**Chapter 6. Why Have We Stopped?**

As the project’s driver, you’re responsible for getting everyone safely to the destination. But there are a lot of reasons your journey might stop early. You could run into roadblocks: accidents, toll booths, or a country road full of sheep. You might find that you’ve lost your map, or that the various people in the car disagree about where you’re going. Or you might just realize you should be going somewhere else.

**The Project Isn’t Moving—Should It Be?**

In [Chapter 5](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#leading_big_projects), we talked about starting projects. Now it’s time to look at ways they can halt. We’ll start with two kinds of temporary stops you might encounter when something’s going wrong: getting blocked by something, and getting lost.

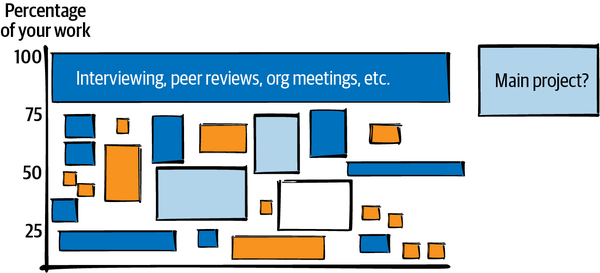
Then we’ll look at ways you might intentionally stop the journey. Sometimes this means declaring victory too soon, but it’s also sometimes just time for the journey to end, whether it’s reached its destination or not.

This is a good opportunity to note that, as a leader in your organization, you can help projects that you’re *not* leading too. Sometimes the best use of your time will be to set aside what you were doing and use pushes, nudges, and small steps (and, OK, sometimes major escalations) to get a stalled project moving again. As Will Larson [says](https://oreil.ly/LKc0I), this small investment of time can have a huge impact:

*A surprising number of projects are one small change away from succeeding, one quick modification away from unlocking a new opportunity, or one conversation away from consensus. With your organizational privilege, relationships you’ve built across the company, and ability to see around corners derived from your experience, you can often shift a project’s outcomes by investing the smallest ounce of effort, and this is some of the most valuable work you can do.*

For consistency, the rest of this chapter will assume that you’re leading the project that has stopped. However, a lot of these techniques will work just as well if you’re stepping in to help.

Don’t forget, though: seeing a problem doesn’t necessarily mean you should jump on it. As I said in [Chapter 4](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch04.html#finite_time-id00004), you need to *defend your time*. Don’t get into the situation depicted in the time graph in [Figure 6-1](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#itapostrophes_possible_to_fill_your_tim), where you’re caught up in so many side quests and assists that you have no time for the work you’re accountable for. Be discerning! Choose the opportunities where your help is most valuable, and then take deliberate action, with a plan for stepping away again afterward.



**Figure 6-1. It’s possible to fill your time with helping and nudging along projects and leave no space for the main project you promised to do.**

Let’s start with the first set of reasons projects stall: they get blocked.

**You’re Stuck in Traffic**

In a perfect world, teams would be autonomous and never have to think about each other’s work. In reality, any big project is going to span multiple teams, departments, and job functions. Even a single person’s procrastination can sometimes be enough to miss an important deadline. And when the project is a migration or a deprecation, success might depend on work from *every* other engineering team, with many opportunities to get blocked along the way.

No matter what kind of blockage you’re dealing with, you’ll use some of the same techniques:

*Understand and explain*

You’ll debug what’s going on and understand the blockage. Then you’ll make sure everyone else has the same understanding of what’s happening.

*Make the work easier*

You’ll work around blockages by not needing as much from the people you’re waiting for.

*Get organizational support*

It’s easier to get work prioritized when you can show that it’s an organizational objective. You’ll demonstrate the value of the work so that you can get that support. And sometimes you’ll escalate to get help getting past a blockage.

*Make alternative plans*

Sometimes the blockage just isn’t going to go away, and you’ll use creative solutions to succeed. Or you’ll accept that the project just can’t happen in its current form.

Let’s look at how to use these techniques when you’re blocked in various ways: waiting for another team, a decision, an approval, a single person, a project that’s not assigned, or all of the teams involved in a migration.

**Blocked by Another Team**

We’ll start with a classic dependency problem: your project is on track but you need work from another team, and it’s *not happening*. If you’re lucky, a leader of that team is telling you up front what’s going on and when they’ll be ready to go. If you’re less lucky, the team has stopped replying to your emails and you’re piecing together what’s going on. Waiting on them is frustrating. They have all the information you have, and they know there’s a launch date! Why don’t they care? Go find out.

**What’s going on?**

If a team you depend on is not delivering what you need, there’s almost certainly a great reason why. Three likely reasons are misunderstandings, misadventures, or misaligned priorities.

*Misunderstandings*

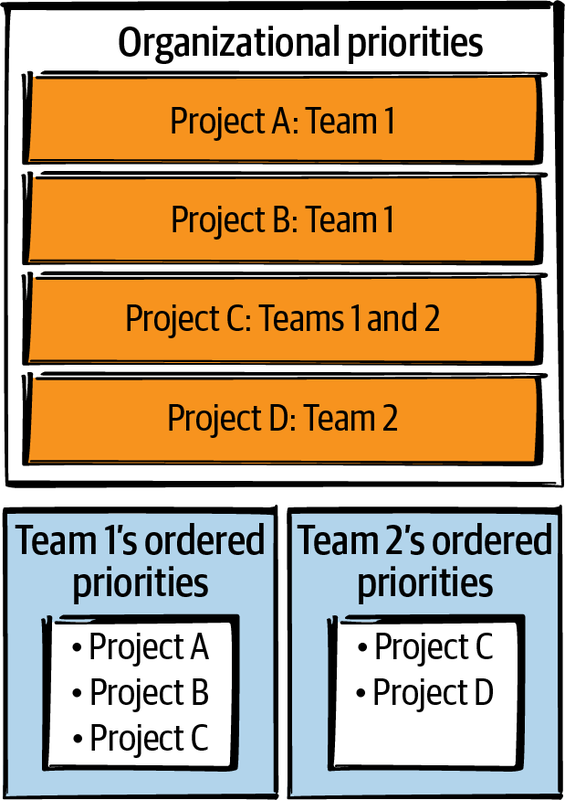
Even in organizations with clear communication paths, information can get lost. One team thinks it’s obvious that something needs to happen by a specific date. The other team has no idea that there’s a deadline, or has taken away a different interpretation of what they’ve been asked to do.

*Misadventure*

Life happens. Someone quits, or gets sick, or needs to take abrupt leave. The team you depend on is understaffed, overloaded, or blocked by their own downstream dependencies. It might be impossible for them to meet the deadline, no matter how important it is.

*Misalignment*

Maybe the team has impressive velocity—just not on your project. Even if you’re working on something vital, the other team may have an even higher priority. Take a look at [Figure 6-2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#teams_with_misaligned_prioritiesdot_if). Project C is Team 2’s highest priority, and they’re focused on it above any other work they have to do. But it’s only Team 1’s third-highest priority! They’ll get to it if they have time.



**Figure 6-2. Teams with misaligned priorities. If each team works on its most important project, Team 2 will likely be waiting on Team 1.**

**Navigating the dependency**

Here are those four techniques in action:

**Understand and explain**

Start by understanding why the other team isn’t moving. Do they not understand what’s needed? Is something in their way? Understanding means talking to each other. If DMs or emails aren’t working, take it to a synchronous voice conversation–yes, this means a meeting. If the team is hard to reach, going through back channels might help: hopefully you’ve built a bridge with someone on or near the team. (See [Chapter 2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch02.html#three_maps).)

Explain why the work is important, and spell out what you’re hoping they’ll deliver and by when. Give them another chance to tell you whether that’s realistic, or whether there’s something else they can do that would solve your problem.

