2

CHAPTER 2

Organizations

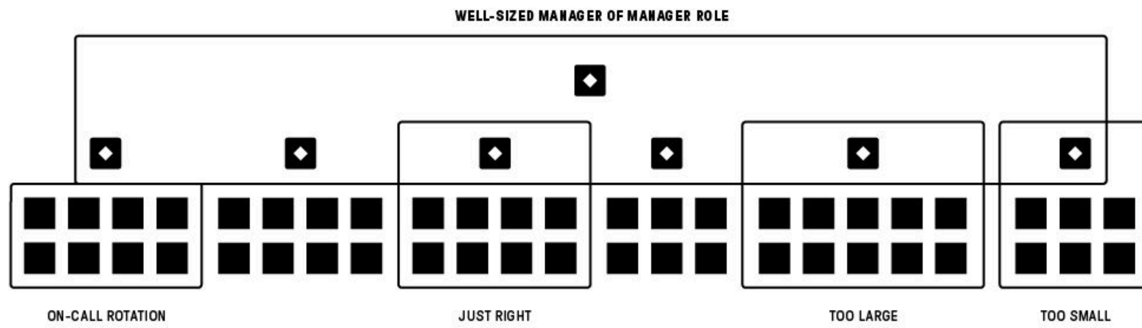


Figure 2.1
Sizing teams and groups of teams using sizing rules.

Organizations

An organization is a collection of people working toward a shared goal. Each organization is an exploration of the possible, undertaken together by the ten, the hundred, or the thousand. Initially, I was tempted to glibly write that sometimes organizations work, but the truly extraordinary thing is that all organizations work.

Some do indeed work better than others, and organizational design is the attempt to understand why some create such energy and others create mostly heat: friction, frustration, and politics. I believe that excellent organizations grow from consistently applying a straightforward process.

When I have a problem that I want to solve quickly and cheaply, I start thinking about process design. A problem I want to solve permanently and we have time to go slow? That's a good time to evolve your culture. However, if process is too weak a force, and culture too slow, then organizational design lives between those two.

This chapter covers the approaches to organizational design and evolution that I've found effective. If you're reading through and find yourself thinking that this sounds easy, I agree! The hard bit is keeping your courage up when circumstances get challenging.

2.1 Sizing teams

When I transitioned from supporting a team to supporting an organization, I started to encounter a new category of problems that I had never thought about. How many teams should we have? Should we create a new team for this initiative, or ask an existing team to take it on? What is the boundary between these two teams?

These questions were the gateway to the obscure art of organizational design. As I've gotten more exposure, I've come to believe that the fundamental challenge of organizational design is sizing teams. You'll find yourself sizing teams during reorganizations,[1] to accommodate growth from hiring, and when considering how to support new projects. It'll be an unusual month that you won't consider some aspect of team design.

While I'm skeptical that there exists a unified law of team sizing, I have iterated my requirements onto a useful framework that solves the majority of cases I encounter. That framework has in turn led to a standard playbook. Both are short, opinionated, and hopefully useful!

The guiding principles I use for sizing teams are:

• Managers should support six to eight engineers

This gives them enough time for active coaching, coordinating, and furthering their team's mission by writing strategies,[2] leading change,[3] and so on.

Tech Lead Managers (TLMs). Managers supporting fewer than four engineers tend to function as TLMs, taking on a share of design and implementation work. For some folks this role can uniquely leverage their strengths, but it's a role with limited career opportunities. To progress as a manager, they'll want more time to focus on developing their management skills. Alternatively, to progress toward staff engineering roles, they'll find it difficult to spend enough time on the technical details.

Coaches. Managers supporting more than eight or nine engineers typically act as coaches and safety nets for problems. They are too busy to actively invest in their team or their team's area of responsibility. It's reasonable to ask managers to support larger teams during the transition to a more stable configuration, but it is a bad status quo.

• Managers-of-managers should support four to six managers

This gives them enough time to coach, to align with stakeholders, and to do a reasonable amount of investment in their organization. On the other hand, it will also keep them busy enough that they won't be tempted to create work for their team.

Ramping up. Managers supporting fewer than four other managers should be in a period of active learning on either the

problem domain or on transitioning from supporting engineers to supporting managers. In the steady state, this can lead to folks feeling underutilized, or being tempted to meddle in daily operations.

Coaches. Similar to supporting a large team of engineers, supporting a large team of managers leaves you functioning purely as a problem-solving coach.

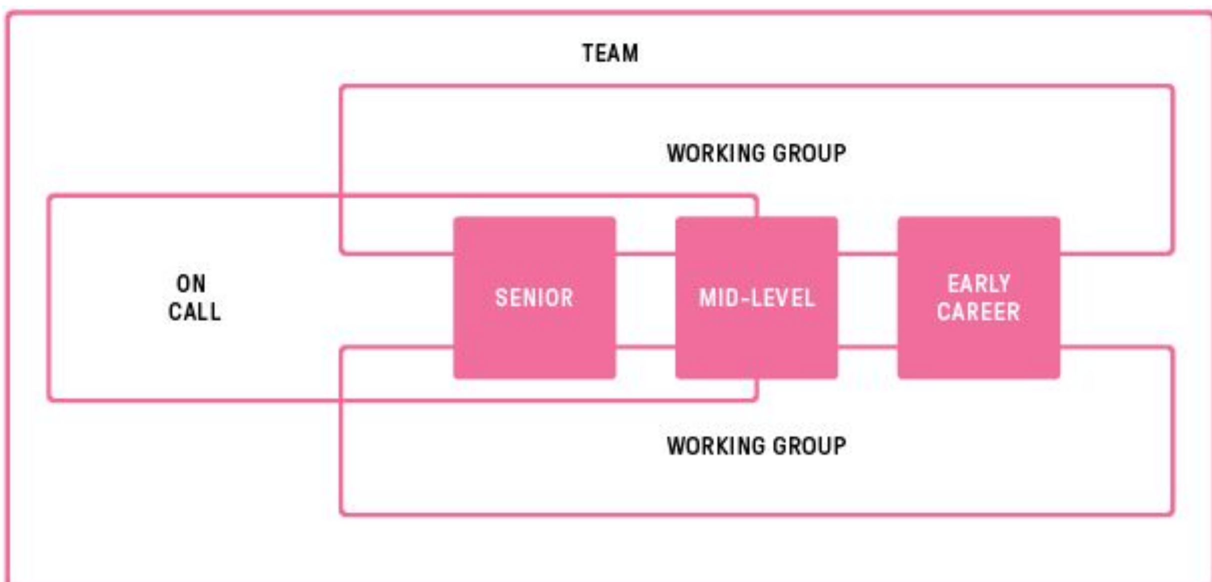


Figure 2.2
A team composed of two working groups, an on-call rotation, and different tenured engineers.

• On-call rotations want eight engineers

For production on-call responsibilities,[4] I've found that two-tier 24/7 support requires eight engineers. As teams holding their own pagers have become increasingly mainstream, this has

become an important sizing constraint, and I try to ensure that every engineering team's steady state is eight people.

