personally see grow.

Here are a few ways you should be thinking beyond the current moment.

**Anticipate What You’ll Wish You’d Done**

Remember our question from in [Chapter 3](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch03.html#creating_the_big_picture): “What will Future You wish Present You had done?” When you’re making plans or doing work, consider your future self and your future team to be stakeholders: after all, they’ll have to deal with whatever decisions you make now.

**Telegraph what’s coming**

Be clear about what your broad direction is, even if you don’t know the details yet. Here’s an example: teams sometimes avoid announcing deprecation dates for old systems, because they’re not quite ready to begin the major migration to the new system. But you can announce the *intention* to deprecate it. If everyone knows a migration will begin in a year or two, new projects will know not to invest in it. Some teams may find themselves with free time and move to the new system without you even asking them. A small amount of work now will set people’s expectations, save their time, and make your future deprecation project a little easier.

**Tidy up**

Have you ever had to work in a tool shed where the last person didn’t clean up after themselves? It’s horrible. You grab the drill and the battery’s out of power. The safety goggles aren’t in their case; you search through three boxes before finding them with the sander. The floor is covered in detritus. There is no flow state in an environment like that. Everything takes three times as long as it should.

Now think about what it’s like when every tool you want is at arm’s reach. Your workflow just *works*. So take the time to leave your production environment, codebase, or documentation so that it *just works* for whoever comes along next. Write tests that will let you refactor your code without breaking things. Follow your style guide so that the people who copy your approach will also be following your style guide. Leave no traps, like dangerous scripts that everyone needs to remember not to run or configurations that are changed locally but not updated in source control. Make it safe to move around.

**Keep your tools sharp**

Don’t *just* tidy up: continually invest in making your environment better. If you can move quickly and safely, you’ll spend less time on repetitive work and you’ll be able to do more. Increasing your velocity increases your reliability, too: every minute you shave off your time to detect a problem or deploy a fix is a minute you’ve taken off every outage.

Look for optimizations that will let you build, deploy, and release more quickly: smaller builds, intuitive tooling, fixing or deleting flaky tests, repeatable processes, automation everywhere. Be judicious about where you invest: building tooling, platforms, or processes takes time, so choose the optimizations that will genuinely make a difference.

**Create institutional memory**

Every time someone leaves your company, you lose institutional knowledge. If you’re lucky, you have some old-timers storing history in their brains. But eventually, inevitably, you’ll have complete staff turnover.[**8**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn94) When an old system breaks, there’ll be nobody left to say “Oh, yes, I remember when we ran into this before. Here’s what we did last time.”

My ex-colleague John Reese, at the time a principal engineer at Google, often also took the role of systems historian: he curated a record of how the site reliability organization had evolved and how running software in production had changed over the years. To create institutional memory, he wrote in-depth articles about the parts of the ecosystem he knew best, then interviewed others to uncover the past, documenting formative systems and practices. Although he’s moved on from Google now, that history lives on with a new set of curators.

While most organizations don’t have someone deliberately writing down their history (though maybe we should!), you can send information into the future by writing things down. This includes decision records that explain what you were thinking, systems diagrams that include the obvious things that “everyone knows,” and code comments that include context on what’s going on. However you create the history, include searchable keywords so that future people have some chance of understanding what you did and why—and think about what you know that future people might not.[**9**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn95)

**Expect Failure**

My all-time favorite incident retrospective is [the one Fran Garcia wrote](https://oreil.ly/zsPgE) about his then-employer, Hosted Graphite, being taken down by an AWS outage. The reason I love this one is that Hosted Graphite didn’t *use* AWS, so the team was quite surprised at being affected by its outage.[**10**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn96) They had no way of predicting it.

How many unpredictable failures like that lurk in your systems? Assume it’s a *lot*. The [network will fail](https://oreil.ly/OuP1u), the hardware will fail, the people will have an off day. There will always be bugs. Odd interactions between parts of the system you haven’t even thought about will cause problems.

You can’t predict everything that will go wrong, but you can predict that *something* will go wrong. Plan for what you’ll do when it does. Build the expectation of failure into your products: test the error paths as thoroughly as the success paths, and make the product do something sensible and user-friendly when it doesn’t get the kind of response it expects. Make sure you’ll find out when your systems aren’t behaving, and have a plan for how you’ll respond to it.