**Make the work easier**

If you need something from a team that doesn’t have time to provide it, try asking them for something smaller. That might mean a single feature you absolutely need, instead of the several you’d really like. If they’re blocked by dependencies of their own, think about whether there’s anything you can do to help unblock them: if you can solve their problem so they can solve yours, then everyone wins! Sometimes you can end up taking a side quest down a chain of dependencies until you find a small nudge that will let everyone start moving again.

Alternatively, you might be able to offer to do part of the work for them, for example by having another team write code and send it to them for review. Be aware that this offer might not be as helpful as you intend: supporting an untrained person through making a change in a difficult codebase, for example, can often take more effort than doing it yourself. Don’t be offended if the team doesn’t take you up on the offer.

**Get organizational support**

If the team you’re waiting for is doing something they think is higher priority, find out whether they’re right. If priorities are unclear, ask your organization’s leadership to adjudicate on which project should “win.” Be respectful and friendly, but ask enough questions to understand. I hope it goes without saying, but if your organization considers your work to be *less* important, you should leave the other team alone to focus on the thing they’re doing.

But if your blocked project is genuinely more important, this may be a case for escalation to a mutual leader. No matter how frustrated you are, deliver the unemotional facts: explain what you need, why it’s important, and what’s not happening. Consider discussing the situation with your project sponsor or manager before escalating. They may have alternatives to suggest, or may be willing to have some of those conversations for you.

**WARNING**

Escalating doesn’t mean raising a ruckus or complaining about the other team. It means holding a polite conversation with someone who has the power to help, and trying to solve a problem together. Keep it constructive.

**Make alternative plans**

If the team’s really not going to be available, you’ll need to find another way around the blockage. That might mean rescoping your project, choosing a different direction, or shipping later than you’d intended to. It is what it is. Make sure you talk with your stakeholders and project sponsor about any change of dates and make sure they understand what you’re blocked on. They may have ideas for unblocking that haven’t occurred to you.

**Blocked by a Decision**

Should the team take path A or path B? Should they design for a single, specific use case or try to solve a broader problem? How should they lay out their architecture, APIs, or data structures? So much depends on the details, and it’s difficult to make progress without them. You could design for maximum flexibility, but that’s expensive, and you want to avoid overengineering. But it will feel terrible if you have to throw your solution away in a year because you took the wrong path. So you ask your stakeholders for specific requirements, use cases, or other decisions. And you get…nothing. How can someone ask you to build something and not know what they want?!

**What’s going on?**

When you’re waiting for a decision from someone else, it’s tempting to think of their work as easier than yours. They just need to decide what they want, right? Then you have the actually difficult work of building it. I’ve seen this bias a lot for decisions that need to come from outside engineering. “Why won’t the product team just…?” As with many uses of the words “just” or “simply,” the answer will be complicated. Product or marketing (or anyone else) can’t make a snap decision any more than you can. Or they may be waiting for information from someone *they* depend on before they can decide.

They might also not understand what you’re asking. This is especially common when you have engineers explaining an engineering problem to nonengineers. I’ve seen an engineer ask a stakeholder, “Do you want X or do you want Y?” and hear “Yes, that would be great!” in response. That stakeholder isn’t intentionally being obtuse! Different contexts and different domain languages mean that they don’t see much distinction between the two things you’ve asked for, so it’s impossible for them to make an informed choice.

**Navigating the unmade decision**

When your decision is blocked on someone else’s decision, have empathy: the decision is likely not any easier where they’re standing. But, of course, you don’t want to wait forever, so here are some techniques for making progress when someone’s having trouble deciding.

**Understand and explain**

Remember that you’re on the same side. Rather than seeing the person you’re waiting for as an obstacle, see if you can navigate the ambiguity together. Understand what information or approval they need to make a decision and try to help them get it. Make sure they know why it matters and what can’t happen until and unless they decide. Explain the impact on the things *they* (not you!) care about. If they’re blocked, understand who or what they’re waiting for.

**Make the work easier**

Think about how you’re asking the question, and build a mental model of how the other person is receiving it. Really try to get into their head: if you were them, how might you interpret the words? Are there easy misunderstandings? Think about whether you can use pictures, user stories, or examples to reframe the question. The decision might be easier once everything clicks for them.

Sometimes the problem is that the decision doesn’t have a clear owner and the various stakeholders just can’t agree. If the decision you need is blocked by conflict, consider playing mediator, helping each side understand the other’s point of view and find a solution that suits them both. If your decision makers are blocked by *their* decision makers, give them the information they need in the format they need to take back. Take the time to give them talking points that the people *they’ll* need to interact with will understand. And if some parts of the decision are more important than others, explain what’s hard or expensive to reverse later and what doesn’t really matter.

**Get organizational support**

If you’re still not able to get a key decision made, talk to your project sponsor about what they’d like to do. They may have some ideas about paths forward, or be in some rooms you’re not in where they can push for a decision to happen.

**Make alternative plans**

Moving a project ahead with a major decision still unmade tends to be unsatisfying and complicated: when you have to stay open to all kinds of future directions, the solutions can’t be as sleek and elegant. They’re often more expensive too. But sometimes keeping your options open is the best choice you have available.

Alternatively, sometimes it’s OK to make your best guess about the right path and take the risk that you’re wrong. If you do guess, document the trade-offs and the decision.[**1**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn77) Make sure the decider knows that you’re guessing and understands the implications of the direction you chose. Spend some time thinking about the worst that could happen and what you might later wish you’d done, and mitigate those risks in any way you can.

Finally, be realistic. If your organization can’t make this crucial decision, is working around it going to be enough? Will you just run into the same problem for the next decision? It may be time to accept that you just can’t continue the work for now. If that’s the case, talk to your project sponsor, tell them what you’ve tried, and make sure they agree that it’s not possible to proceed.

**Blocked by a Single $%@$% Button Click**

We’ve all been there, and it’s incredibly frustrating. You’re waiting on a team or an approver that just needs to check a box, deploy a config, or review a five-line pull request. It’ll take them 10 minutes! Why don’t they just click the freakin’ button?!

**What’s going on?**

I used to be on a team that configured load balancing for everyone else at the company. Our documents said we needed a week’s notice to add a new backend to our balancers but, truthfully, each request took about half an hour. We needed to provision extra capacity, change configurations, and restart services. It was a frequent task, and a straightforward one for someone who knew what they were doing. Other teams knew that what we were doing wasn’t rocket science, so we regularly got requests like “We’re launching tomorrow, so please set up our load balancing today.” And, usually, we didn’t.

Why did we insist on a week’s notice? Because these configurations weren’t our only work. Hundreds of teams used our banks of balancers, and load balancing was just one of the four critical services my team supported. We didn’t want to react constantly: we wanted to plan out our weeks, and to make these configuration changes in batches rather than continually restarting services every time. As a result, we had little sympathy for people who came in hot and angry about why we hadn’t done the thing they’d told us about only a few hours ago. Our team motto became “lack of planning on your part is not an emergency on mine.”[**2**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn78)

Once again, the world looks very different when you’re on the other team! Your request is only one small part of someone’s work, one little block on their time graph. It may be only a single button click for them, but they have a lot of other people sending them unclicked buttons too. They might be understaffed and struggling. They might be finding edge cases as they work to improve their operations. I’ve seen teams realize that their process for handling requests can’t scale, and dedicate some team members to building a better approach, making the team short-staffed and even slower in the short term.

Bear in mind that other teams often take on accountability when they give approval. When you ask a security team to “just click the button” to approve your launch, or a comms team to approve your external messaging, you’re asking them to share (or entirely own!) responsibility if something goes wrong. They shouldn’t do that lightly.