Shared rotations. It is sometimes necessary to pool multiple teams together to reach the eight engineers necessary for a 24/7 on-call rotation. This is an effective intermediate step toward teams owning their own on-call rotations, but it is not a good long-term solution. Most folks find being on-call for components that they're unfamiliar with to be disproportionately stressful.

• Small teams (fewer than four members) are not teams

I've sponsored quite a few teams of one or two people, and each time I've regretted it. To repeat: I have regretted it every single time. An important property of teams is that they abstract the complexities of the individuals that compose them. Teams with fewer than four individuals are a sufficiently leaky abstraction that they function indistinguishably from individuals. To reason about a small team's delivery, you'll have to know about each on-call shift, vacation, and interruption.

They are also fragile, with one departure easily moving them from innovation back into toiling to maintain technical debt.

Keep innovation and maintenance together. A frequent practice is to spin up a new team to innovate while existing teams are bogged down in maintenance. I've historically done this myself, but I've moved toward innovating within existing teams.[5] This requires very deliberate decision-making and some bravery,

but in exchange you'll get higher morale and a culture of learning, and will avoid creating a two-tiered class system of innovators and maintainers.

Fitting together those guiding principles, the playbook that I've developed is surprisingly simple and effective:

- Teams should be six to eight during steady state.
- To create a new team, grow an existing team to eight to ten, and then bud into two teams of four or five.
- Never create empty teams.
- Never leave managers supporting more than eight individuals.

Like all guidelines, this is a structure to aid thinking through sizing problems, not a straitjacket to restrict every exception. The context of any situation deserves careful examination, but increasingly I've found that the long-term costs of exceptions outweigh what I once considered their strengths.

2.2 Staying on the path to high-performing teams

A friend is six months into supporting a 60 person engineering group. Perhaps unsurprisingly, most of their teams believe that they have urgent hiring needs. Should my friend spread hiring equally across the teams in need, or focus hiring on just one or two teams until their needs are fully staffed? That is the question.

It’s a great question, and captures a deeply challenging aspect of leading an organization. It’s fun to do initial discovery, learning from and about everyone. The rare moment when you choose to reorganize the team[6] is painful, but concludes quickly. What’s much harder is keeping the faith when you’ve played your cards and need to find space for your plans to come to fruition. Staying the course is particularly fraught when it comes to growing an organization, because some teams always need more than you choose to provide.

When you talk about growing an organization, the conversation usually leads to hiring. While I believe that hiring is a very important approach to growing organizations, I also believe that we reach for it too often. In order to prioritize hiring for scenarios in which it’ll do the most good, over the past year I’ve developed a loose framework for reasoning about what a given team needs to increase performance.

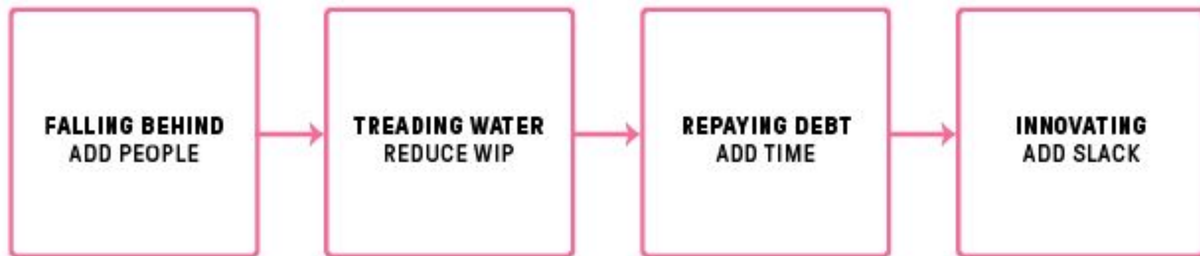


Figure 2.3
Four states of a team.

2.2.1 Four states of a team

The framework starts with a vocabulary for describing teams and their performance within their surrounding context.

Teams are slotted into a continuum of four states:

A team is falling behind if each week their backlog is longer than it was the week before. Typically, people are working extremely hard but not making much progress, morale is low, and your users are vocally dissatisfied.

A team is treading water if they're able to get their critical work done, but are not able to start paying down technical debt or begin major new projects. Morale is a bit higher, but people are still working hard, and your users may seem happier because they've learned that asking for help won't go anywhere.

A team is repaying debt when they're able to start paying down technical debt, and are beginning to benefit from the debt repayment snowball: each piece of debt you repay leads to more time to repay more debt.

A team is innovating when their technical debt is sustainably low, morale is high, and the majority of work is satisfying new user needs.

Teams want to climb from *falling behind* to *innovating*, while entropy drags them backward. Each state requires a different tact.

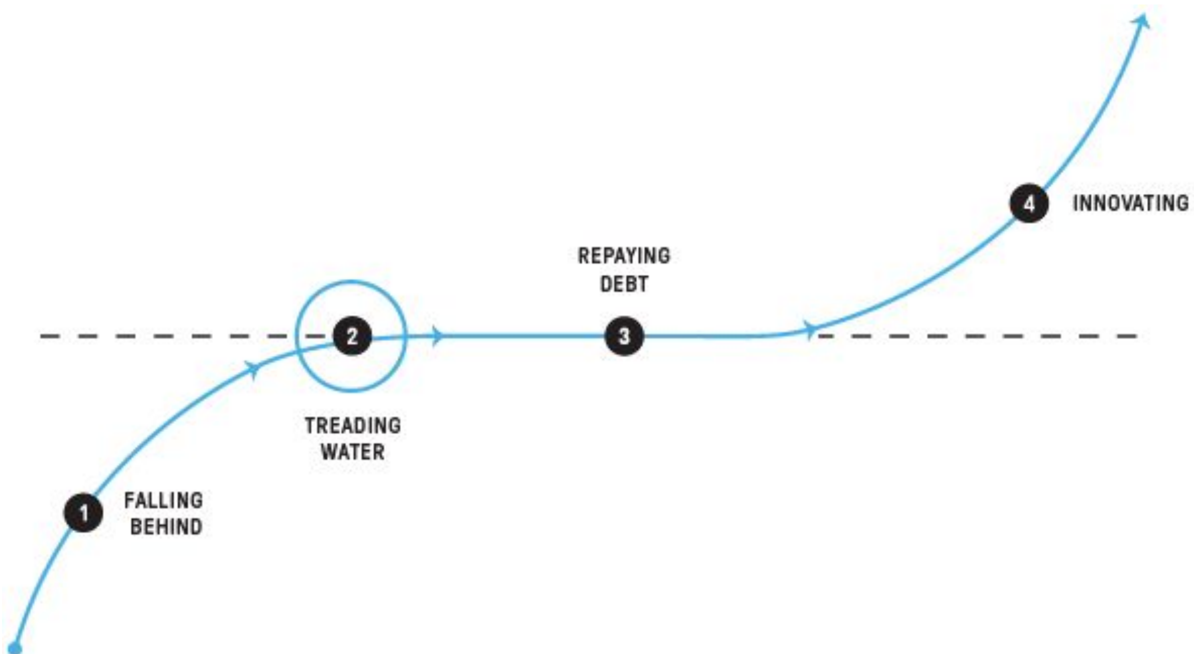


Figure 2.4
Four stages of a team's progress, from falling behind to innovating.

