**Chapter 8. Good Influence at Scale**

How do you raise the skills of the people around you? As a staff engineer, part of your job is to enable your colleagues to do better work, to create better solutions, to be better engineers. We already started on this journey last chapter with the idea of being a *role model engineer*: doing the best engineering work you can and letting others see it. That’s what we usually mean when we say someone is “a good influence.” They behave in the way we’d like others to behave.

But now we’ll go further and look at more active ways you can use your good influence to improve other people’s skills and your organization’s engineering culture.

**Good Influence**

When you work with someone who is missing skills or has lower standards than you, don’t get frustrated: take the time to bring them up a level.

Why is it so important to help other engineers do better work? First, good engineering inevitably goes beyond yourself. If your colleagues do better work, you can do better work too. While some engineers are extraordinary solo artists, even the most powerful virtuoso will meet some problems that are too big to solve alone. If you can help your colleagues become better engineers, you’ll be working with more competent people, which means your own work will be easier (and less annoying). Better engineers means better software, which means better business outcomes.

The second reason is that the industry keeps changing. Even if your engineering organization is on the cutting edge right now, at some point there’ll be a new game-changing architecture, tool, or process that you want everyone to adopt. Getting the teams you work with to use it will be an exercise in frustration unless you know how to influence and teach new skills.

The last reason is that it’s just the right thing to do. As a senior person, you have outsize influence on how well your organization creates software, and even on how our industry behaves and evolves. In the same way that you take pride in improving your code quality, reliability, and usability, you can take pride in your high standards. If you teach your midlevel colleagues to be fantastic engineers, think of the midlevel engineers *they’ll* be teaching in 10 years. You’re sending high standards into the future.

**Scaling Your Good Influence**

For most of us, leadership through influence starts with individual relationships: reviewing someone’s code, hosting an intern, or mentoring a new grad. You might progress to leading small teams, probably having one-on-one meetings with each person on the team. As your *scope* grows, though, it becomes harder to have enough influence purely through individual interactions. There just aren’t enough hours in the day. As Bryan Liles, a principal engineer at VMware, [describes it](https://oreil.ly/ScVHa): “My job at VMware is to be able to influence 14,000 engineers…I’m trying to think ‘What can I do to make 14,000 engineers better?’”

Depending on your seniority, your scope, and your aspirations, you might be aiming to influence far fewer people (or many more!). In this chapter, we’re going to look at good influence at the micro and macro levels: from improving the skills of your coworkers, team, or group to changing the trajectory of your whole organization or even the entire industry. I’ll describe three tiers of influence (see [Figure 8-1](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#tiers_of_influencedot_group_influence_g)):

*Individual*

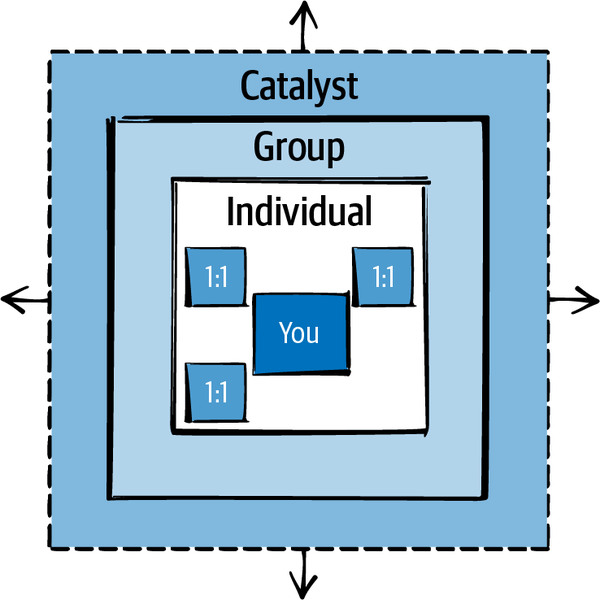
You’re working in a way that grows another person’s skills.

*Group*

You’re scaling your influence by bringing new skills or a change of approach to multiple people at once.

*Catalyst*

The change you make goes beyond your direct influence. You’re setting up frameworks or community structures that let your positive influence continue even after you step away.



**Figure 8-1. Tiers of influence. Group influence goes further than individual influence. Catalyst influence keeps going even when you stop investing more effort.**

What forms can influence take? I’ll describe four mechanisms for bringing your colleagues up a level, and give examples at each tier:

*Advice*

We’ll start with giving advice, both solicited and unsolicited. At the individual level, this can mean mentoring, peer feedback, or just answering questions. You can scale your advice to the group level through writing and presenting, and get to the catalyst level by making it easy for your colleagues to advise each other.

*Teaching*

We’ll look at deliberately teaching skills to individuals through training, pairing, shadowing, or coaching. Then we’ll scale to groups using onboarding materials, codelabs, classes, and workshops. The catalyst level is teaching other people to teach, setting curricula, and influencing the topics that everyone is exposed to.

*Guardrails*

We’ll explore how to give people guardrails so they can work safely. For individuals, I’ll talk about code review, design review, and how to be someone’s project guardrail. For groups, we’ll look at some of the processes, policies, and robots that can keep us on track. Finally, at the catalyst level, we’ll explore the ultimate guardrail: culture change.

*Opportunities*

Finally, we’ll look at helping people grow by matching them with opportunities that will help them learn. For individuals, I’ll talk about delegation, sponsorship, and highlighting good work. For groups, the biggest opportunity you can give your team might be stepping back, making space, and sharing the spotlight. And it becomes a catalyst when the entry-level and midlevel folks you sponsored become dynamic, skilled, problem-solving senior people who take on challenges and create new opportunities that you never imagined.

[Table 8-1](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#some_examples_of_scaling_advicecomma_te) shows these three tiers and four mechanisms, with some examples.

|  | **Individual** | **Group** | **Catalyst** |
| --- | --- | --- | --- |
| **Advice** | Mentoring, sharing knowledge, feedback | Tech talks, documentation, articles | Mentorship program, tech talk events |
| **Teaching** | Code review, design review, coaching, pairing, shadowing | Classes, codelabs | Onboarding curriculum, teaching people to teach |
| **Guardrails** | Code review, change review, design review | Processes, linters, style guides | Frameworks, culture change |
| **Opportunity** | Delegating, sponsorship, cheerleading, ongoing support | Sharing the spotlight, empowering your team | Creating a culture of opportunity, watching with pride as your superstar junior colleagues change the world |
| Table 8-1. Some examples of scaling advice, teaching, guardrails, and opportunity across the organization and beyond. | | | |

Think of these tiers and mechanisms as a list of options available to you, not a checklist to try to complete. Play to your strengths and do the ones that you enjoy, find easy, or want to get better at. And although they’re framed as a hierarchy, don’t skip past the “smaller” ones. Leveling up your colleagues through code review or sponsorship will have a ripple effect across the company, and even the most world-changing technologists still mentor people they believe are worth investing in.