**Navigating the unclicked button**

If the team is just not doing the work, then you can use most of the mechanisms for handling blocked dependencies that I discussed earlier in the chapter. But if it turns out that there’s a standard way of interacting with them and you didn’t use it, you’ll need a different approach. If you’re not in a real hurry, maybe you can just wait and let the team get to the work when they get to it. But if you have a real deadline, you can’t exactly go back in time and use the process. Here are those techniques again:

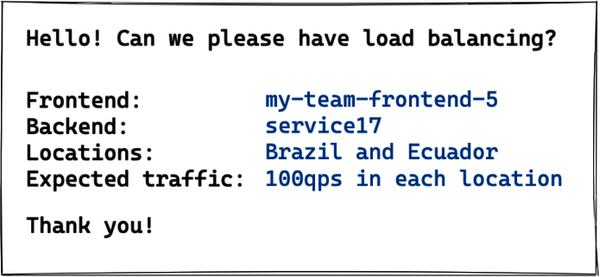
**Understand and explain**

If you really need to skip the queue, try asking. Be as polite and friendly as possible: you’ll get better results by apologizing rather than yelling at the busy team.

If someone goes out of their way to help you, say thank you. In companies that have peer bonuses or spot bonuses, there’s already a structure for saying thank you: use it. If the team is located together, can you send them a thank-you gift, like fancy tea or chocolate? At least include them in the list of people you thank in the launch email. Afterward, you won’t be remembered as the team that asked for something at the last minute, you’ll be remembered as the team that built bridges and made friends.

**Make the work easier**

Just like with a team dependency, make the thing you need as small as possible. Structure your request so it’s easy to say yes to, with as little reading needed as possible. (See [Figure 6-3](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#a_request_with_all_of_the_information_l) for an example.) If you have access to other kinds of requests the team gets, look at what problems tend to arise: what information is missing, what’s complicated, what’s controversial. Try to make your ticket be one of the easy ones, the kind that the person doing the work picks up first because they won’t need to think too hard about it.



**Figure 6-3. A request with all of the information laid out and not much reading to do.**

**Get organizational support**

If you *really* can’t get help any other way and the deadline matters, you may need to ask for help skipping the queue. Be warned: escalating may build bad blood with the team you’re looking for help from: nobody likes having their director or VP asking them to move a request to the front of the line. Mitigate any potential animosity as much as possible by being clear that you understand why the team has this process, and that you’re apologetically asking to circumvent it just this once.

**Make alternative plans**

It’s possible that having (or making) a connection with someone on that team will be enough to raise the priority. (This isn’t how the world *should* work, but often it’s how it *does* work.) If none of these options do work, though, you may just be waiting. Let your stakeholders know you’ll be delayed.

**Blocked by a Single Person**

If waiting for a team is frustrating, it can be even worse when a whole important project is waiting on a *single person.* The work is allocated, it’s on someone’s desk, and they’re just not doing it.

**What’s going on?**

I once had a project blocked by a coworker who needed to write a couple of short Python scripts to solve a problem. Weeks passed, and the scripts didn’t materialize, and my urge to just write them myself got stronger every day. My colleague always had a good excuse: there was an outage, or he’d needed the day off, or his computer had a hardware problem that needed to be fixed. After a few weeks of being frustrated at his lack of progress, I realized he wasn’t slacking: he was intimidated. He’d come from an operations role and had been used to the kind of interrupt-driven work where you bounce from fire to fire, rarely getting a block of focus time. This project was his opportunity to begin writing code, but he didn’t really believe he could do it, and so he couldn’t get started.

The reasons your colleague gives you for being blocked aren’t necessarily the real ones. The person could be intimidated, stuck, or oversubscribed. They could be stressed out about something in their personal life that makes it impossible to focus, or they might be getting messages from their leadership about what’s most important—and are nervous to tell you that your project isn’t on that list. Or maybe they didn’t understand what you were asking for the first two times you explained it, and they’re too embarrassed to ask a third time. All of these causes look the same from the outside: the person’s just not doing the work.

**Navigating a colleague who isn’t doing the work**

When someone is having trouble getting their work done, there are some ways you can help. Let’s be clear: you’re not this person’s boss or therapist, and it’s not your role to fix their procrastination. But you may be able to get the outcome you want and, if it’s a less experienced person, take the opportunity to teach them some skills. Here’s what you can do:

**Understand and explain**

See if you can learn more about what’s going on with your colleague. Next time you talk with them, don’t just accept their promise that the work will be ready in another week: dig a little deeper. You might not be able to find out what’s really going on, but you’ll get a sense for whether it’s something you’ll be able to help with.

Be very clear about why the work is necessary. This is particularly useful when you’re waiting on a more senior colleague, who (presumably!) is making deliberate judgments about relative importance and not just procrastinating. Describe the business need, and show what can’t happen until their work happens. If they have a heavy workload, ask them to let you know if they won’t be able to get the work done, so you’ll have time to find an alternative. Consider setting an earlier go/no-go deadline after which you’ll both understand that they’re not going to be ready in time and you should find an alternative approach.

**Make the work easier**

Make it as easy as possible for the person to do the thing you want. That might mean adding structure, breaking the problem down, or creating milestones: when the project is too difficult, sometimes even thinking about how to break it up can be too difficult.[**3**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn79) Don’t obfuscate the request by making the person search through a document for action items assigned to them or intuit what you’re asking them to do: just spell out what you need. Look at Brian Fitzpatrick and Ben Collins-Sussman’s “Three Bullets and a Call to Action” technique for asking for something from a busy executive: it works here too!

**WHAT ARE YOU ASKING FOR?**

I love the “Three Bullets and a Call to Action” method that Brian Fitzpatrick and Ben Collins-Sussman outline in their book, *Debugging Teams* (O’Reilly). As they write: “A good Three Bullets and a Call to Action email contains (at most) three bullet points detailing the issue at hand, and one—and only one—call to action. That’s it, nothing more—you need to write an email that can be easily forwarded along. If you ramble or put four completely different things in the email, you can be certain that they’ll pick only one thing to respond to, and it will be the item that you care least about. Or worse, the mental overhead is high enough that your mail will get dropped entirely.”

If your colleague is overwhelmed, this may be an opportunity for coaching. Reassure them that what they’re working on is legitimately difficult but learnable. Help them, but try not to take over.[**4**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn80) Ask questions, answer questions, and help them find their way.

If your colleague seems willing to do the work but is having trouble getting started, see if you can work with them on it. This was the solution for the colleague I mentioned earlier: pairing on the scripts got him past the intimidating first steps of the work and able to continue on his own. Pairing can also take the form of whiteboarding together or sitting down together to edit a document at the same time. This last one can sometimes be a good approach when you’re waiting on your manager: you can meet for a one-on-one meeting, suggest that they use the time to do the thing you need, and stay there with them so you’re available to answer any questions they have along the way.

**Get organizational support**

While you should try other approaches before escalating, ultimately someone is not doing their job, and that’s a people management issue. Having a difficult conversation with their manager is uncomfortable, but if the other person is the reason a project is going to fail, you’re not doing your job either if you ignore it. Just like I’ve said for other situations, escalating doesn’t mean complaining: it means asking for help. If your colleague is blocked because they’re working on something their manager thinks is more important, for example, talking to that manager may be the only way to adjust the priorities.