2.2.2 System fixes and tactical support

In this framework, teams transition to a new state exclusively by adopting the appropriate **system solution** for their current state. As a manager, your obligation is to identify the correct system

solution for a given transition, initiate that solution, and then support the team as best you can to create space for the solutions to work their magic. If you skip to supporting the team tactically before initiating the correct system solution, you'll exhaust yourself with no promise of salvation. For each state, here is the strategic solution that I've found most effective, along with some ideas about how to support the team while that solution comes to fruition:

1. **When the team is falling behind**, the system fix is to hire more people until the team moves into treading water. Provide tactical support by setting expectations with users, beating the drum around the easy wins you can find, and injecting optimism.

   As a caveat, the system fix is to hire net new people, increasing the overall capacity of the company. Sometimes people instead attempt to capture more resources from the existing company, and I'm pretty negative on that. People are not fungible, and generally folks end up in useful places, so I'm skeptical of reassigning existing individuals to drive optimality. By nature, it's also impossible for this kind of discussion to not become political, even when everyone involved has deep trust in and respect for each other.

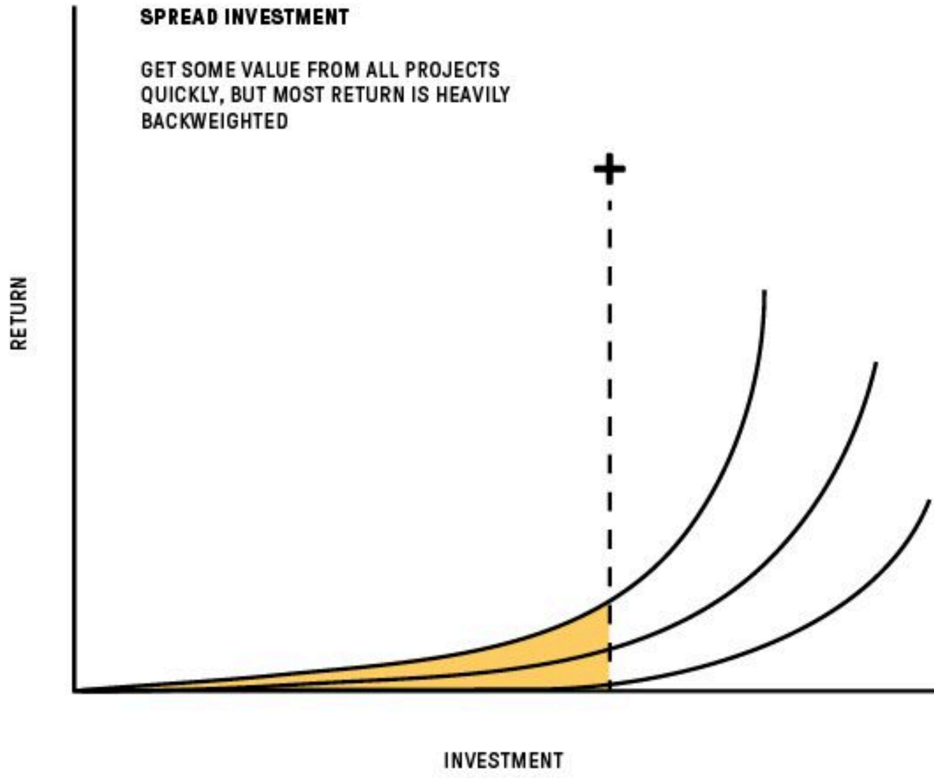
2. **When the team is treading water**, the system fix is to consolidate the team's efforts to finish more things, and to reduce concurrent work until they're able to begin repaying debt (e.g., limit work in progress). Tactically, the focus here is on helping people transition from a personal view of productivity to a team view.

3. **When the team is repaying debt**, the system fix is to add time. Everything is already working, you just need to find space to allow the compounding value of paying down technical debt to grow. Tactically try to find ways to support your users while also repaying debt, to avoid disappearing into technical debt repayment from your users' perspective. Especially for a team that started out falling behind and is now repaying debt, your stakeholders are probably antsy waiting for the team to start delivering new stuff, and your obligation is to prevent that impatience from causing a backslide!

4. **Innovating** is a bit different, because you've nominally reached the end of the continuum, but there is still a system fix! In this case, it's to maintain enough slack in your team's schedule that the team can build quality into their work, operate continuously in innovation, and avoid backtracking. Tactically, ensure that the work your team is doing is valued: the quickest path out of innovation is to be viewed as a team that builds science projects, which inevitably leads to the team being defunded.

I can't stress enough that these fixes are *slow*. This is because systems accumulate months or years of static, and you have to drain that all away. Conversely, the same properties that make these fixes slow to fix make them extremely durable once in effect!

The hard part is maintaining faith in your plan—both your faith and the broader organization's faith. At some point, you may want to launder accountability through a reorg, or maybe skip out to a new job, but if you do that you're also skipping the part where you get to learn. Stay the path.