Similarly, don’t get too focused on chasing the *catalytic* types of influence in the right-hand column of [Table 8-1](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#some_examples_of_scaling_advicecomma_te). Having too many programs and frameworks can be overwhelming for your organization, and you will often have more impact by taking part in an existing initiative than by setting up something new. If there’s already an onboarding curriculum, for example, teaching a class in it will usually be more valuable than setting up a separate education initiative. Navigating a single team through a difficult design can often be much more important than tinkering with the RFC process. Do the individual and group work first, and only go broader if the need and value is very clear.

All that said, let’s start with advice.

**Advice**

“Free advice,” the maxim goes, “is worth exactly what you paid for it.” And it’s true that advice is *noisy*: there’s bad signal mixed in with good, and it tends to not be tailored to the person who’s receiving it. However, everyone needs advice sometimes, and giving it is one of the ways you can pass on your experience. If you’ve had successes and made mistakes, let other people learn from them too.

**WHO ASKED YOU?**

Before you offer advice, think about whether it’s welcome*.* Solicited advice is when someone else asks for something: a recommendation, feedback on their work, help deciding what to do. When it’s unsolicited, they haven’t asked.

Before you offer your thoughts, think about whether the other person is asking for them. Think too about whether you even have enough context to tell them something that’s both helpful and nonobvious. If you’re not sure whether your advice will be welcome, *ask them*.

There are times when unsolicited advice is valuable. “Your slides were amazing, but you talk to your shoes when you present and it’s hard to understand you” is difficult but probably kind unsolicited advice.[**1**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn100) But think about your role and your relationship with the person. Some advice should come from friends, not strangers. And, again, don’t just launch in. Ask “Can I offer some advice?” and get their permission.

Here are some other places where unsolicited advice might be helpful:

* When you think the person is in a situation they can’t see out of. “Hey, I know this isn’t the question you’re asking, but that job situation you described doesn’t seem healthy. You deserve better.”
* When you have key information the other person doesn’t have. “I heard you say you’re going to start using the Foo platform; just want to make sure you know it’s going to be turned down next year.”

If you’re itching to give unsolicited advice on a topic nobody is asking you about, consider writing a blog post or tweeting about it instead.[**2**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn101)

**Individual Advice**

There are a few ways that you might give someone individual advice on a situation they’re in. These include being a mentor figure to them, answering their questions, commenting on work they did, or giving peer feedback at performance review time.

**Mentorship**

Mentoring is often an engineer’s first experience of leadership. In companies with formal mentorship programs, you might be assigned to help a new person get oriented. Mentorship can happen organically too: sit next to someone, introduce yourself, and answer their questions, and you might find yourself with an informal mentee.

By sharing your perspective and what you’ve learned, you can accelerate other people’s learning and save them from making unnecessary mistakes. You can tell them stories about similar situations you’ve been in, what you did, and what the outcomes were. Senior engineering director Neha Batra [describes mentoring as](https://oreil.ly/2QWel) “sharing your experience so an engineer can leverage it themselves.” But remember, mentoring is focused on *your* experience.

Solicited or not, the advice that worked for you might not work for someone else. Author and management coach Lara Hogan (whose name will come up a lot this chapter!) warns that “advice that might work for one person (“Be louder in meetings!” or “Ask your boss for a raise!”) may undermine someone else, because members of underrepresented groups are unconsciously assessed and treated differently.”[**3**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn102) The best practices of a decade ago also might not work for a younger coworker now, and the social dynamics (or technology stack) of the experience you had might not map well to the one your colleague is facing. I’ve had colleagues push back on advice that I targeted wrong. “Just DM the director and ask them to invite you to the meeting” is an easy thing to say when the director is a peer, and much more difficult when they’re your boss’s boss’s boss!

Be careful of unsolicited advice in mentor/mentee conversations. When someone starts describing a difficult project, for example, the easy, intuitive response is to say what you, the sage advice-giver, would do in the same situation. It’s kinder (if more difficult) to figure out what they actually need. Maybe it turns out that they do want help. But it’s just as likely that they’re looking for reassurance that other people would also find the problem or situation difficult. They might be doing [rubber duck debugging](https://oreil.ly/vCJ0t), explaining something to you so they can unpack it for themselves. They might just want to tell a war story, to hear commiserations and congratulations for what they’ve navigated so far. Unsolicited advice derails all of those things. It can help to ask “Do you need space to vent or are you looking for advice?” and then give either comfort and validation or solutions as requested.

Mentoring is not just for new people: I have mentees with decades of experience as well as mentors of my own who I ask for advice. It’s not necessarily one-way either. Your mentee might give you a new perspective or teach you about topics that they know better than you do.

If you’re getting into a mentoring relationship, set it up for success. Set expectations, such as that you’ll meet once a week for six weeks. Agree on what you’re trying to achieve: does the mentee want to onboard and feel comfortable in a new company, learn a new codebase, or get career advice to strive toward a new role? If all of this is settled up front, you’re less likely to find yourselves sitting in a room staring at each other all “What were we supposed to talk about?”

**Answering questions**

If you have a vast repository of knowledge and everyone’s afraid to ask you anything, your knowledge will stay in your own head. Be accessible. Depending on your work style, that might mean offering office hours, being friendly and easy to DM, or spending some of your time just hanging out near the teams you work with—office spaces with sofas are fantastic for this sort of thing.

Make reaching out to you worth the effort. Some engineers seem to guard their knowledge preciously, answering only direct questions and not a word more.

*“Will the /user endpoint give me the user’s full name?” asks the junior engineer.*

*“No,” replies the senior engineer, “it can only give you the username.”*

*The junior engineer tries to hack around the problem for an hour before nervously asking, “Is there a different endpoint that could give me the full name?”*

*“Sure,” says the senior engineer, “use /fulluser.”*

What a waste of an hour! The senior person had information, and it didn’t occur to them to impart it. That doesn’t mean you should *infodump,* offloading every snippet of information that could connect to the topic at hand. But understand what advice is being *implicitly* solicited, even if it’s not directly asked for. If you’re not sure, ask what your colleague is trying to do and ask if they need help.