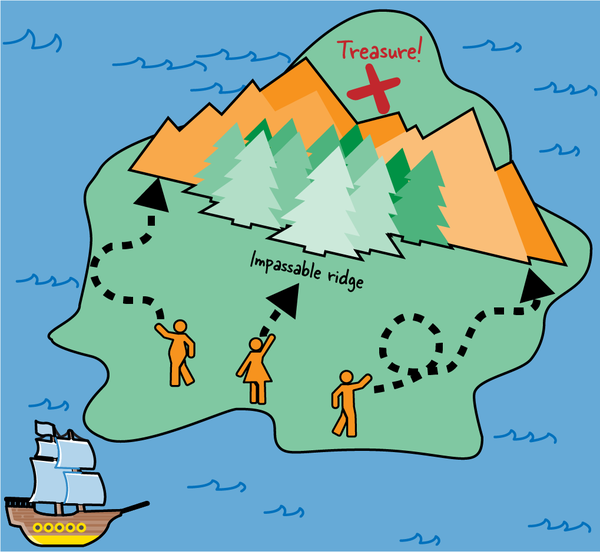
**Blocked by Unassigned Work**

What about when the work isn’t assigned to anyone? When a group of teams are working together to solve a problem, sometimes there’s an effort that everyone agrees needs to happen but that isn’t on anyone’s roadmap. It’s too big for any one engineer to tackle, it needs a lot of dedicated time, or it will involve creating a new component that will need ongoing ownership and support: it needs a team to own it. Maybe there are several different teams that consider themselves *involved* in the work—they’d turn up to a meeting about it and have many opinions on any RFC—but none of them intend to commit code to achieve it.

**What’s going on?**

This is an example of a *plate tectonics* problem (see [Chapter 2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch02.html#three_maps)). Every team has clear boundaries on what they’re responsible for. Strong boundaries can be great—they keep everyone focused—but now there’s some crucial foundational work that doesn’t belong to anyone. We saw this situation in the SockMatcher scenario in [Chapter 3](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch03.html#creating_the_big_picture). Many engineers cared about the architecture: Geneva’s working group to discuss the problems drew many participants. But the challenges were too big for anyone to solve as a side project. Unless it gets dedicated attention, the work is just not going to happen.

If nobody is assigned to do the work, there’s limited value in breaking it down, optimizing it, or making plans: you’re stuck until someone owns it. It’s very common for engineers to keep trying to solve an organizational blocker like this by putting more effort into the adjacent technical work. (See [Figure 6-4](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#unless_you_can_find_a_way_past_the_impa) for an illustrative treasure map: that impassable ridge is your lack of staffing!)



**Figure 6-4. Unless you can find a way past the impassable ridge, it doesn’t really matter which of the first three paths you take. But teams invest a lot of time debating which doomed path to take.**

But unless the organizational problem is solved, no amount of designs and clever solutions will help. The organizational gap is the crux of the problem. If you can’t solve that, you’re wasting your time.

**Navigating the unassigned work**

When you’re blocked on work that doesn’t seem to be anyone’s responsibility, there are a few things you can do.

**Understand and explain**

It’s not always obvious that work is unowned, particularly when there are a lot of teams that are closely adjacent to it. Since you’re the person most interested in its success, don’t be surprised if other people think that *you’re* the owner now. Have a whole lot of conversations, follow the trail of clues, and figure out exactly what’s going on. Interpret the information and draw explicit conclusions. Then write them down! (See the following sidebar.) Keep it brief and make clear statements: just like I said about design documentation in [Chapter 5](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#leading_big_projects), wrong is better than vague, and you won’t uncover any misunderstandings unless you remove the ambiguity.

**ROLLUP**

Denise Yu, a manager at GitHub, describes [“the art of the rollup”](https://oreil.ly/5U1wf): summarizing all of the information in one place to “create clarity and reduce chaos.” It’s a versatile technique, useful in any situation where there’s a ton of backstory and several different narrative threads and where some people might not have kept track of what’s going on.

Aggregating the facts that go into the rollup is a great way to build your knowledge and make sure you understand what’s going on. But writing it all down also might mean you synthesize new information that nobody had articulated before. Perhaps Alex says, “The new library will give us authentication for this end point.” And later Meena tells you, “We won’t be able to upgrade to the new version of the library until Q3.” You can write down both facts, but also the interpretation, “We won’t have authentication until at least Q3.”

That conclusion might seem obvious with all of the context, but if nobody has reached it before, it may come as a shock to some. By spelling it out, you give everyone an opportunity to react to the information and course-correct.

**Make the work easier**

If you have time to mentor, advise, or join the team doing the work, mention that in your rollup. Directors may be more inclined to build a team around an existing volunteer, and the chance to learn from a staff+ engineer can be an incentive that they can offer to other team members they’re inviting to join.

**Get organizational support**

Your highest-value work on this project will be advocating for organizational support and an owning team. Before you set out to find a sponsor for the work, make sure you’ve honed your elevator pitch and can explain why the problem is worth your organization’s time. You’ll want your sponsor not just to believe you, but to be able to justify the effort to their peers and leadership if they need to.

**Make alternative plans**

If there’s been a credible promise of staffing, be patient and give it time to happen: project staffing involves readjusting teams, which managers and directors (understandably) prefer to be deliberate about. But if you can’t get a commitment for anyone to own the work, or if it’s always “next quarter” without any specifics about where the staffing will come from, that’s a sign that your project isn’t a high priority for the organization and should be postponed.

**Blocked by a Huge Crowd of People**

The last type of roadblock I’m going to talk about is when the project needs help from everyone: it’s the kind of deprecation or migration where *all* of the teams using a service or component need to change how they work. Chances are, not all of them will want to.

I’ve worked on many, many software migrations, and I know how frustrating they can be. You have good reasons for moving off an old system, you know exactly what needs to happen and you’ve communicated plenty, but other teams are ignoring your emails. When you chase them, they say they’re busy—but you’re busy too! Why can’t they see that this work matters?

**What’s going on?**

Every team and every person has their own story. One might be broadly in favor of the migration but just doesn’t have time. Another might be opposed to the change: they might prefer the old system, or feel that the new one is missing some feature they care about. A third group might not have feelings either way, but they’re just tired of a constant, demoralizing stream of upgrades, replacements, and process changes that they don’t seem to get any benefit from.

The people pushing *for* the migration and the people pushing *against* are all being reasonable. But being stuck in a half-migration situation isn’t good for anyone: teams need to spend time on supporting both the old system and the new system, and new users may have to spend time understanding which to use, particularly if the migration stalls and seems at risk of being canceled.

**Navigating the half-finished migration**

The half-migration slows down everyone who has to engage with it. This is a place where a staff engineer can step in and have a lot of impact. Here are some ways you can get everyone over the finish line.

**Understand and explain**

The narrative of a migration can get lost, and I’ve sometimes heard a migration framed as “that infrastructure team just wants to play with new technology” when the reality is an immovable business or compliance need that infrastructure team doesn’t love either. Equally, I’ve heard “that product team doesn’t care about paying down technical debt; they only want to create new features,” when the team in question has already spent half of the quarter reacting to other migrations and are desperate to get started on their own objectives. Understand both sides and help tell both stories. Show why the work is important and also why it’s hard. Be a bridge.

**Make the work easier**

Once again, the key to convincing other people to do something is to make the thing easy to do. In general, lean toward having the team that’s pushing for the migration do as much extra work as possible, rather than relying on every other team to do more. If there’s any step you can automate, automate it. If you can be more specific about exactly what you want each team to do—which files you want them to edit, for example—spend extra time providing that information. Also, it hopefully goes without saying, but the new way has to *actually work* without forcing the teams to jump through hoops.

Try to make the new way the default. Update any documentation, code, or processes that point people toward the old path. Identify people who might encourage the old way and ask for their help. If you have the organizational support for it, consider an allow-list of existing users, or some friction to make the old way harder to adopt. One approach I’ve seen is to keep the old way working fine so long as new users write “i\_know\_this\_is\_unsupported” or similar as part of their configuration.

Show the progress of the work. I worked on one project where I needed to get hundreds of teams to change their configs to use a different endpoint. When I shared the graph showing how many had been done and how many were left to do, people were more eager to do their part to get the numbers down. Something in our brains really likes seeing a graph finish its journey to zero![**5**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn81)

Some teams will be genuinely too busy to move, or will have a use case that isn’t quite supported by your automation. There might also be some components that nobody owns anymore, so there’s no team to update them. Can you pair with the team, or make the change for them?