SPREAD INVESTMENT
GET SOME VALUE FROM ALL PROJECTS
QUICKLY, BUT MOST RETURN IS HEAVILY
BACKWEIGHTED
RETURN
INVESTMENT

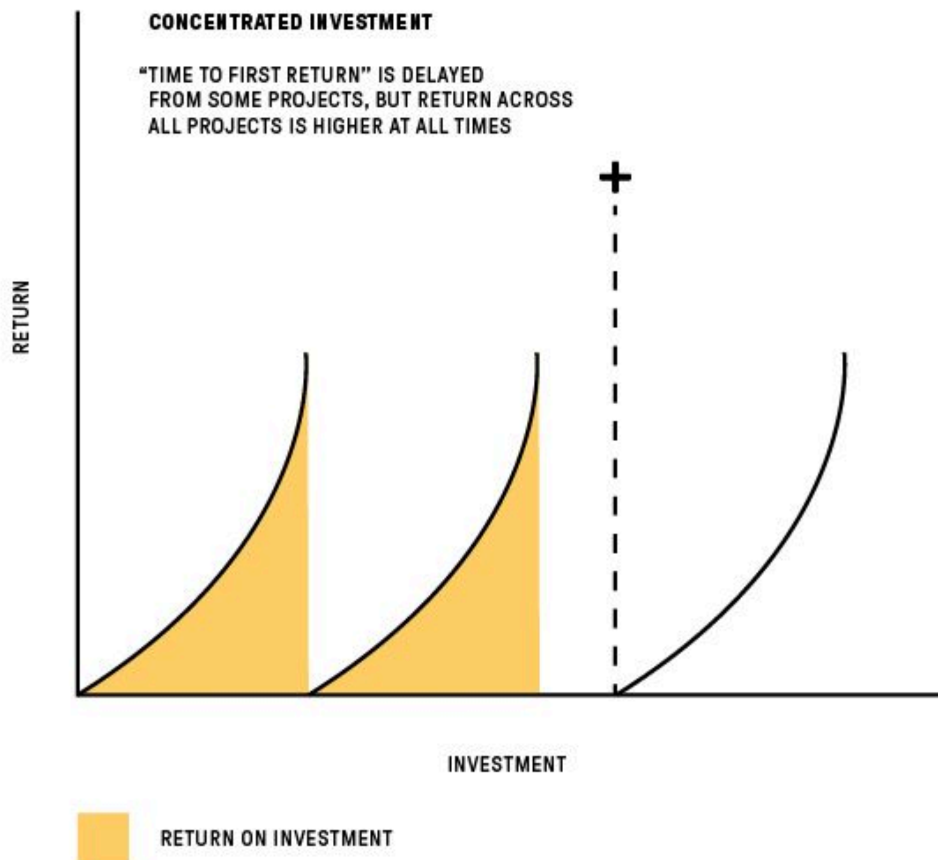


Figure 2.5
Return on investment when consolidating versus spreading investment.

2.2.3 Consolidate your efforts

As an organizational leader, you'll be dealing with a number of teams, each of which is in a different place on this continuum. You'll also have limited resources to apply, and they'll usually be insufficient to simultaneously move every team down the continuum. Many folks try to move all teams at the same time, peanut buttering[7] their limited resources, but resist that

indecision-framed-as-fairness: it's not a fair outcome if no one gets anything.

For each constraint, prioritize one team at a time. If most teams are falling behind, then hire onto one team until it's staffed enough to tread water, and only then move to the next. While this is true for all constraints, it's particularly important for hiring.

Adding new individuals to a team disrupts that team's gelling process, so I've found it much easier to have rapid growth periods for any given team, followed by consolidation/gelling periods during which the team gels. The organization will never stop growing, but each team will.

2.2.4 Durable excellence

This approach to nurturing great organizations is the opposite of a quick fix. While it's slow, I've found that it consistently leads to enduring, real improvement in the happiness and throughput of an organization. Most importantly, these improvements stick around long enough to compound, creating a durable excellence.

2.3 A case against top-down global optimization

After I wrote "Staying on the Path to High-Performance Teams,"[8] quite a few people asked the same follow-up question: "Once a team has repaid its technical debt, shouldn't the now surplus team members move to other teams?"

This makes a lot of sense, because the team, with so little technical debt left, is now overstaffed relative to its global priority. Repeated across many teams, this could lead to an organization having far too many engineers allocated against last year's problems, and too few against today's.

This is an important problem to address!

First, let me explain why I'm skeptical of reallocating individuals to address global priority shifts, and then I'll suggest a couple alternative approaches to this conundrum.

2.3.1 Team first

Fundamentally, I believe that sustained productivity comes from high-performing teams, and that disassembling a high-performing team leads to a significant loss of productivity, even if the members are fully retained. In this worldview, high-performing teams are sacred, and I'm quite hesitant to disassemble them.

Teams take a long time to gel. When a group has been working together for a few years, they understand each other and know how to set each other up for success in a truly remarkable way. Shifting individuals across teams can reset the clock on gelling, especially for teams in the early stages of gelling, and when there are significant differences in team culture. That's not to say that you want teams to never change—that leads to stagnation—but perhaps preserving a team's gelled state requires restrained growth.

Sometimes you will want to grow faster than a gelled team allows, and that's okay! The lesson is that you have to account for re-gelling costs after periods of change, not that you should never change them. This is part of why my proposed model[9] recommends rapidly hiring into teams loaded down by technical debt, not into innovating teams, which avoids incurring re-gelling costs on high-performing teams.

2.3.2 Fixed costs

Another reason that I lean away from moving folks off high-performing teams is that most teams have high fixed costs and relatively small variable costs: moving one person can shift an innovating team back into falling behind, and now neither team is doing particularly well. This is especially true on teams responsible for products and services.

My rule of thumb is that it takes eight engineers on a team to support a two-tier on-call rotation, so I'm generally reluctant to move any team with membership below that line. However, fixed

costs come in many other varieties: "keeping the lights on" work, precommitted contracts, support questions from other teams, etc.

There are some teams with very low fixed costs—a startup without any users, a team supporting a product that you've turned off entirely—and I suspect that the rules for those teams are different. I also suspect that such teams are quite uncommon in successful companies.

2.3.3 Slack

The premise of moving folks to optimize global efficiency also implies a deeper understanding of how productivity is generated than I've ever personally achieved. I'm a strong believer in not adding more resources to a team with visible slack, but I'm less convinced that the inverse applies.

The expected time to complete a new task approaches infinity as a team's utilization approaches 100 percent, and most teams have many dependencies on other teams. Together, these facts mean you can often slow a team down by shifting resources to it, because doing so creates new upstream constraints.

In further defense of slack, I find that teams put spare capacity to great use by improving areas within their aegis, in both incremental and novel ways. As a bonus, they tend to do these improvements with minimal coordination costs, such that the local productivity doesn't introduce drag on the surrounding system.

Most importantly, "slackful" teams function as an organizational debugger: you don't have to consider them when debugging the overall organizational throughput. I've found it much easier to work a couple constraints at a time, solving forward without needing to revisit previous constraints.

The Goal by Eliyahu M. Goldratt[10] and *Thinking in Systems: A Primer* by Donella H. Meadows[11] are both phenomenal books on this topic.

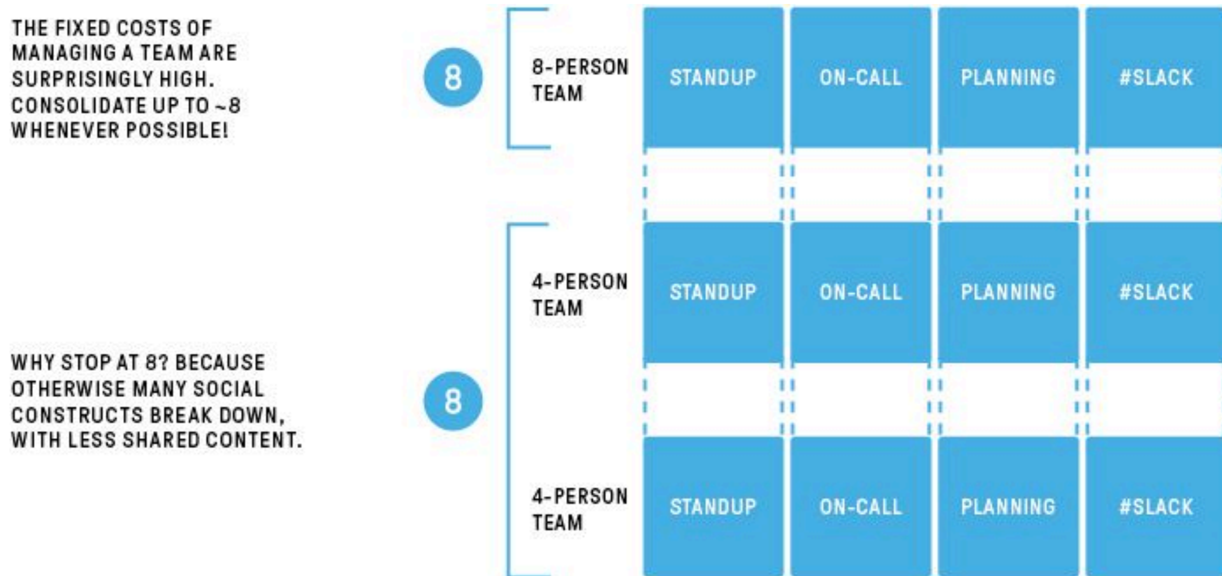


Figure 2.6
Fixed costs of running a team.