Plan in advance for major incidents by adding some conventions around how you work together during an emergency: introduce the incident command system I mentioned earlier, for example, and practice the response before you need it. Your disaster plans will invariably have something go wrong, so simulate disaster with [chaos engineering](https://oreil.ly/NWys8) tooling or controlled outages. Drills, game days, or tabletop exercises can let you uncover which parts of your response won’t work. And of course, if you haven’t tested restoring your backups, assume you don’t have any backups.

**Optimize for Maintenance, Not Creation**

Software is created once, but it will need to be maintained for years. If you’ve got a binary running in production, it will need monitoring, logging, business continuity, scaling, and so on. Even if you intend to never touch the code again, the technical or regulatory ecosystem may force you to care: think of all the old systems that needed to be updated for Y2K, to support IPv6 or HTTPS, or for compliance concerns like SOX, GDPR, or HIPAA. Those won’t be our last disruptive changes.[**11**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn97)

Software gets maintained for much longer than it takes to create it, so don’t build code that’s hard to maintain. Here are some ways you can help Future You and your future team.

**Make it understandable**

At the moment you create new code or design a new system, you understand it well. Probably the people on your team also have a strong mental model of how it works. Expect that knowledge to decay a little every day. The system will never again be as well understood as it is on the day it’s created. If it’s hard to understand then, good luck in two years, when something breaks and you’re trying to load that mental model back into your brain.

You have two choices to let future people understand your system.

One option is to focus on education and hands-on experience. You can run continual classes about the system, making sure that everyone who might have to work on it in future is fully trained and has logged enough hours to handle any problems that might arise.

The other option is to make it as easy as possible for people to understand the system when they need it. That means writing documentation with that future person as the main audience: a clear, short introduction; at least one big, simple picture (use arrows to show which direction data moves); links to anything they might wonder about. Then expose the system’s inner workings as clearly as possible. Make it possible to see what it’s doing, through tooling, tracing, or useful status messages. Make your systems *observable*: easy to inspect, analyze, and debug. And keep them simple, which I’ll talk about next.

**Keep it simple**

There’s a Martin Fowler quote that I love: “Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”[**12**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn98) Senior engineers sometimes think they can demonstrate their prowess with the flashiest, most complicated solutions. But it’s easier to make something complicated. It’s much harder to make it simple!

How can you make something simple? Spend more time on it. When you think of a solution to the problem you’re working on, treat it as “just the first.” Spend at least the same amount of time on another solution. Now that you understand it better, see if you can make it simpler: fewer lines of code, fewer branches, fewer teams, fewer hours of maintenance, fewer running binaries, fewer files touched.[**13**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn99) The longer the system is intended to last, the longer you should spend trying to make it as simple as you can. Make it easy to build mental models of the system or the code.

Beware of organizations that seem to reward complexity. Ryan Harter, a staff data scientist, [has written about](https://oreil.ly/Su2IS) how he’s seen people create complicated solutions to prove that they’re doing hard work. “I’ve seen folks slip machine learning into places it doesn’t belong to get a flashy launch.” He cautions, “Really, what we should want are simple solutions to complex problems. The complexity of our work is a cost to bear, not something to maximize!”

When you’re dealing with inherently complex problems, make a deliberate decision about where in the system you’re going to put the complexity: that one terrifying module with the inscrutable business logic or performance optimizations. Make it so that someone looking at the entire system can treat that component as a magic black box and reason about everything else, so that there’s a single place to go to when it’s time to understand and modify the complex part.

**Build to decommission**

Someday your system will be turned off. How hard is that going to be for the people working on it then? Will they have to dig deep into the logic of other systems, unwinding tendrils that touch business logic and tracing through code to understand what data they’re accessing? Or will there be a clean interface and a simple cutover?

Your architecture will evolve, and your components will settle into the middle. While it might be faster now for you to just wire in the new system, library, or framework, think about what will happen afterward. Will it be possible to replace it later without demolishing whatever other people have built on top of it?

Imagine knowing that *you personally* will need to decommission this component in 10 years. Future You won’t be any less busy than Present You, so what can you do to help them out? Might you add a clean interface, make it easy to see which clients are still using a server, or design in a way that keeps a little distance between two systems that are being integrated? If you set out from the start to build a component that’s easy to decommission, you’ll have the side effect of building something modular and easy to maintain.