Code and design review can be another time to answer implicit questions and give advice. If you see a place where your colleague could solve a problem in a better way, tell them. But be clear about whether you’re just sharing interesting information or asking them to change course: it’s frustrating to get a bare comment like “The Foo library also would work here” without context about whether that means you hate the current approach.

Domain-specific information goes beyond technology. If you’ve learned ways to navigate your organization and get things done, that’s valuable knowledge that you can pass on to mentees and other colleagues. Share the topographical map you built in [Chapter 2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch02.html#three_maps)!

**Feedback**

One of my self-appointed roles in my current job is to be a test audience for colleagues doing conference talks. I love watching tech talks and I’ve invested a lot of time in learning how to do them well, so the presenter and I both benefit. As I watch the talk, I take a ton of notes, highlighting the parts I found funny, insightful, or educational. But I also point out anything that didn’t work, anything I thought wasn’t correct, anywhere I started to tune out. Good and bad, I tell the truth. It’s a waste of the presenter’s time otherwise.

When someone asks you to review a document or pull request or conference talk, do call out the sections that you think are great, but pay them the respect of being (kindly!) honest about their work. Giving constructive and critical feedback isn’t easy. It takes effort to tease out exactly what isn’t working and find the words to explain why it’s not as good as it can be. It’s a more difficult conversation. But you won’t help your colleague if you hide the truth.

**Peer reviews**

If your company has a performance management cycle, you might be asked for a specific kind of feedback: peer reviews. These reviews have two audiences and you should keep both in mind.

The first is the person who asked you for feedback. Assume that they asked because they genuinely want to know how to improve. I’ve seen people struggle with “What could this person do better?” questions in peer feedback, because it’s easy to see the answers as criticism. But take the question literally: what *could* your colleague do better? How could they become more awesome? If you can’t think of anything, ask yourself why they aren’t one level more senior (or two!), and give them advice on behaviors they should focus on to get there.

But also remember that the feedback will be read by the person’s manager and potentially others who are calibrating their performance or evaluating them for promotion. Give those people the information they need to help your colleague grow and to notice patterns that need to be addressed. But think about how your words will be perceived and whether they can be taken out of context by someone who doesn’t know the work you’re describing. If you find yourself holding back on describing an area for growth because you don’t want to accidentally torpedo a well-deserved promotion, consider delivering private feedback by email or in person instead.

Just like giving mentorship advice, remember that what works for one person might be terrible advice for another. This is especially prevalent in advice about communication style. For example, “be aggressive” is advice that will make some people seem more like leaders, but will get others in trouble. It’s a common joke in tech women circles that you know you’re acting at senior level when you get your first peer review saying you’re “abrasive.”[**4**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn103) (That might also be the first time your reviews don’t say you should be “more assertive.” It’s a fine line to walk.) So watch out for [implicit bias](https://oreil.ly/KYbCm) and be aware of how you’re describing the same behavior across different people. Was it “consensus building” or “indecisiveness”? Were they “refreshingly down to earth or “unprofessional”? Often it depends on who you’re talking about. The folks at Project Include offer more [recommendations for providing feedback](https://oreil.ly/UMO42).

**Scaling Your Advice to a Group**

You can’t meet individually with everyone who needs advice or write feedback for your whole organization: you’d have no time left in your quarter. But you can give a tech talk or write an article on literally anything you want, and chances are it will reach some people who find it helpful. That’s a great way to scale your advice.

Years ago, a group of volunteers at Google wanted to increase the quality of testing and hit on a novel solution: they started writing their advice as simple one-pagers, printing them out, and putting them up in toilet stalls. [“Testing on the Toilet”](https://oreil.ly/F2bnz) is the ultimate in unsolicited advice, but it’s popular and amusing, and people read it!

If you want to tell something to more people, write it down. Documentation means that you don’t have to explain how to do something again and again. A FAQ, a how-to, even a descriptive channel topic can let you say something once but have it read by many people. And if you write something that applies to people outside your company, consider sharing it further as a blog post or article.

In addition, look for opportunities to get a microphone and an audience. Can you get a slot at an all-hands meeting, conference, or tech talk event? Sometimes you can use these opportunities to deliver a message you’re focused on *alongside* the message you’ve been asked to share. I was invited once to present to a huge group about an incident of my choice. At the time, I was really trying to get the word out about dependency management and how important it is to be deliberate about what systems you depend on. So I talked the group through a recent outage that had been made much worse by a circular dependency that stopped the systems from coming back online.[**5**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn104) The opportunity to present at this big meeting let me frame the outage story so that it highlighted the message I wanted to share. It was an audience and a microphone at the right time.

**Being a Catalyst**

If you want to be a catalyst, set up advice flows that don’t need you to be involved. Make it easy for your colleagues to help each other.

If your team or organization relies on one-on-one conversations to understand how anything works, one of the most powerful things you can do is encourage people to write things down. Start small: going from an oral culture to writing *everything* down won’t win you any friends. Instead, look for a small but meaningful change. Are you missing an easy-to-use documentation platform? Might teams sign on for creating a FAQ of the most common questions they’re interrupted by? Might your director endorse a quarterly documentation day?

You can scale audience-and-microphone-style advice by setting up monthly tech talks or lunch-and-learn meetings. This is a bigger commitment than just scheduling the meeting: you’ll have to solicit talks, send reminders, and possibly watch practice runs. Be clear about what you’re getting into, plan for it on your time graph*,* and ideally start with at least three people to share the load.

Similarly, you can scale mentorship further by setting up a mentorship program. Be warned: the administrative work will be more time-consuming than you might expect, and this work is generally not considered to be part of an engineer’s job. If you can find a manager who is interested in doing the same thing, they will likely find it easier to frame the work as part of their job description. Convincing *someone else* to set up a mentorship program still counts as being a catalyst!

**Teaching**

On to the second type of good influence, and it’s a step up from advice: teaching. What’s the difference between *telling* people things and *teaching* them things? Understanding. When you’re giving advice, you’re explaining how *you* relate to the topic, and the receiver can take your advice or leave it. When you’re teaching, you’re trying to have the other person not just *receive* the information but internalize it for themselves.

Deliberate teaching is not just for a more senior person to help a more junior one: it’s useful any time someone new is joining a team, or when you’ve got more domain knowledge than someone else. I love asking a colleague for an overview of their systems, for example, so I can fill in a mental gap in the overall architecture. At the end of an hour of whiteboarding, I’ll have a much clearer picture of how their systems interact with others, and I’ll have made my own diagram to refer back to or to add to their documentation.

**Individual Teaching**

Anywhere there’s a knowledge gap between you and someone you’re working with, there’s an opportunity to teach. Sometimes this means formal training or coaching. But there’s also plenty of teaching to be done within the regular structures of working together: pair programming, shadowing, and review.