2.3.4 Shift scope; rotate

Okay, so what *does* work? I've found it most fruitful to move scope between teams, preserving the teams themselves. If a team has significant slack, then incrementally move responsibility to them,

at which point they'll start locally optimizing their expanded workload. It's best to do this slowly to maintain slack in the team, but if it's a choice of moving people rapidly or shifting scope rapidly, I've found that the latter is more effective and less disruptive.

Shifting scope works better than moving people because it avoids re-gelling costs, and it preserves system behavior. Preserving behavior keeps your existing mental model intact, and if it doesn't work out, you can always revert a workload change with less disruption than would be caused by a staffing change.

The other approach that I've seen work well is to rotate individuals for a fixed period into an area that needs help. The fixed duration allows them to retain their identity and membership in their current team, giving their full focus to helping out, rather than splitting their focus between performing the work and finding membership in the new team. This is also a safe way to measure how much slack the team really has!

A coworker of mine suggested that some companies have very successfully moved toward the swarming model (at the organization level, not just at the team level), and I hope that I eventually get a chance to hear from people who've successfully gone the other direction! One of the most exciting aspects of organizational design is that there are so many different approaches that work well.

2.4 Productivity in the age of hypergrowth

You don't hear the term hypergrowth[12] quite as much as you did a couple years ago. Sure, you might hear it during any given week, but you also might open up Techmeme and not see it, which is a monumental return to a kinder, gentler past. (Or perhaps we're just unicorning[13] now.)

Fortunately for engineering managers everywhere, the challenges of managing within quickly growing companies still very much exist.

When I started at Uber, we were almost 1,000 employees and were doubling the headcount every six months. An old-timer summarized their experience as: "We're growing so quickly that every six months we're a new company." A bystander quickly added a corollary: "Which means our process is always six months behind our head count."

Helping my team be successful when a defunct process merges with a constant influx of new engineers and system load has been one of the most rewarding opportunities I've had in my career. This is an attempt to explore the challenges and propose some strategies I've seen for mitigating and overcoming them.

2.4.1 More engineers, more problems

All real-world systems have some degree of inherent self-healing properties: an overloaded database will slow down enough that someone fixes it, and overwhelmed employees will get slow at finishing work until someone finds a way to help.

Very few real-world systems have efficient and deliberate self-healing properties, and this is where things get exciting as you double engineers and customers year after year after year.

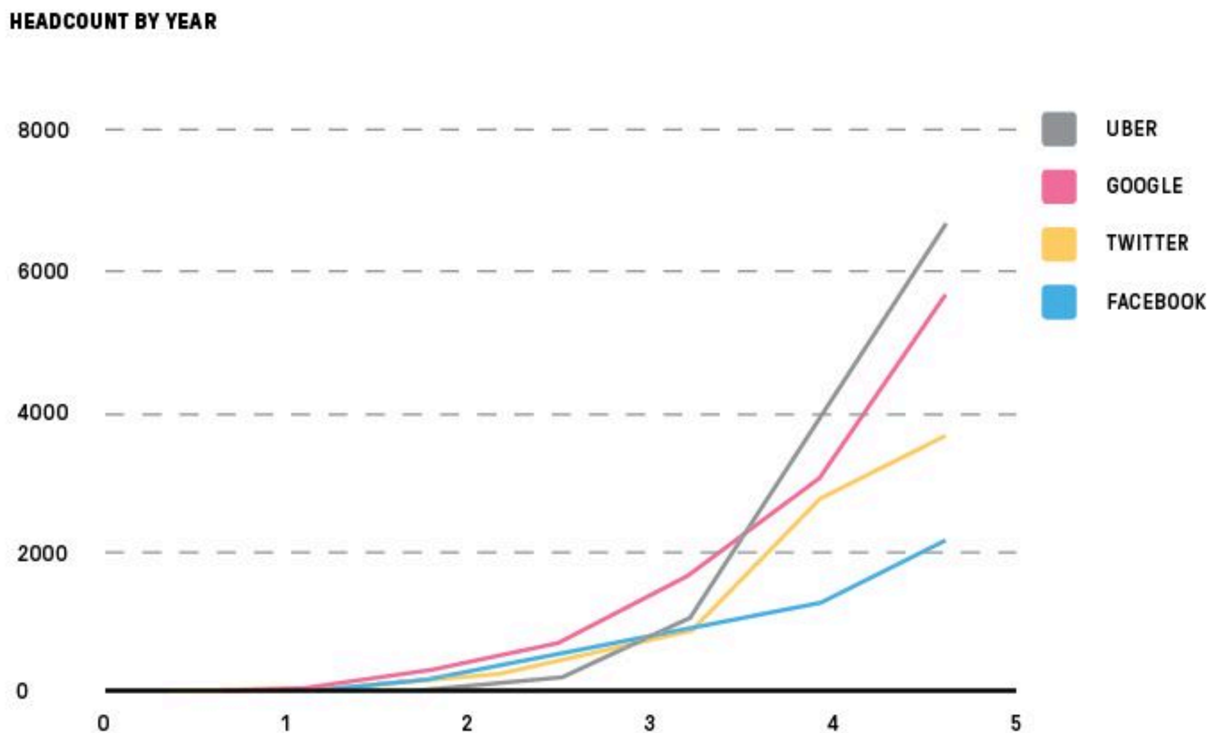


Figure 2.7
Employee growth rate of fast-growing companies.

Productively integrating large numbers of engineers is hard.