**Get organizational support**

The migration will be easier if you can show that this work is a priority for your organization. The corollary to this, of course, is that the work *should* actually matter. If teams are snowed under by changes, your organization should be prioritizing the most important ones and finishing the first set of changes before starting the next set. If you can’t convince your leadership that your migration should be reflected on the organization’s quarterly goals or list of important projects, maybe it’s a sign that this is not where you should be spending time right now.

**Make alternative plans**

By the end of the migration, you may need some creative solutions. If you have an organizational mandate and teams are still not moving, can you withdraw support for the old way, or even start adding friction along the path, for example by introducing artificial slowness or even turning it off at intervals? Don’t do this without strong organizational support (and don’t do it at all if it’s going to break things for your customers; be sensible). If the final teams are really refusing to move off the old system, can you make them its sole supporters and owners, so that it runs as a component of their own service? Last out turns out the lights, friends.

**You’re Lost**

Let’s move on to the second set of reasons you might have trouble making progress: you’re just lost. It’s not that you’re blocked by anything that you can see: you just don’t know the way. This might be because you don’t know where to go now, because the problem is too hard, or because you’re not sure if you still have organizational support for what you’re doing. You’ll use different techniques in each case.

**You Don’t Know Where You’re All Going**

Imagine 40 teams working on the same legacy architecture. There’s a single massive codebase, a decade of regrettable decisions, a tangled mess of data that’s owned by everyone all at once. Teams are scared to refactor existing code, so they bolt new features onto the outside. You’ve been tagged as the leader who is responsible for fixing it and you have a team dedicated to the work. There’s an organizational goal to “modernize the architecture.” It feels like you have a clear mandate to do the work! And yet…there are so many decisions to make. There are so many stakeholders, a huge number of comments on every document, too many voices in every meeting. It’s been a few months and you’ve made no real progress.

**What’s going on?**

We saw this with the SockMatcher scenario in [Chapter 3](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch03.html#creating_the_big_picture): it’s difficult to solve a problem that half the company cares passionately about! Everyone has an opinion. Everyone’s sure they know the right thing to do. In this example, there’s almost certainly a group of people advocating for creating microservices. Another faction might want feature flagging and fast rollbacks so that it’s safer to make changes. A third wants to move to event-driven architecture. A fourth doesn’t care what happens so long as the underlying data integrity problems are resolved. Those are just four of the many fine suggestions.

A huge group tackling an undefined mess will almost inevitably get stuck in analysis paralysis. Everyone agrees that *something* should be done, but they can’t align on what. You can’t steer this project because it’s just a bunch of ideas: there’s no project to steer.

**Choosing a destination**

You can’t start finding a path to the destination until you’re very clear about what that destination is. Here are some approaches for choosing it:

**Clarify roles**

With a group this big, the leader can’t be just one voice in the room. Be clear about roles from the beginning. Be explicit that you want to hear from everyone, but that you’re not aiming for complete agreement: you will ultimately make a decision about which direction to take. If you don’t feel that you have the power to name yourself the decider, ask your project sponsor or organizational lead to be clear that they will back you up. If you don’t have that kind of organizational support, you may not be set up to succeed.

**Choose a strategy**

Unless you know where you’re going, you have little chance of getting there. Set a rule that, until you all agree on exactly what problem you’re solving, nobody is allowed to discuss implementation details.[**6**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn82) If you can, choose a small group to dig into the problems and create a technical strategy. Emphasize that any strategy will, by definition, make trade-offs, and that it can’t make everyone happy. You’ll pick a small number of challenges, and leave the other real problems unsolved for now. [Chapter 3](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch03.html#creating_the_big_picture) has lots more on how to write a strategy: set the expectation that it will not be a short or painless journey.

**Choose a problem**

If you’ve been assigned engineers who are eager to start coding in the service of the goal, it can be frustrating (and politically unpopular) to say that you need to spend time on a strategy first. If you really don’t have time to evaluate all of the available challenges and rank the work by importance, choose *something*, any real problem. Set expectations that you won’t allow the group to get diverted by the other (very real) issues, but that you fully intend to return to them after solving the first one. Once again, wrong is better than vague: any deliberate direction will probably be better than staying frozen in indecision.

**Choose a stakeholder**

One way you can choose a problem to solve is by choosing a stakeholder to make happy. Rather than solving “the shared datastore stinks and we need to rethink our entire architecture!” can you solve “one team wants to move its data elsewhere”? Reorient the project around getting *something* to *someone*. Aim to solve in “vertical slices”: first you help one stakeholder complete something, and then another. Progressing in *some* direction can help break the deadlock and clarify the next steps. Once you’re showing some results, consider revisiting the idea of creating a strategy and having a big-picture goal.

**You Don’t Know How to Get There**

What if you know exactly where you’re going? The destination is well understood and there aren’t blockers in your path, but you’re still not getting there. You’re not sure how to solve the next problem in front of you, or the project is huge and you’re not even sure which problem you’re supposed to solve next. I mentioned last chapter how self-fulfilling imposter syndrome can be: if you’re finding the work difficult, that can become a vicious cycle that makes you have less capability for tackling it. Maybe you’re avoiding thinking about it, but the longer you ignore the project, the worse it feels.

**What’s going on?**

The project is just difficult! It might be that there are a massive number of topics to keep track of and you feel entirely out of your depth, particularly when something is going wrong. Or, there’s one impossible task, a technical challenge or an organizational hurdle, and you just don’t know how to get past it. If you haven’t seen this kind of problem before, it might take you time to even recognize what’s happening, and longer to find solutions.

**Finding the way**

The path forward is *unknown* but not *unknowable*! Here are some techniques to start finding your way:

**Articulate the problem**

Make sure you have a crisp statement of what you’re trying to do. If you’re struggling to articulate it, try writing it out or explaining it out loud to yourself. Notice places where you could be more precise about who or what you’re referring to or what *is* happening versus what *should be* happening. Refine your understanding by talking through the problem with anyone who’s willing to discuss it with you.

**Revisit your assumptions**

Is it possible that you’ve already assumed a specific solution and are struggling to solve the problem only within that context?[**7**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn83) Are you looking for a solution that’s an improvement on every axis when trade-offs might be acceptable? Are you dismissing any solutions because they seem “too easy”? Explaining out loud why you think the problem can’t be solved might help you discover some movable constraints you hadn’t noticed before.

**Give it time**

Have you ever been blocked by a coding or configuration problem that you just couldn’t crack, and then the next morning you could immediately solve it? Sleep is amazing. Vacations can do the same thing. I’ve found that if I take a few days away from a problem, I’ll almost always come back with better ideas, even if I haven’t thought about it in the meantime.

**Increase your capacity**

Trying to solve a problem in the tiny spaces between meetings will constrain the ideas you can have. Schedule yourself some dedicated time to really unpack the situation in your mind: it can take a few hours even to clear out the noise of whatever else you were thinking about. Aim to bring your best brain to that meeting with yourself: for me that means good sleep, nonstodgy food, plenty of water, and a room with good light and air. You know your own brain: do whatever makes you smart.

**Look for prior art**

Are you really the first person to ever solve a problem like this? Look for what other people have done, internally and externally. Don’t forget that you can learn from domains other than software: industries like aviation, civil engineering, or medicine often have well-thought-out solutions to problems tech people think we’re discovering for the first time!

**Learn from other people**

Talking through the problem with a project sponsor or stakeholder can sometimes give you enough extra context or ideas to find your next steps. You can also learn from people outside your company. Most technical domains have active internet communities. See if there’s a place where experts on the topic hang out, and spend time there absorbing how they think and what keywords and solutions they mention that you don’t know about.