**Unlocking a topic**

Think back to the best classes you ever took. What was successful about them? I bet you walked away feeling like you had a handle on something you didn’t have before: a new skill or understanding that you could build on. Great classes “unlock” a topic for you, sparking curiosity and interest.

Teachers have a specific goal, often formalized in a lesson plan. If you’re teaching, you should too. Some examples:

* Are you giving an overview of a system? If so, by the end of the session, the person you’re teaching should be able to draw the system on a whiteboard and describe it to someone else.
* Are you walking through a codebase? Aim to give them everything they need to send their first pull request.
* Are you showing them how to use a tool or API? Describe three to five common scenarios they should be able to handle on their own by the end of the session.

Successful teaching includes hands-on learning and activating knowledge: the student should be *doing* as well as listening. Find opportunities to let them lead, whether that means using the tool themselves, typing commands, or opening tabs on their laptop instead of yours. It’s even better if they end up with an “artifact” to refer back to, like a diagram or code snippet.

**Pairing, shadowing, and reverse shadowing**

Here’s another way you can teach: working directly with someone else. Working together has a spectrum of approaches (see [Figure 8-2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#the_spectrum_of_working_togetherdot_dif)), from shadowing, where you’re doing all of the work with your coworker observing, to pairing, where you’re working together, to reverse shadowing, where *they’re* doing all of the work with you watching to give them feedback.



**Figure 8-2. The spectrum of working together. Different points on this line will be useful in different situations.**

Shadowing is a way of teaching by demonstrating: your “shadow” watches you execute a skill and takes notes on how you’re approaching it. It’s a great opportunity to be a visible role model and show your colleague how to work with high standards.

With pairing, you’re still working together, but the “shadow” has become an active participant. Pairing can mean pair programming, coauthoring a document, whiteboarding an architecture, or solving a problem together. It’s another opportunity for role modeling, but also for teaching: working side by side means you’ll be able to share knowledge and check for understanding in real time.

Finally, you might “reverse shadow,” where the learner performs the task and the experienced person watches and gives notes. No matter how closely the learner has paid attention, they’ll learn the most by activating the knowledge and practicing the task. Reverse shadowing can also serve as a type of guardrail, which I’ll discuss later in this chapter.

**Code and design review**

Reviewing code and designs can be an excellent form of teaching. You get to highlight perils your colleague might not know about and suggest safer alternatives. You also get to encourage behaviors you want to see more of.

A review from a senior person can be a real confidence booster. Be careful, though: a review done *wrong* can destroy someone’s confidence rather than boosting it. It can be soul-destroying to work through a barrage of comments that are condescending, seem arbitrary, or that you don’t understand.

As a teacher, your job is to impart the knowledge and point out problems in a way that retains your student’s confidence and growth mindset. Code review will show up again in the “guardrails” section, and I’ll talk there about reviewing to prevent harm to your systems, but for now, here are some ideas to bear in mind as you review to *teach*:

**Understand the assignment**

Be aware of the context. Is your colleague new to the language or technology and looking to learn, or do they just need a second pair of eyes for safety?[**6**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn105) Understand the stage of work, too: if you’re reading a first high-level draft, start with the foundations and the approach and don’t get into the nitpicky details. If everyone has bought in and this is the last review before launch, it’s not the time for big directional questions: get right into the weeds and be extra alert for what could go wrong.

**Explain why as well as what**

A review comment like “Don’t use *shared\_ptr,* use *unique\_ptr*” only tells the code author what to do right now. They won’t know what to do next time. Teaching means sharing understanding, not just facts. While the code author can go read documentation on whatever you just told them about, they might not recognize why it applies. A short explanation or a link to a relevant article or specific Stack Overflow post (rather than a general manual) will be a shortcut to help them learn.

**Give an example of what would be better**

If a section of a design is confusing, don’t just say “please make this more clear.” It’s hard to know what to do with that! Offer a couple of suggestions of what you think the author is trying to say.

**Be clear about what matters**

When you’re less experienced, it can be hard to calibrate the advice you’re given. Some things are vitally important, some are nice to have, and some are just personal preference. Annotate your advice so it’s clear. Some examples:

* “Use parameterized queries here instead. You’re opening yourself up to SQL injection attacks—a malicious user could drop our database!”
* “The way you’re approaching this will work fine, but we prefer to avoid the singleton pattern. Here’s a link to the section of our style guide that talks about why.”
* “I’d recommend one bigger microservice here rather than two small ones—I think it’ll be easier to maintain. But I don’t feel strongly; your call.”
* “This is just a nitpick, but all of the other spellings in this file are in American English, so let’s call this one *organization* instead of *organisation.*[**7**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn106)

**Choose your battles**

John Turner, a software engineer who has written about code review for the Squarespace engineering blog, [recommends reviewing code in several passes](https://oreil.ly/3VLqa): first high-level comments, then in increasing detail. As he points out: “If the code doesn’t do what it’s supposed to, then it doesn’t matter if the indentation is correct or not.” This advice works for RFCs too: if your first comment is that the author is solving the wrong problem, it’s not helpful to leave a hundred technical suggestions.

**If you mean “yes,” say “yes”**

Make it clear whether you consider your comments to be blocking or not, and whether you’re otherwise happy with the change. Call out the good as well as the bad. In particular, explicitly say “This looks good to me” on design documents. Code review tends to end by clicking a button to say that you believe the change is safe to merge. When there are a lot of reviewers, though, each one may be hesitant to approve until the others have weighed in. If you have no objections, say so.

Remember that engineers who are earlier in their careers may find you intimidating or be reluctant to question your suggestions even when they think you’re wrong. Think about how to make your comments friendly, approachable, and, well, *human*. If the pull request or RFC needs a lot of work, the most constructive approach might not be to bury your colleague in comments: consider setting up time to talk or pair with them instead.

**Coaching**

The last form of individual teaching I’m going to talk about is coaching. Whereas *mentoring* involves sharing your personal experiences, *coaching* teaches people to solve problems for themselves. It can be a slower process, but your colleague will learn much more by making their own connections than they will from you giving them the answers.

Coaching’s a set of odd skills that sound pretty straightforward but take time and deliberate effort to learn. You shouldn’t expect to be immediately good at it! Here are the three big skills you’ll need:

**Asking open questions**

Open questions are the ones that can’t be answered with “yes” or “no”: they yield much more information. Try to dig into the problem rather than starting with a solution. Ask questions that help your coachee identify and unpack aspects of the problem that they might not already have considered.