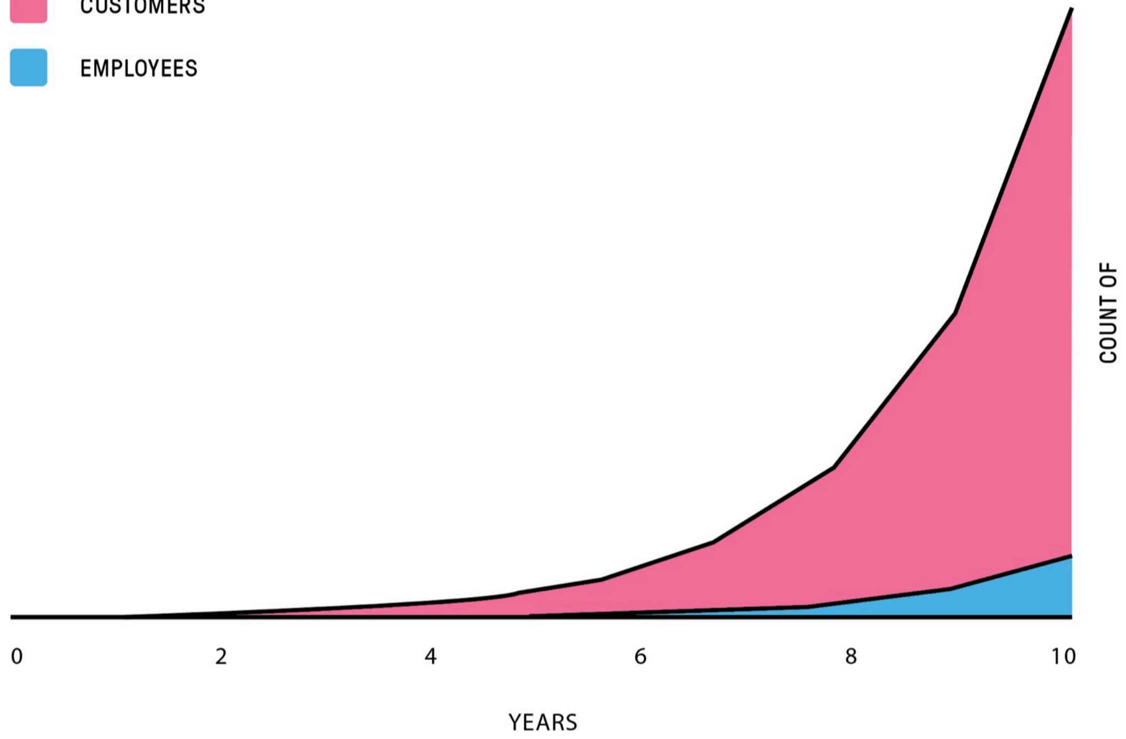
Just how challenging this is depends on how quickly you can ramp engineers up to self-sufficient productivity, but if you're doubling every six months and it takes six to twelve months to

ramp up, then you can quickly find a scenario in which untrained engineers increasingly outnumber the trained engineers, and each trained engineer is devoting much of their time to training a couple of newer engineers.

Imagine a scenario in which training a single new engineer takes about 10 hours per week from each trained engineer, and in which untrained engineers are one-third as productive as trained engineers. The result is the right-hand chart's (admittedly, pretty worst-case scenario) ratio of two-untrained-to-one-trained. Worse, for those three people you're only getting the productivity of 1.16 trained engineers (2 x .33 for the untrained engineers plus .5 x 1 for the trainer).

You also need to factor in the time spent on hiring.

If you're trying to double every six months, and about 10 percent of candidates undergoing phone screens eventually join, then you need to do ten interviews per existing engineer in that time period, with each interview taking about two hours to prep, perform, and debrief.


EMPLOYEES AND CUSTOMERS
CUSTOMERS
EMPLOYEES
COUNT OF
0
2
4
6
8
10
YEARS

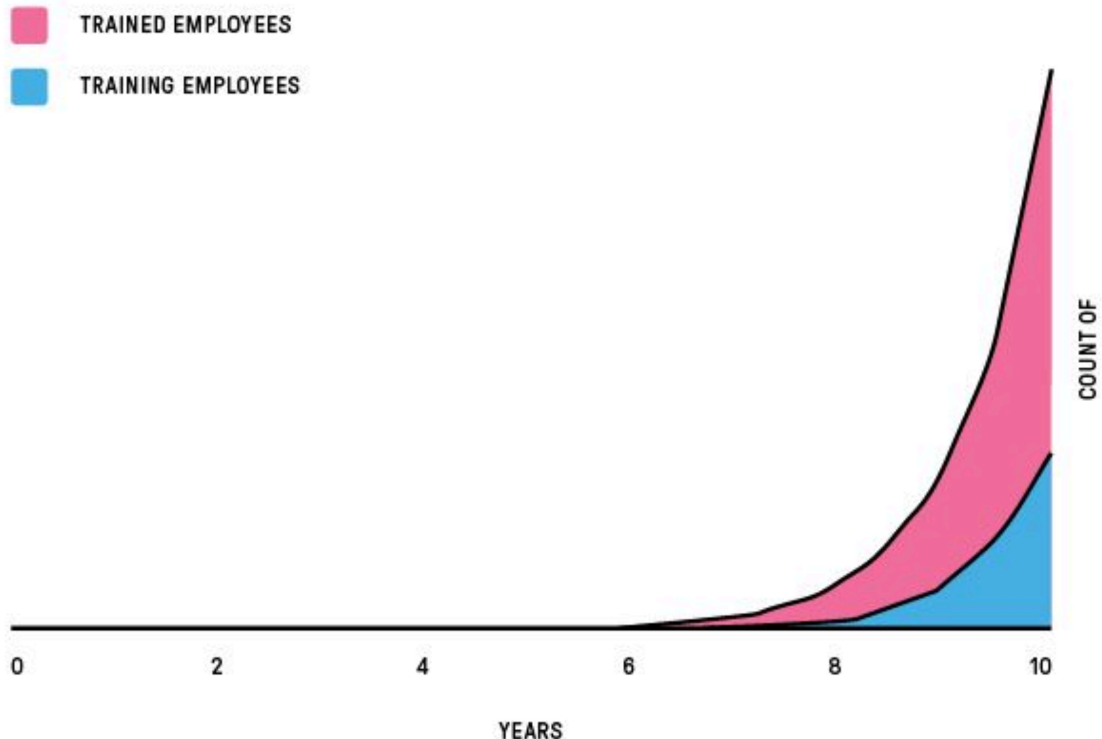


Figure 2.8
More employees, more customers, more problems.

That's less than four hours per engineer per month if you can leverage your entire existing team, but training comes up again here: if it takes you six months to get the average engineer onto your interview loop, each trained engineer is now doing three to four hours of hiring-related work per week, and your trained engineers are down to approximately 0.4 efficiency. The overall team is getting 1.06 engineers' worth of work out of every three engineers.

It's not just training and hiring, though:

1. For every additional order of magnitude of engineers, you need to design and maintain a new layer of management.
2. For every ~10 engineers, you need an additional team, which requires more coordination.[14]
3. Each engineer means more commits and deployments per day, creating load on your development tools.
4. Most outages are caused by deployments, so more deployments drive more outages, which in turn require incident management, mitigations, and postmortems.
5. Having more engineers leads to more specialized teams and systems, which require increasingly small on-call rotations so that your on-call engineers have enough system context to debug and resolve production issues. Consequently, relative time invested in on-call goes up.

Let's do a bit more handwavy math to factor these in.

Only your trained engineers can go on-call. They're on-call one week a month, and are busy about half their time on-call. So that's a total impact of five hours per week for your trained engineers, who are now down to 0.275 efficiency, and your team is now getting less than the output of a single trained engineer for every three engineers you've hired.