**Try a different angle**

Spark creative solutions by looking from another angle. If you’re trying to solve a technical problem, think about organizational solutions. If it’s an organizational problem, imagine how you’d approach solving it with code. What if you had to outsource it: who would you pay to solve this problem and what would they do? (Might that be an option?) If you weren’t available, who would this work get reassigned to and how would they approach it?

**Start smaller**

If you’re overwhelmed with tasks and it’s not clear what comes first, try solving one single small part and see if you can feel a sense of progress and make the rest of the work feel more achievable. Another angle is to ask yourself whether you really need to solve the problem *well*. Could a hacky solution be good enough for now? Or can you start with a terrible solution and iterate so that you’re not starting with a blank page?

**Ask for help**

While it might feel like your skills are the only thing standing between success and failure, you’re not alone. Ask for help from coworkers, mentors, or local experts. If you’re someone who hates asking for help, remember that by learning from other people’s experiences, you’re amortizing the time they had to spend learning the same thing: it’s inefficient to have you both figure out solutions from first principles.

**You Don’t Know Where You Stand**

Here’s the last form of being lost and, in some ways, it’s the scariest: you don’t know if your work is still necessary. A comment you heard at the latest all-hands meeting might make you nervous that a new initiative will derail yours. Maybe you’ve noticed that your manager or project sponsor is checking in less often, and seems less interested in your results. Or there was a company announcement that listed all of the important projects—and yours wasn’t on it. Yikes. Some of the people you’re working with seem disengaged: they talk about your project more as “if” than “when” and they’re prioritizing other work. Have they heard something you haven’t? Is the project still happening? Nobody is telling you anything. Are you still in charge?

**What’s going on?**

Organizational changes, leadership changes, and company priority changes can all affect enthusiasm for your project. If you have a new VP or director, they may not think the problem you’re solving is important, or, sometimes worse, they may think it’s so important that they’re solving it with a much bigger scope than you had taken on and a different lead. Your project genuinely could be at risk of being killed, and nobody’s thought to tell you. This missing communication is especially common if your leadership position is an unconventional one—for example, if you’re in an organization adjacent to the one that’s doing most of the work, or if you’re leading people who are more senior than you are. It’s easy to be forgotten when priorities change, or left out of meetings where decisions are happening.

Or it might not be that at all! Your project might be going so well that you’re just getting less attention from your leadership. When there are fires elsewhere, nobody checks in on a project that’s humming along. Silence could mean “keep doing what you’re doing.”

**Getting back on solid ground**

Continuing on the same path without knowing where you stand is just a recipe for stress, and you may be wasting your time. Here are some things you can do:

**Clarify organizational support**

Brace for the idea that you might not like the answers, and go find out what’s happening. Talk with your manager or project sponsor, explain what you’ve heard, and ask whether your project is intended to continue.

**Clarify roles**

If you’re the lead but find yourself hesitant to claim that title, or if you aren’t sure what you’re allowed to do, you need to formalize roles. The RACI matrix I described in [Chapter 5](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch05.html#leading_big_projects) might be a useful tool here, as might the role description document from [Chapter 1](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch01.html#what_would_you_say_you_do_herequestion). By the way, if you’re trying to run a project with a title like “unofficial lead,” that’s an invitation to fail—if you’re the lead and nobody else knows you are, you’re not the lead.

**Ask for what you need**

If you’re missing the authority, the official recognition, or the influence to do the work, who can help you find those things? It’s natural to want some reassurance that your project is still important. It’s fine to ask for a mention of the project at an all-hands meeting or to have it listed on the organization’s goals. You might not get what you want, but you definitely won’t if you don’t ask.

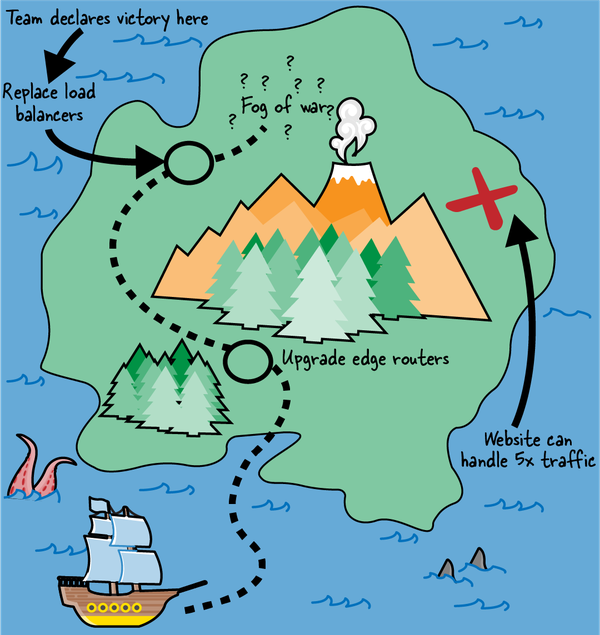
**Refuel**

It’s demoralizing to work on something that nobody else seems to care about. If you and your team are feeling low on energy, you may need to deliberately build that back up again. Refueling could mean setting new deadlines, building a new program charter, or having a new kickoff or off-site meeting: resetting with “Welcome to Phase 2 of the project” is somehow more motivational than “Let’s keep doing what we’ve been doing, but I swear it will be different this time.” If you can add a new team member or two who are raring to go, their enthusiasm can be enough to get the team moving again.

**You Have Arrived…Somewhere?**

There’s a third reason projects stop: the team thinks the project has reached its destination…yet somehow the problem is not solved.

I’ve seen many projects end like [Figure 6-5](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#the_team_declared_victory_and_went_home): just short of their goal. All of the tasks on the project plan are completed, the team members have collected their kudos and moved on to other things, yet the customer is still not happy.



**Figure 6-5. The team declared victory and went home—but there was another, better treasure they never got to.**

In this section we’ll look at three ways you might declare victory without actually reaching your destination: doing only the clearly defined work, creating solutions you don’t tell your users about, and shipping something quick and shoddy. Watch out for these end states!

**But It’s Code Complete!**

I feel like I’ve had this conversation a hundred times in my career:

*“I’m excited for the new foo functionality. When will it be available?”*

*“Oh, it’s done!”*

*“Amazing! How do I start using it?”*

*“Well…”*

The “Well…” is always followed by the reasons I can’t use it *today.* Foo still has hardcoded credentials; it’s only running in staging, not in production; one of the PRs is still out for review. But it’s *done*, the other person will insist. It’s just not *usable* yet*.*

**What’s going on?**

Software engineers often think of their job as writing software. When we plan projects, we often only list the parts of the work that are about writing the code. But there’s so much more that needs to happen to allow the user to *get* to the code: deploying and monitoring it, getting launch approvals, updating the documentation, making it run for real. The software being ready isn’t enough.

Heidi Waterhouse, a developer advocate for LaunchDarkly, once blew my mind with the observation that “nobody wants to use software. They want to catch a Pokémon.” A user who wants to play a video game doesn’t care what language the code is in or which interesting algorithmic challenges you’ve solved. Either they can catch a Pokémon or they can’t. If they can’t, the software may as well not exist.

**Making sure the user can catch a Pokémon**

As the project lead, you can prevent work from falling through the cracks by looking at the big picture of the project, not just which tasks were done. Here are some techniques you can use:

**Define “done”**

Before you start the work, agree on what the end state will look like. The Agile Alliance [proposes](https://oreil.ly/3Mray) setting a *definition of done*, the criteria that must be true before any user story or feature can be declared finished.[**8**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn84) This might include a general checklist for all changes—I’ve often seen PR templates include a section for explaining how the change was tested, for example—as well as a specific set of criteria for individual projects. Similarly, user acceptance testing lets intended users of a new feature try out the tasks they’ll want to do with it and agree that those features are working well. Even internal software can have user acceptance tests. Until those are completed and the user declares that they’re happy, nobody gets to claim the project is done.