**Active listening**

Reflect back what you’ve heard, to make sure that you’ve really understood and to let the person hear how you frame what they’ve said. The feeling of being understood can be powerful and help people feel less alone. Your framing can give your colleague a new way to describe what they’re working through, helping them come up with new solutions.

**Making space**

Leave enough space and silence for the coachee to reflect. If you tend to reflexively jump in when there’s a silence, count to five in your head before speaking again so the other person can process.

When you’re new to being a coach, it feels very strange to not just *help.* If you have the answer, shouldn’t you just *give* it? No. As management consultant Julia Milner [warns in her TEDx talk](https://oreil.ly/ghwkC), you can’t know every detail of the situation. When you provide a solution, the coachee is likely to reflexively respond with what Milner calls a “Yes, but…,” a reason your advice can’t work. Instead, she says, good coaching involves drawing out their own best ideas, providing them the space to reflect, and helping them take their own journey to a solution that works for them.

**Scaling Your Teaching to a Group**

Teaching one person at a time is great for that one person, but it’s a slow way to spread information. You can scale your teaching by creating materials for a class.

Putting together a class takes a ton of effort. There’s a high up-front cost for the first time you teach it, but you’ll amortize that cost every time you teach it again. Just like individual teaching, your class should have a specific goal for what your students will know or be able to do at the end. Include exercises or some way for the students in the class to practice what they’ve learned.

If you want to make your class asynchronous, consider making something that students can use at their own pace. A great example of this kind of teaching is a *codelab*: a guided tutorial that takes the student step by step through building something or solving exercises.[**8**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn107)

I used to work on a project that was perpetually underfunded and survived on a rotating cast of volunteers, interns, and people on short-term residency programs. Many were new to the company or even the industry. Dropping them straight into our codebase (an intimidating networking library) would have made them run screaming, so we created and documented a learning path.

On day one, we had them send two pull requests: one to add a joke to our repository of jokes, and one to add their name to the list of people on the team. We would find a reason to do a little back-and-forth on this, so they’d get used to the code review process. On day two, we’d have them build and run a tiny client and server we’d created just for this purpose. They’d watch it running in production, and look at its UI and the logs and metrics it was exporting. Then they’d make a local change to the library—adding a new log message, maybe, or even tweaking the logic—and deploy it to prove to themselves that the code still worked and that they could see their change in the monitoring data. It was very effective. By the time they got to making actual changes on week two, they weren’t scared of the code. It was just code.

When the majority of your contributors only stay for three months, you need to get them up to speed quickly. We used this same well-documented learning path for every new contributor, and it didn’t take long for it to repay our investment.

**Being a Catalyst**

You can scale your classes further by teaching other people to teach them. Different teachers have different styles; embrace that. Let your new teachers begin to own their own classes: they should have access to edit the slides and exercises, or they should have your blessing to create their own variants. Shadow them and give honest feedback: they want to learn. Once *they* start teaching other people to teach the class, it has life without you. (At this point I usually slink away and remove myself from the teaching rotation.)

If your class is applicable to all your engineers, try to add it to an onboarding curriculum or internal learning and development path—all of your new hires will learn what you want them to know without you having to find a way to reach them. If your company doesn’t already have this kind of learning culture, you can have a huge impact by advocating for an onboarding process, evangelizing learning paths, or setting up a framework that makes it easy to create codelabs.

**Guardrails**

Think of the railings you might find along a cliffside walking path. They’re not for *leaning* on, but they’re there to steady yourself when you need them. A small stumble won’t doom you: the railings will stop you going over the edge. Guardrails encourage autonomy, exploration, and innovation. We all move faster when the going is safe. In this section we’ll look at some ways you can add guardrails for your colleagues, first individually and then at scale.

**Individual Guardrails**

You can provide guardrails by reviewing code, designs and changes, and by offering support through scary projects.

**Code, design, and change review**

I’ve already described how code and design review can be great teaching tools. There’s a third type of review that can be effective, too: *change management.* That’s the process of writing down exactly what you’re going to do before you do it and having someone else agree that you’ve described the right set of steps. It’s like code review, but for command lines or clicking buttons in the right order.

You can use code, design, and change reviews as powerful guardrails to help your colleagues. When someone knows their work will be reviewed, it’s easier for them to feel confident working independently. They know there will be a check to make sure they don’t cause an outage or spend months building an architecture that can’t work. The guardrail helps them avoid dangerous mistakes.

If you want to be a good guardrail, don’t ever rubber-stamp changes. Read carefully: every line of code, every section of a design, every step of a proposed change. Here are some categories of problems you should look for:

*Should this work exist?*

What problem does your colleague intend to solve? Are they using a technical solution to solve a problem that should have been solved by talking to someone?

*Does this work actually solve the problem?*

Will the solution work? Will users be able to do what they need and what they expect? Are there errors or typos? Any bugs or performance issues? Does the design propose using a system in a way that won’t work?

*How will it handle failure?*

How will the solution handle weird edge cases, malformed input, the network randomly disappearing, load spikes, or whatever else can go wrong? Will it fail in a clean way, or will it corrupt data or take a user’s money without giving them the service they’ve paid for? How will you discover problems?

*Is it understandable?*

Will other people be able to maintain and debug new code or systems? Are the components or variables named intuitively? Is the complexity contained in a well-chosen place?

*Does it fit into the bigger picture?*

Does the change set a precedent or create a pattern you might not want other people to copy? Does it force other teams to do extra work for future changes? Is this a risky change that’s scheduled at the same time as a high-profile launch?

*Do the right people know about it?*

Is everyone copied on the change who should be? Are there names attached to any actions that need to happen, or is there a lot of passive voice where it isn’t clear which team is doing what? Do the people involved know what is expected of them?

As a reviewer, be open to the idea that you don’t know everything. Ask questions and be constructive. A good guardrail is not an arbitrary gatekeeper: you’re on the same side as the person you’re keeping safe, and you want them to succeed.

**Project guardrails**

If you’ve ever stretched to take on a difficult project, you’ll know that it’s a great way to build skills, but it’s also nerve-wracking! You’re more likely to fail, because you’re doing something that’s hard for you. So it’s nice when you have a more experienced colleague who has done a project of this size or shape before: they can help keep you safe. That doesn’t mean they’ll do the work for you or protect you from all possible mistakes, but they’ll let you know if you’re getting close to a disaster you won’t be able to recover from. That’s what it means to act as a project guardrail.