(This is admittedly an unfair comparison because it's not accounting for the on-call load on the smaller initial teams, but if you accept the premise that on-call load grows as engineer head

count grows, and that load grows as the number of rotations grows, then the conclusion should still roughly hold.)

Although it's rarely quite this extreme, this is where the oft-raised concern that "hiring is slowing us down" comes from: at high enough rates, the marginal added value of hiring gets very slow, especially if your training process is weak.

Sometimes very low means negative!

2.4.2 Systems survive one magnitude of growth

We've looked a bit at productivity's tortured relationship with engineering head count, so now let's also think a bit about how the load on your systems is growing.

Understanding the overall impact of increased load comes down to a few important trends:

1. Most system-implemented systems are designed to support one to two orders' magnitude of growth from the current load. Even systems designed for more growth tend to run into limitations within one to two orders of magnitude.

2. If your traffic doubles every six months, then your load increases an order of magnitude every 18 months. (And sometimes new features or products cause load to increase much more quickly.)

3. The cardinality of supported systems increases over time as you add teams, and as “trivial” systems go from unsupported afterthoughts to focal points for entire teams as the systems reach scaling plateaus (things like Apache Kafka, mail delivery, Redis, etc.).

If your company is designing systems to last one order of magnitude and is doubling every six months, then you’ll have to re-implement every system twice every three years. This creates a great deal of risk—almost every platform team is working on a critical scaling project—and can also create a great deal of resource contention to finish these concurrent rewrites.

However, the real productivity killer is not system rewrites but the migrations that follow those rewrites. Poorly designed migrations expand the consequences of this rewrite loop from the individual teams supporting the systems to the entire surrounding organization.

If each migration takes a week, each team is eight engineers, and you’re doing four migrations a year, then you’re losing about 1 percent of your company’s total productivity. If each of those migrations takes closer to a month, or if they are only possible for your small cadre of trained engineers—whose time is already tightly contended for—then the impact becomes far more pronounced.

There is a lot more that could be said here—companies that mature rapidly often have tight and urgent deadlines around pursuing various critical projects, and around moving to multiple data centers, to active-active designs, and to new international regions—but I think we’ve covered our bases on how increasing

system load can become a drag on overall engineering throughput.

The real question is, what do we do about any of this?

2.4.3 Ways to manage entropy

My favorite observation from *The Phoenix Project* by Gene Kim, Kevin Behr, and George Spafford[15] is that you only get value from projects when they finish: to make progress, above all else, you must ensure that some of your projects finish.

That might imply that there is an easy solution, but finishing projects is pretty hard when most of your time is consumed by other demands.

Let's tackle hiring first, as hiring and training are often a team's biggest time investment.

When your company has decided that it is going to grow, you cannot stop it from growing, but, on the other hand, you absolutely can concentrate that growth, such that your teams alternate between periods of rapid hiring and periods of consolidation and gelling. Most teams work best when scoped to approximately eight engineers, as each team gets to that point, you can move the hiring spigot to another team (or to a new team). As the post-hiring team gels, eventually the entire group will be trained and able to push projects forward.

You can do something similar on an individual basis, rotating engineers off of interviewing periodically to give them time to

recuperate. With high interview loads, you'll sometimes notice last year's solid interviewer giving a poor experience to a candidate or rejecting every incoming candidate. If your engineer is doing more than three interviews a week, it is a useful act of mercy to give them a month off every three or four months.

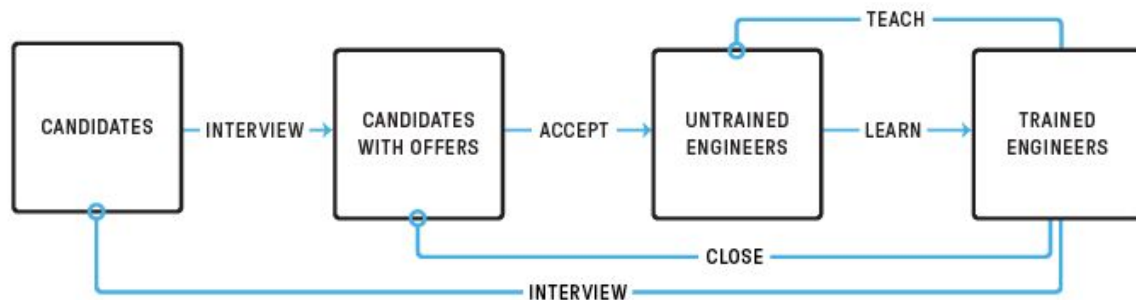


Figure 2.9
Candidates get offers, become untrained, and then learn.

I have less evidence on how to tackle the training component of this, but generally you start to see larger companies do major investments in both new-hire bootcamps and recurring education classes.

I'm optimistically confident that we're not entirely cargo-culting this idea from each other, so it probably works, but I hope to get an opportunity to spend more time understanding how effective those programs can be. If you could get training down to four weeks, imagine how quickly you could hire without overwhelming the existing team!

The second most effective time thief that I've found is ad hoc interruptions: getting pinged on HipChat or Slack, taps on the

shoulder, alerts from your on-call system, high-volume email lists, and so on.

The strategy here is to funnel interruptions into an increasingly small area, and then automate that area as much as possible. Ask people to file tickets, create chatbots that automate filing tickets, create a service cookbook, and so on.

With that setup in place, create a rotation for people who are available to answer questions, and train your team not to answer other forms of interruptions. This is remarkably uncomfortable because we want to be helpful humans, but it becomes necessary as the number of interruptions climbs higher.

One specific tool that I've found extremely helpful here is an ownership registry, which allows you to look up who owns what, eliminating the frequent "Who owns X?" variety of question. You'll need this sort of thing to automate paging the right on-call rotation, so you might as well get two useful tools out of it!

A similar variant of this is ad hoc meeting requests. The best tool that I've found for this is to block out a few large chunks of time each week to focus. This can range from telecommuting on Thursday, to blocking out Monday and Wednesday afternoons, to blocking out from 8–11 each morning. Experiment a bit and find something that works well for you.

Finally, the one thing that I've found at companies with very few interruptions and have observed almost nowhere else: really great, consistently available documentation. It's probably even harder to bootstrap documentation into a non-documenting company than it is to bootstrap unit tests into a non-testing

company, but the best solution to frequent interruptions I've seen is a culture of documentation, documentation reading, and a documentation search that actually works.

There are a non-zero number of companies that do internal documentation well, but I'm less sure if there are a non-zero number of companies with more than 20 engineers that do this well. If you know any, please let me know so that I can pick their brains.

In my opinion, probably the most important opportunity is designing your software to be flexible. I've described this as "fail open and layer policy"; the best system rewrite is the one that didn't happen, and if you can avoid baking in arbitrary policy decisions that will change frequently over time, then you are much more likely to be able to keep using a system for the long term.

If you're going to have to rewrite your systems every few years due to increased scale, let's avoid any unnecessary rewrites, ya know?