**Be your own user**

Is it possible for you to regularly use what you’re building? Of course, this isn’t always going to apply, but if there’s a way for you to share your customers’ experience, take the time to do that. This is sometimes called [eating your own dog food](https://oreil.ly/qHyWM) or “dogfooding.”

**Celebrate landings, not launches**

Celebrate shipping things to users, rather than milestones that are only visible to internal teams. You don’t get to celebrate until users are happily using your system. If you’re doing a migration, celebrate that the old thing got turned off, not that the new thing got launched.

**It’s Done but Nobody Is Using It**

Have you ever seen a platform or infrastructure team spend months creating a beautiful solution to a common problem, launch it, celebrate, and then get frustrated that nobody seems to want to use it? They’re sure it’s better: it genuinely improves its users’ lives. But teams are still doing things the old, difficult way.

**What’s going on?**

The team is not thinking beyond the technical work. Unfortunately, internal solutions are often marketed like this: we create some useful thing. We give it a cute name. We (maybe) write a document explaining how to use it. And then we stop. The thing we created probably has potential users who would love it, but they have no way to know it exists; if they stumble across it, the name offers no hint as to what it does. Common search terms for the problem don’t suggest this solution. I call these “Beware of the Leopard” projects, from one of my favorite parts of Douglas Adams’s classic book *The Hitchhiker’s Guide to the Galaxy* (Pan Books)*.*

*“But the plans were on display…”*

*“On display? I eventually had to go down to the cellar to find them.”*

*“That’s the display department.”*

*“With a flashlight.”*

*“Ah, well, the lights had probably gone.”*

*“So had the stairs.”*

*“But look, you found the notice, didn’t you?”*

*“Yes,” said Arthur, “yes I did. It was on display in the bottom of a locked filing cabinet stuck in a disused lavatory with a sign on the door saying ‘Beware of the leopard.’”*

It’s like the creators of the solution are trying to hide it! The information *exists*—someone got to mark their documentation milestone as green—but it will never be found by anyone who doesn’t know what they’re looking for.

**Selling it**

Michael R. Bernstein has a [great analogy](https://oreil.ly/u65tj) for creating solutions and then not marketing them at all. He says it’s like a farmer planting seeds, watering, weeding, and growing a crop, and then just leaving it in the field. You need to harvest what you grew, take it to people, and show them why they want it. The best software in the world doesn’t matter if users don’t know it exists or aren’t convinced it’s worth their time. You need to do the marketing.

**Tell people**

You don’t just need to tell people that the solution exists: you need to *keep* telling them. A lot of migrations stay at the half-migration stage because engineers assume that users will just come find the software. Help them find it. Send emails, do road shows, get a slot at the engineering all-hands. Offer white-glove service to specific customers who are likely to advocate for you afterward. If you’re in a shared office, consider putting up posters! Get testimonials. Understand what anyone might be wary of, or unenthusiastic about, and make sure your marketing shows that you’ve thought about (and hopefully fixed) those problems. Be persistent and keep telling people until they start telling each other.

**Make it discoverable**

Whatever you’ve created, make it easy to find. This means linking to it from anywhere its intended users are likely to look for it. If you have multiple documentation platforms, make sure a search on any of them will end up at the right place. If your company uses a shortlink service, set up links for all of the likely names, including the misspellings and any hyphenations people are likely to guess.[**9**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn86)

**It’s Built on a Shaky Foundation**

The last type of “done but not *really* done” I’m going to talk about is one that can cause a lot of conflict. It’s when a prototype or minimum viable product has gone into production and the users can use it pretty well, but everyone knows that it’s hacked together. The user can catch a Pokémon, and the job is done: time to move on to the next thing, say the product managers! But the engineers know that the infrastructure won’t scale, the interfaces aren’t reusable, or the team is pushing appalling technical debt into the future.

**What’s going on?**

There might be good reasons to ship something as cheaply and quickly as possible. When there’s a competitive market—or a risk that there’s no market at all—it’s often more important to get *something* launched than for that something to be solid. But when the team has moved on, that cheap solution remains in place. The code may be untested—or untestable. The feature may be an architectural hack that everyone else now needs to work around.

I used to work in a data center, a long time ago, and one thing I learned there is that there’s no such thing as a temporary solution. If someone ran a cable from one rack to another without neatly cabling and labeling it, it would stay there until the server was decommissioned. The same is true for every temporary hack: if you don’t have it in good shape by the end of the project, it’s going to take extraordinary effort to clean it up later.

**Shoring up the foundations**

While you can get away with shipping shoddy software in the short term, it’s not a sustainable practice. You’re pushing the cost of the software to yourself in future. As a staff engineer, you have more leverage than most. Here are some ways you can advocate for keeping standards high.

**Set a culture of quality**

Your engineering culture will be led by the behavior of the most senior engineers. If your organization doesn’t have a robust testing culture, start nudging it in the right direction by being the person who always asks for tests. Or be the squeaky wheel who always asks, “How will we monitor this?” or the one who invariably points out that the documentation needs to be updated with the pull request. You can scale these reminders for the project with style guides, templates, or other levers. We’ll look at setting standards and causing culture change in [Chapter 8](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch08.html#good_influence_at_scale).

**Make the foundational work a user story**

Ideally, the organization agrees that shipping solutions isn’t just about features—it’s about preparing for the future. There’s a healthy balance of feature and maintenance work, and teams build the time for high quality into the cost of their projects. Even teams that build lightweight first versions to get early user feedback or take on technical debt to get a competitive product to market faster always return to improve whatever they’ve shipped.

Unfortunately, few organizations work in ideal ways. You can help by making sure the user stories for the project include any cleanup work you’ll need to do. You might frame this as part of user experience (nobody’s excited about a product that’s flaky or falls over a lot), or as laying the foundation for the next feature. If users file related bug reports or there are action items after outages, you can sometimes use those to justify the cleanup work. Focus the conversation back on the customer’s needs and show that the work has a real impact on them.

**Negotiate for engineer-led time**

If your company doesn’t have a regular culture of cleaning up as you go along, see if it’s possible for you to get momentum around having regular cleanup weeks. I’ve heard these called “fix-it weeks” and “tech debt weeks,” and I’ve also seen a fair amount of cleanup happen during engineering exploration time, like “20% time” or “passion project week.” Another option is to set up a rotation where one person on the team is always dedicated to responding to issues and making things better. Don’t get hung up on the name or whether something really “counts” as technical debt. The point is that it’s a dedicated time to do work that everyone in engineering knows is necessary.

**The Project Just Stops Here**

As you navigate the obstacles and deal with the difficult situations, you’ll get closer to your destination—or discover that it isn’t reachable after all. Let’s end the chapter by looking at four ways the project can come to a deliberate end: deciding you’ve done enough, killing it early, having it canceled by forces outside your control, and declaring victory.

**This is a better place to stop**

It’s possible to get to a point where further investment is just not worth the cost and to declare that the project is “done enough.” In [Chapter 2](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch02.html#three_maps) I talked about the local maximum: a team working on something low-priority because it’s the most important problem in their own area, while the team next door has five more important projects that they don’t have time to get to. Is your team polishing and adding features to something that’s really as done as it needs to be? It might be time to declare success and do something new. Before you do, look at the failure modes in the previous section and make sure you’re not abandoning unhappy customers, bailing once the technical work is done, or walking away from a half-migration. If you’re all good, congratulations on reaching the new destination!