I remember leading a project that was a real stretch for me. It had more moving parts than anything I’d done before, more stakeholders, and *way* more politics. Every week when I met with my team lead, he’d ask questions about the project: “Just out of interest, how were you planning to balance these two conflicting business priorities?” Of course, I didn’t have a plan: I hadn’t noticed the problem creeping up. But the questions were enough to put me back on course, and I was able to suggest paths forward. Although I didn’t realize it at the time, the team lead was acting as a guardrail. He was making sure I’d noticed that I was walking close to a cliff edge, and if I didn’t have a good idea for what to do, he was ready to coach.

Being a project guardrail isn’t just for less experienced folks: you can play this role for any colleague who is leading a project or taking on a difficult task. Even very seasoned people can use support in a project that’s using a new skill set. If someone asks you to be a mentor or adviser on a project, they’re probably hoping for at least a little guarding.

A guardrail can offer support as well as safety. Lara Hogan suggests [being specific](https://oreil.ly/gAC0X) about how you can help with the project, like promising to review designs or advocate for ideas with upper management, as well as being explicit about when and how your colleague should ask you for help. She suggests lines like “Shoot me an email if person B is unresponsive to you for three days; I can be your muscle there.”

**Scaling Your Guardrails to a Group**

You can’t personally review *every* change and support every project, and you’ll just slow everyone down if you try. Let’s look at how you can add guardrails for your team or organization without getting in everyone’s way.

**Processes**

Rather than individually teaching your coworkers the right thing to do, you can write down a standard set of steps and convince the organization to follow them. For example, what’s the “right way” to launch a new feature at your company? Your prelaunch process might include answering questions like:

* Do we need security approval?
* How much notice should we give the marketing team or customer support?
* Should we ship behind a feature flag?
* Is there standard monitoring, eventing, and documentation we should add?
* Do we need to tell other teams to expect extra load?

And many more! As the company grows and the organization gets more complex, there will be more ways for a launch to cause an outage or public relations mess. And so a process is born.

Opinions about processes vary. Some people will be delighted to have clear steps and the safety that comes from standardization. Others will insist that people should *think* instead of mindlessly following protocols and that checklists and approvals just slow them down. Nobody’s wrong; it’s a trade-off. But the bigger the company, the more likely you’ll need *some* sort of structure that helps people do the right thing without having to ask the same questions every time.

Here are some other examples where adding a process or checklist might be helpful:

* Responding to a major outage or security incident
* Sharing and agreeing on RFCs or designs
* Adopting a new technology or language
* Making and recording decisions that cross multiple organizations

In general, aim to make processes as lightweight as possible. If you add a complicated procedure with lots of boilerplate, central approval, and long waiting periods, it’s not going to be a good guardrail: people will just sneak around it. Make the right way the easy way.

**PROCESS PREAMBLE**

Here’s the introduction I wrote for a process FAQ document at work. Feel free to use it if it’s helpful for you too.

There are a lot of questions about how <topic> should work. It’s hard to find a balance for how prescriptive to be with processes like this.

* If you write nothing down, most people hate that and complain that they don’t know how to do anything.
* If you write down guidelines, people interpret them as law and argue that they’re wrong because they don’t cover edge cases.
* And if you write down every edge case, you end up with a three-ring binder of policy and legalese, and it probably still won’t cover every situation. And everyone still hates it!

This document attempts to give mostly correct answers to some frequently asked questions. These answers will not apply perfectly in every situation. Think twice before discarding them, but if they don’t make sense for a situation you’re in, do the thing that makes sense instead. All guidelines are wrong sometimes. (If these guidelines are wrong *a lot,* propose a change.)

When in doubt, think hard about the other humans involved in what you’re doing, assume they’re reasonable people trying their best, and also be a reasonable person trying your best.[**9**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn108)

**Written decisions**

Here’s another way you can make it easier to do the right thing: make a decision once and write it down, so people don’t have to have the same argument again and again. Written decisions remove a little decision fatigue from people’s lives: the rules say we usually do X, so that’s what we’ll do!

Here are four examples:

**Style guides**

As Google’s [style guide site explains](https://oreil.ly/gkUmT), a style guide for a project is “a set of conventions (sometimes arbitrary) about how to write code for that project. It is much easier to understand a large codebase when all the code in it is in a consistent style.” The word *style* here covers a lot of ground, from naming conventions to error handling to which language features are OK to use. By making the decisions once and writing them down, you save teams from having the same “do we use lower camel case or snake case for variable names?” arguments for every new project. You’ll end up with more consistent code, too.

**Paved roads**

Some companies document their set of standard, well-supported technologies and recommend (or mandate) that teams don’t step off that “paved road.” I like the format popularized by the [Thoughtworks Tech Radar](https://oreil.ly/FUHQP), marking technologies as “Adopt,” “Trial,” “Access,” and “Hold.”

**Policies**

Companies can make rules: for example, “Every team should run a retrospective after an outage.” If the rule is enforced, breaking it could be seen as failing to do one’s job, with implications for performance reviews. Use policies sparingly. It’s hard to account for all the edge cases—and there will always be edge cases. Besides, if there are too many policies, people just won’t remember all the things they’re supposed to do.

**Technical vision and strategy**

A technical vision or strategy (see [Chapter 3](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch03.html#creating_the_big_picture)) gives a clear direction within which teams can choose their own paths to solving problems.

**Robots and reminders**

Software consultant Glen Mailer says he looks for ways to make it as easy as possible for people to remember to do the right thing. This means putting the right solution in their faces—sometimes literally! He gave an example of a workplace where everyone was supposed to track their project time using timesheets. Of course, people often forgot until someone came up with a solution: they stuck a timesheet grid and a pen on the exit door at head height. When anyone pushed the door to leave, the timesheet would be in front of their eyes—much harder to forget.

If you’re trying to introduce a process or a written decision, see if there are ways you can (gently) put it in people’s faces. Even better, have an automated system do the right thing so humans don’t have to. Some examples:

**Automated reminders**

Rather than always reminding someone that it’s their week to follow the release process, set up automation that puts it in their calendar or DMs them about it. The reminder should include a link to the process.

**Linters**

Have a code linter enforce as much of your style guide as it can, so reviewers don’t have to.

**Search**

Make sure that any search for how to do something brings up the *right* way to do it, even if that means updating all of the “wrong” documents to have headers that point to the right place.

**Templates**

If all RFCs are supposed to have a security section, make sure there’s an easy RFC template and that it includes a security section.