**Create Future Leaders**

Building up your team is an important part of future planning. It often will be easier and faster for you to solve problems or lead projects than for others to do it, but that doesn’t mean you should take over. Your junior engineers are future senior engineers. Give them the space to learn, and opportunities to do hands-on work and solve increasingly difficult problems. [Chapter 8](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#good_influence_at_scale) will have a lot more about how to continually raise their skill levels.

I’ll leave you with one more quote from John Allspaw’s [“On Being a Senior Engineer”](https://oreil.ly/aANg3):

*The degree to which other people want to work with you is a direct indication of how successful you’ll be in your career as an engineer. Be the engineer that everyone wants to work with.*

If you take nothing else away from this chapter, take that last sentence: *the metric for success is whether other people want to work with you.* If they don’t, reevaluate your approach.

**To Recap**

* Your words and actions carry more weight now. Be deliberate.
* Invest the time to build knowledge and expertise. Competence comes from experience.
* Be self-aware about what you know and what you don’t.
* Strive to be consistent, reliable, and trustworthy.
* Get comfortable taking charge when nobody else is, including during a crisis or an ambiguous project.
* When someone needs to say something, say something.
* Create calm. Make problems smaller, not bigger.
* Be aware of your business, budgets, user needs, and the capabilities of your team.
* Help your future self by planning ahead and keeping your tools sharp.
* Write things down, even when they’re “obvious.”
* Expect failure and be ready for it.
* Design software that’s easy to decommission.
* The metric for success is whether other people want to work with you.

[**1**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn87-marker) There’s also sometimes an implication of “doesn’t seem nerdy” or “doesn’t look like an engineer”; we all should be aware of our [implicit bias](https://oreil.ly/jRVit). But I’m speaking here to the people who are genuinely trying to help themselves or their coworkers build skills.

[**2**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn88-marker) When I started a new job after 12 years at Google, a company famed for using its own internal technology stack for everything, I relied on this kind of pattern-matching. I did lots of drawing systems on whiteboards and asking, “Is there a thing that looks kind of like this, and you would use it in this situation? Oh, *that’s* what Envoy does! OK, got it!”

[**3**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn89-marker) If you’re thinking, “I don’t make mistakes because I’m competent and careful,” the gut-punch feeling when you do make one will be so, so much worse.

[**4**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn90-marker) The [Incident Command System](https://oreil.ly/DUxaG) was introduced by fire departments in the ‘60s and is now used by most emergency services in the United States to coordinate disaster response. One of the roles it defines is the incident commander, someone whose job is not to fight the fire, but to coordinate and take command. It works well for the kinds of software outages that are chaotic or that cross multiple teams.

[**5**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn91-marker) But don’t help a business to survive at the cost of releasing software that endangers, exploits, or hurts other people: the company has bought your time and energy, but not your moral compass. We’ll look more at values in [Chapter 9](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch09.html#whatapostrophes_nextquestion_mark).

[**6**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn92-marker) Perfectly spherical users, as some friends say.

[**7**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn93-marker) Think of it as a [Ship of Theseus](https://oreil.ly/nBaaK): every individual component may get replaced over the years, but the fundamental system continues. It’s all metaphysical architecture.

[**8**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn94-marker) Another Ship of Theseus! The people have all changed but the organization remains.

[**9**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn95-marker) Be inspired by the [Sandia National Laboratories report](https://oreil.ly/PWTxV) on creating pictographic information to deter future humans from interfering with nuclear waste repositories in 10,000 years when current languages will be long gone. You don’t need to think quite that far ahead, but imagine if the systems you work with are still around in 10 years: what will people need to know? How can they accidentally hurt themselves?

[**10**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn96-marker) In case you’re curious: the outage meant that a lot of Hosted Graphite’s users became slow all at once and their usually short-lived connections stayed open, increasing the number of connections until they reached a limit in the load balancer and prevented anyone else from connecting. The write-up is a good time.

[**11**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn97-marker) [2038 is coming](https://oreil.ly/SOdWl)!

[**12**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn98-marker) *Refactoring: Improving the Design of Existing Code* by Martin Fowler et al. (Addison-Wesley).

[**13**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch07.html#ch01fn99-marker) If it’s so few lines of code that it’s getting obfuscated and complicated again, you went too far. We’re aiming for understandability, not stunt programming.