**It’s not the right journey to take**

One of the biggest successes I ever saw celebrated could have been considered a failure. A team of senior people worked for months to create a new data storage system. Other teams were eagerly awaiting it. But as the work progressed, the team discovered that the new system wouldn’t work at scale. Rather than deny reality and search for possible use cases for what they’d created, they killed the project and wrote a detailed retrospective.

Was that a failure? Not really! Other teams were disappointed not to get the system they were hoping for. But the storage team couldn’t have discovered that the solution wouldn’t work without exploring it. If they’d realized that the technology couldn’t work for them and pushed on *anyway*, then *that* would have been a failure.

Have you heard of the *sunk cost fallacy*? It’s about how people view the investments of time, money, and energy they’ve already made: if you’ve already put a lot of time and energy into something, you’re more likely to stick around and see it through, even if that’s a bad idea. It can be difficult to break out of this frame of mind, but without a highly tuned sense of whether something’s still a good idea, you can stay on the wrong path for a long time.

Try to notice if you’re in an impossible situation; if so, bail out early. Pushing on with a doomed project is just postponing the inevitable and it prevents you from doing something more useful. Good judgment includes knowing when to cut your losses and stop. Consider writing a retrospective and sharing as much as you can about what happened. Few things are as powerful for psychological safety as saying “That didn’t work. It’s OK. Here’s what we learned.”

**The project has been canceled**

The company’s enthusiasm for what you’re all doing can change. Maybe the project has been dragging on too long or is more difficult than expected, and the benefit no longer justifies the cost. Maybe a new executive has a different direction in mind, the market has changed, or your organization is overextended and looking for initiatives to cut. For whatever reason, the project isn’t happening. Someone in your management chain takes you aside and tells you the difficult news. If you’re lucky, you find out before the rest of the company does.

Let’s start with the feelings, because this is a tough situation. It feels *bad*. Even if your team understands and accepts the reason for the cancellation—even if you all agree!—it’s jarring to suddenly drop the plans and milestones you’ve built. You all might feel like the work you’ve done has been for nothing. If you weren’t part of the decision to cancel, it can feel like a personal failure. You might feel angry, disappointed, and cheated or resent that a change that affects you so much has happened in a room you weren’t in. This may be a legitimate complaint: if managers make big technical decisions but leave the technical-track folks out of the room, that might be worth a conversation. But most of the time, these decisions are made at a higher level, by people who are looking at a much bigger picture and optimizing for something different than you are.

Work through your own feelings and acknowledge them. Talk it through with your manager or your sounding board. Try to understand the bigger picture and get as much perspective as you can. Then talk with your team or subleads. Tell them what’s happening, leading with the why. Give them time to talk about their own reactions to the news. Respect that they might be mad at you, the project lead, as the bearer of the bad news or the person who didn’t manage to “save” the project. It’s important that they hear it directly from you or another leader: don’t let them find out from the gossip mill, a mass email, or the company all-hands meeting.

Give yourself and your team a little time, then shut the project down as cleanly as you can. If you can stop running binaries, turn them off; if you can delete code, delete it. If you think there’s a real chance that the project will be restarted later, document what you can so that some future engineer can understand what you were trying to do. (Be realistic about the chances of this resurrection happening, though.) Consider a retrospective if there’s something to learn.

It’s not fair, but a canceled project can derail promotions or break a streak of great performance ratings. Do what you can to showcase everyone’s work. If your team members are moving on to other projects, make a point of telling their new managers about their successes and offer to talk with them or write peer reviews at performance review time. If someone’s on the cusp of promotion, emphasize that to their new lead, so they don’t have to build up a new track record from scratch.

Celebrate the team’s work and the experience you had together. It’s sad to dissolve a team that was working well together, but look for opportunities to work with those people again.

**This is the destination!**

Congratulations! You have done the thing you set out to do!

Before you celebrate, double-check that you have actually reached the destination. Are your measurable goals showing the results you want? Can the user catch a Pokémon? Are the foundations solid and clean? If you’re really at the end, it’s time to declare victory. Take the time to mark the occasion and make your success feel special. For some teams, celebrating means parties, gifts, or time off. There might be email shout-outs or recognition at all-hands meetings. Look for opportunities to give team members (and yourself!) visibility through internal and external presentations or articles on your company blog. And consider a retrospective: they’re not just for looking at things that went wrong. You can learn just as much from things that went right.

If there’s an aspect of your culture that you want to enforce, like people helping each other or communicating well, highlight the ways that it showed up during the project. Shout out people who went above and beyond or demonstrated any behaviors you’d like to see more of. A successful project can be a fantastic opportunity to celebrate the aspects of your culture that you most appreciate and show others what great engineering looks like.

And in the spirit of showing your organization how to do great engineering, let’s move on to [Part III](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/part03.html#part3) of the book, Leveling Up.

**To Recap**

* As the project lead, you are responsible for understanding why your project has stopped and getting it started again.
* As a leader in your organization, you can help restart other people’s projects too.
* You can unblock projects by explaining what needs to happen, reducing what other people need to do, clarifying the organizational support, escalating, or making alternative plans.
* You can bring clarity to a project that’s adrift by defining your destination, agreeing on roles, and asking for help when you need it.
* Don’t declare victory unless the project is truly complete. Code completeness is just one milestone.
* Whether you’re ready or not, sometimes it’s time for the project to end. Celebrate, retrospect, and clean up.

[**1**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn77-marker) Consider [Lightweight Architectural Decision Records](https://oreil.ly/BO1Kq) for showing why you made the choice you did.

[**2**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn78-marker) Looking back, I have much more sympathy for the other teams. The number of configurations needed to run a service in production was massive, and any team coming up to a launch had a lot to think about. Yes, they probably should have realized they needed load balancing ahead of time, but we were one of 15 prelaunch conversations they needed to have. And we should have figured out a way to replace our manual steps with something self-service for the most common cases. Perspective.

[**3**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn79-marker) If you’re the procrastinator, consider the calendaring trick I mentioned in [Chapter 4](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch04.html#finite_time-id00004): put the task you need to do in your calendar. If even understanding the first step is difficult, make a calendar block just for figuring out what the first step of the work will be, and schedule a separate block for working on that step. Give Future You the smallest possible tasks to do.

[**4**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn80-marker) It is *really* difficult to watch work that you care about being done badly, but try not to step in and do it for them. If it’s critical that the problem gets solved right now, see if you can work with the other person and get them to take each step, rather than just doing it yourself. Your colleague won’t learn to drive if you take the wheel.

[**5**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn81-marker) This only works if the graph is showing progress: if it’s clear no one is doing the work, it can backfire. But the social side of influencing people is such a great tool: “Everyone else is doing it, why aren’t we?”

[**6**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn82-marker) This will never entirely work, but keep trying. The more you can keep people out of the weeds, the more chance you have of succeeding.

[**7**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn83-marker) As [Leslie Lamport cautioned](https://oreil.ly/LO1jT), you should “specify the precise problem to be solved independently of the method used in the solution.” He wrote, “This can be a surprisingly difficult and enlightening task. It has on several occasions led me to discover that a ‘correct’ algorithm did not really accomplish what I wanted it to.”

[**8**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn84-marker) They also distinguish between “done” and “done-done,” and credit a [2002 article from author and agile coach Bill Wake](https://oreil.ly/2Tsnm) where he asks the enigmatic question, “Does ‘done’ mean ‘done’?”

[**9**](https://learning.oreilly.com/library/view/the-staff-engineers/9781098118723/ch06.html#ch01fn86-marker) Google used to have a project called Sisyphus, a name that’s memorable to some but an unlikely series of letters to others. I’ll always be impressed with whoever set up the shortlinks go/sysiphus and go/sisiphus as redirects to go/sisyphus. It’s a good security practice too; it prevents someone standing up a fake service at the misspelled place.