**Config checkers and presubmits**

Can you add automation that automatically runs unit tests, or runs safety checks on configs before committing them? Google’s data center safety system, SRSly, is a great example: it allows setting guardrails like “No more than 5% of servers may be rebooted at once” and “Don’t decommission a server for this system if the on-call for it recently got paged.”[**10**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn109)

**Being a Catalyst**

Creating robots, policies, and processes that reinforce your message scales further than being a guardrail for individual colleagues. But they all still rely on *you* doing something. If you really want a message to stick, you need everyone to believe in it and care about it. You want your organization to get to a state where it would be considered weird to do something else. The most effective guardrail is also the most difficult to put in place: culture change. Unless you can make the guardrail part of your culture, you’ll always be chasing compliance.

Most tech companies now have code review and write tests. But that wasn’t always the case! All of the guardrails that we take for granted today were introduced by people who cared enough to argue for why the change was worth the time. If you’re introducing a culture change, be patient. It takes a lot of time and dedicated effort to make everyone behave in a different way, but it’s the only way you’ll ever be able to stop pushing the process along manually.

Here are some ways you can make your culture change journey easier:

**Solve a real problem**

The culture change should be closely aligned with what the organization needs. Expect any proposal to be confronted with a lot of “why” questions. Have good answers that aren’t just aspirational: really, what does the business get out of this?

**Choose your battles**

Rather than a process for design review, try to instill a *respect* for design review and trust that teams will make their own choices about what form works for them. Offer some easy defaults, but don’t get hung up on whether everyone’s following exactly the same process.

**Offer support**

Your processes and automations should support the change you want: they should make it easy to do the right thing.

**Find allies**

Don’t try to change the culture on your own. Ideally, your allies will include high-level sponsors and influencers in the organization you’re trying to change. Consult the *shadow org chart* you mapped out in [Chapter 2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch02.html#three_maps).

**Opportunity**

The last type of good influence we’re going to look at is finding people the experiences they need to grow. People learn by *doing*, even more than they do from teaching or coaching or advice. Every project or role is a chance for visibility, relationships, and résumé lines, all of which can lead to further opportunities. Let’s look at how you can send those experiences to your colleagues, both individually and at scale.

**Individual Opportunities**

As a senior person, you will have many occasions where you can help other people find opportunities to grow. You’ll be able to directly offer projects and learning experiences through delegation. But you’ll also be able to suggest people for assignments, promote their work, or connect them with information that can help them.

**Delegation**

Delegation means giving part of your work to someone else. When you delegate, you’re usually not just tossing someone a project and walking away: you’re invested in the outcome. That might mean you’re tempted to micromanage or to handle all of the difficult parts of the project yourself. But when you hand over the work, *really* hand it over. As [Lara Hogan says](https://oreil.ly/PJ06s), your colleagues won’t learn as much if you only delegate the work after you’ve turned it into “beautifully packaged, cleanly wrapped gifts.” If you instead give them “a messy, unscoped project with a bit of a safety net,” they’ll get a chance to hone their problem-solving abilities, build their own support system, and stretch their skill set. A messy project is a learning opportunity that’s hard to get otherwise.

Target the level of difficulty to the person you’re delegating to—don’t throw organizational chaos to a new grad! But when you’re looking for someone to delegate to, think beyond the most obvious people. Anyone who can do an A+ perfect job on a project isn’t going to learn from it.[**11**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn110) Instead, try to find someone who will find the work a bit of a stretch but manageable with support. Promise them that support. You might need to give them a little push to help them see that the work is within their reach: they might not yet see themselves as a project lead, an incident commander, etc., but the fact that *you* see them like that can be a tremendous boost to their confidence. Describe the guardrails you can provide for them and explain why you think they can handle the project.

Be warned that, when you delegate, you’re *not going to get a clone*. (Sorry, we don’t have that technology yet.) Inevitably the person you’ve delegated to is going to take a different approach than you would have. Be a guardrail, coach them, and ask questions, but inhibit the urge to step in. So long as they’re going to achieve the goals, let them do it their way. I absolutely love how Molly Graham, most recently the chief operating officer at Quip, frames handing off work as [“giving away your Legos”](https://oreil.ly/ltqKy):

*There’s a lot of natural anxiety and insecurity that the new person won’t build your Lego tower in the right way, or that they’ll get to take all the fun or important Legos, or that if they take over the part of the Lego tower you were building, then there won’t be any Legos left for you. But at a scaling company, giving away responsibility—giving away the part of the Lego tower you started building—is the only way to move on to building bigger and better things.*

One key Lego-relinquishing behavior to watch out for is that you should be *redirecting* questions about the project to the other person, not *proxying* the information. That is: when someone asks you a question about the project, you may think that you have the current state. But answering the question makes you the point of contact, potentially undermining the colleague to whom you handed off the project. You might also have the wrong answer! Instead, give visibility to the person who took over the project: note that they’re the expert and owner for the topic and show that you trust them to make decisions. You’ll give the project owner a connection to someone new and any extra opportunities that come out of that connection. The next time the interested person has a question about the project, they’ll know where to go. And the project will be off your plate.

**Sponsorship**

Sponsorship is using your position of influence to advocate for someone else. It’s more active than mentorship: you’re deliberately unlocking opportunity for other people, not just giving them advice. It takes more work, too: if you want to be a great sponsor, you need to know what your colleague will benefit from and what opportunities they’re looking out for. You’re investing your time and social capital in their growth.

Rosalind Chow, associate professor of organizational behavior and theory at Carnegie Mellon University, [has described what she calls “the ABCDs of sponsorship”](https://oreil.ly/Ndm87):

*Amplifying*

Promoting your colleague’s good work and making sure other people know about their accomplishments

*Boosting*

Recommending them for opportunities and endorsing their skills

*Connecting*

Bringing them into a network, giving them access to people they wouldn’t otherwise be able to meet

*Defending*

Standing up for them when they’re unfairly criticized; changing any negative perceptions of them

The opportunities that you can offer through sponsorship may seem like small ones, but they lead to greater things. If you recommend someone to lead a small project, you’re setting them up to later be seen and chosen for a bigger one. Every time you give someone a shout-out, comment on their work, or even retweet them, you’re signal-boosting their good work and making sure the other people in your network know about them and think of them when opportunities arise. If someone in another team is being a superstar, make sure their boss knows. If you’re in a company that writes peer reviews, a review is a great way to make sure the good work goes on the record.

Who should you sponsor? Look for people who want opportunities to grow and who you trust to do good work. You’re spending your social capital by recommending them, so don’t waste that by sending opportunities to people who don’t actually want them, or who won’t put in the effort. Sponsor colleagues who have untapped potential, who are worth investing in. Watch out for in-group favoritism, though, a cognitive bias that can let you evaluate other people more favorably if they’re like you. [In the words of Mitch Kapor](https://oreil.ly/bbO1B), founder of Lotus, cofounder of the Electronic Frontier Foundation, and cochair of the Kapor Center for Social Impact: “We talk about the meritocracy of Silicon Valley, when really it’s a mirrortocracy, as people tend to hire people who look like themselves at greater rates than other sectors.” Pay attention to who you’re recommending or helping, and make sure you’re not accidentally only sponsoring people who look like you. It’s surprisingly easy to do.

**Connecting people**

Even if a role or project isn’t yours to delegate and you aren’t being asked to make a recommendation, you can still offer opportunities just by knowing that they exist. As a staff+ engineer, you’ll probably have broader context than other engineers you work with: as I described in [Chapter 2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch02.html#three_maps), you’ll be spending more time just *knowing* *things.* Keep an ear out for opportunities that will help your colleagues. You can remember that a conference call for papers is open, that a team lead role is opening up, or that a new internal training program is offering a skill someone is trying to learn. By connecting people with information, you expand their options.

**Scaling Your Opportunities to a Group**

Some of the catalytic types of influence I’ve already described give people visibility and opportunities to learn leadership and teaching skills. But here’s one more way you can scale opportunity: sharing the stage.

**Share the spotlight**

As the most senior person on a team or on a project, there’s going to be a lot of things that you do best. That might make it tempting to jump on every difficult problem. But while it might feel amazing to be the resourceful and knowledgeable senior person who’s leading the team to greatness every day, what you’re actually doing is overshadowing the rest of the team and preventing them from growing.

Instead, translate your superstardom into helping *everyone* do better work. Let other people do work that’s not as good as you would have done it, so long as it’s *good enough.* That’s how they learn. Sharing the stage includes delegation, but it can also mean making space: letting other people notice that the work needs to be done so they can build leadership skills by picking it up or learning to delegate for themselves.

Here are some ways you can make sure you’re sharing the spotlight:

* If someone asks a question in a group meeting, leave a gap or explicitly hand off the floor to another person on your team.
* Add a less senior colleague to a meeting you go to, and let them speak about their work.
* Invite a less senior colleague to review designs or code that connects to their work, and be clear that their opinion matters.

**WARNING**

While being the face of every project limits your team, the opposite extreme can have problems too. If you prefer to delegate *everything*, make sure you’re aligned with your management chain about how you work. Most managers will expect some direct execution and visible accomplishments from their staff engineers. Don’t put yourself into so much of a support role that nobody’s quite sure what you do.

**Being a Catalyst**

You can take steps to keep opportunity and sponsorship flowing even when you’re not there. While most workplaces have some concept of mentorship, sponsorship might not already be well understood. Look for ways that you can teach your colleagues to look beyond the obvious candidates to suggest for opportunities. Some ideas include: advocating for an [inclusive interview process](https://oreil.ly/PdxHq), inviting a speaker on implicit bias, or setting the culture that open team roles are posted on an internal job board.

The best way you can be a catalyst in the industry is by empowering your colleagues to become great engineers who do great things. As you offer opportunities that turn your midlevels into seniors and then into staff engineers, teach *them* to offer advice, teaching, guardrails, and opportunities to the people who will follow them. The staff engineers you grew will grow the staff engineers who follow them. Your leadership will keep going.

**WARNING**

For someone to be promoted to your level, they don’t need to be as good as you are *now*: they need to be as good as you were when you were first promoted to the level. If you keep seeing people join your level who aren’t as capable as you are, don’t snark about lowered standards: think about whether that means you’ve grown.

At the start of the chapter I listed three reasons to improve your coworkers’ skills: getting more done, keeping your technology up to date, and improving the industry. Here’s a fourth: other people’s growth is your growth. If you can delegate, you’ll be able to take responsibility for bigger, more difficult problems, handing off parts of them to the rest of your group. The more your colleagues can do, the more you can do. [As Bryan Liles says](https://oreil.ly/ScVHa), “How you can get pushed up is by building a whole bench behind you.”

At some point, you might look at the people who are doing the job you used to do and realize you’ve gone up a level. What do you do then? In [Chapter 9](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch09.html#whatapostrophes_nextquestion_mark) we’ll look at what’s next.

**To Recap**

* You can help your colleagues by providing advice, teaching, guardrails, or opportunities. Understand what’s most helpful for the specific situation.
* Think about whether you want to help one-on-one, level up your team, or influence further.
* Offer your experience and advice, but make sure it’s welcome. Writing and public speaking can send your message further.
* Teach through pairing, shadowing, review, and coaching. Teaching classes or writing codelabs can scale your teaching time.
* Guardrails can let people work autonomously. Offer review, or be a project guardrail for your colleagues. Codify guardrails using processes, automation, and culture change.
* Opportunities can be much more valuable than advice. Think about who you’re sponsoring and delegating to. Share the spotlight in your team.
* Plan to give away your job.

[**1**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn100-marker) “Watch out, there’s a bear behind you” is also acceptable unsolicited advice.

[**2**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn101-marker) Don’t name or shame the person you think needs advice, though.

[**3**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn102-marker) *Resilient Management* (A Book Apart)

[**4**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn103-marker) 2014. A good year.

[**5**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn104-marker) X couldn’t start without Y and Y couldn’t start without X, and so neither could start.

[**6**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn105-marker) I’ve sometimes tried to anchor reviewers by noting the context in the change description. For example: “I’m new to this language, so if something looks weird, it’s probably not a deliberate stylistic choice! I welcome nitpicking and advice on what’s idiomatic.”

[**7**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn106-marker) A little window into my life. :laughing:

[**8**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn107-marker) The [Kotlin Koans](https://oreil.ly/XXtus) are a great example, and fun to work through. Google also has a ton of [great codelabs](https://oreil.ly/OqlOA).

[**9**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn108-marker) This is also just generally good life advice.

[**10**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn109-marker) Check out Christina Schulman and Etienne Perot’s [entertaining talk](https://oreil.ly/k8ohr) on SRSly, including its origin story: automation accidentally sent all of the disks in a data center to be erased at once. As they noted, if you ask efficient automation to do something *stupid*, it will do so very efficiently.

[**11**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#ch01fn110-marker) This pattern is common in recruiting mails: “Come do exactly the thing you’re currently doing, but at another company.” There are times when that will work (we’ll explore motivations for changing jobs in [Chapter 9](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch09.html#whatapostrophes_nextquestion_mark)), but the most successful recruitment I’ve seen is for roles that offer people a step up, something slightly scary.