Along these lines, if you can keep your interfaces generic, then you are able to skip the migration phase of system re-implementation, which tends to be the longest and trickiest phase, and you can iterate much more quickly and maintain fewer concurrent versions. There is absolutely a cost to maintaining this extra layer of indirection, but if you've already rewritten a system twice, take the time to abstract the interface as part of the third rewrite and thank yourself later. (By the time you'd do the fourth rewrite, you'd be dealing with migrating six times as many engineers.)

Finally, a related antipattern is the gatekeeper pattern. Having humans who perform gatekeeping activities creates very odd social dynamics, and is rarely a great use of a human's time. When at all possible, build systems with sufficient isolation that you can allow most actions to go forward. And when they do occasionally fail, make sure that they fail with a limited blast radius.

There are some cases in which gatekeepers are necessary for legal or compliance reasons, or because a system is deeply frail, but I think that we should generally treat gatekeeping as a significant implementation bug rather than as a stability feature to be emulated.

2.4.4 Closing thoughts

None of the ideas here are instant wins. It's my sense that managing rapid growth is more along the lines of stacking small wins than identifying silver bullets. I have used all of these techniques, and am using most of them today to some extent or another, so hopefully they will at least give you a few ideas. Something that is somewhat ignored a bit here is how to handle urgent project requests when you're already underwater with your existing work and maintenance. The most valuable skill in this situation is learning to say no in a way that is appropriate to your company's culture. That probably deserves its own chapter. There are probably some companies where saying no is culturally impossible, and in those places I guess you either learn to say your noes as yeses, or maybe you find a slightly easier environment to participate in.

How do you remain productive in times of hypergrowth?

2.5 Where to stash your organizational risk?

Lately, I'm increasingly hearing folks reference the idea of *organizational debt*. This is the organizational sibling of *technical debt*, and it represents things like biased interview processes and inequitable compensation mechanisms. These are systemic problems that are preventing your organization from reaching its potential. Like technical debt, these risks linger because they are never the most pressing problem. Until that one fateful moment when they are.

Within organizational debt, there is a volatile subset most likely to come abruptly due, and I call that subset *organizational risk*. Some good examples might be a toxic team culture, a toilsome fire drill, or a struggling leader.

These problems bubble up from your peers, skip-level one-on-ones,[16] and organizational health surveys. If you care and are listening, these are hard to miss. But they are slow to fix. And, oh, do they accumulate! The larger and older your organization is, the more you'll find perched on your capable shoulders.

How you respond to this is, in my opinion, the core challenge of leading a large organization. How do you continue to remain emotionally engaged with the challenges faced by individuals you're responsible to help, when their problem is low in your problems queue? In that moment, do you shrug off the responsibility, either by changing roles or picking powerlessness?

Hide in indifference? Become so hard on yourself that you collapse inward?

I've tried all of these! They weren't very satisfying.

What I've found most successful is to identify a few areas to improve, ensure you're making progress on those, and give yourself permission to do the rest poorly. Work with your manager to write this up as an explicit plan and agree on what reasonable progress looks like. These issues are still stored with your other bags of risk and responsibility, but you've agreed on expectations.

Now you have a set of organizational risks that you're pretty confident will get fixed, and then you have all the others: known problems, likely to go sideways, that you don't believe you're able to address quickly. What do you do about those?

I like to keep them close.

Typically, my organizational philosophy is to stabilize team-by-team and organization-by-organization. Ensuring any given area is well on the path to health before moving my focus. I try not to push risks onto teams that are functioning well. You do need to delegate some risks, but generally I think it's best to only delegate solvable risk. If something simply isn't likely to go well, I think it's best to hold the bag yourself. You *may* be the best suited to manage the risk, but you're almost certainly the best positioned to take responsibility.

As an organizational leader, you'll always have a portfolio of risk, and you'll always be doing very badly at some things that are

important to you. That’s not only okay, it’s unavoidable.

2.6 Succession planning

Two or three years into a role, you may find that your personal rate of learning has trailed off. You know your team well, the industry particulars are no longer quite as intimidating, and you have solved the mystery of getting things done at your company. This can be a sign to start looking for your next role, but it's also a great opportunity to build experience with succession planning.

Succession planning is thinking through how the organization would function without you, documenting those gaps, and starting to fill them in. It's awkward enough to talk about that it doesn't get much discussion, but it's a foundational skill for building an enduring organization.

2.6.1 What do you do?

The first step in succession planning is to figure out what you do. This seems like it should be easy, but I've found it surprisingly hard! There are the obvious things you do—one-on-ones, meetings, head count planning—but you're probably filling in a hundred little holes that you don't even think about.

The approach I've taken is to consider your work from several different angles:

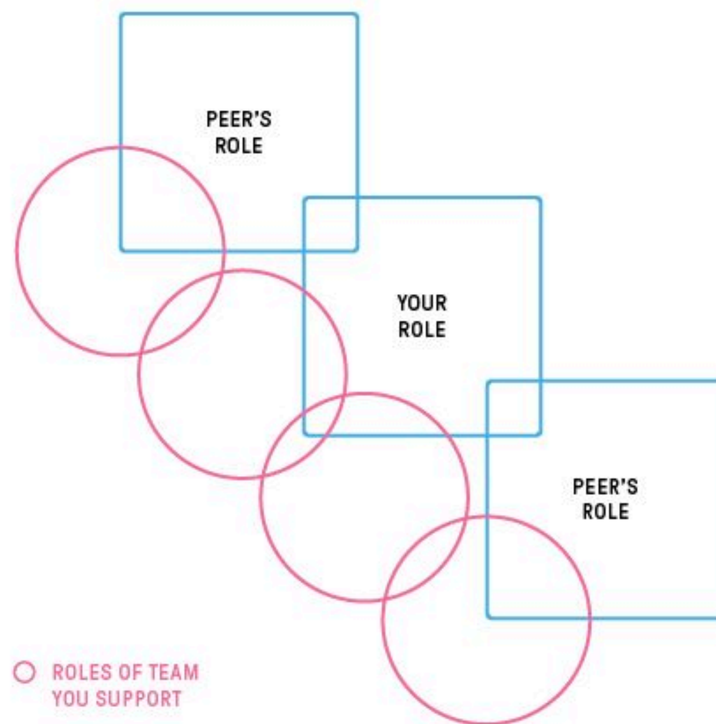


Figure 2.10
Succession planning.

- Take a look at your calendar and write down **your role in meetings**. This goes for explicit roles, like owning a meeting's agenda, and also for more nuanced roles, like being the first person to champion others' ideas, or the person who is diplomatic enough to raise difficult concerns.

- Take a second pass on your calendar for non-meeting stuff, like interviewing and closing candidates.

- Look back over the past six months for **recurring processes**, like roadmap planning, performance calibrations, or head count decisions, and document your role[17] in each of those processes.

- For each of the **individuals you support**, in which areas are your skills and actions most complementary to theirs? How do you help them? What do they rely on you for? Maybe it's authorization, advice navigating the organization, or experience in the technical domain.
- Audit inbound chats and emails for requests and questions coming your way.
- If you keep a **to-do list**, look at the categories of the work you've completed over the past six months, as well as the stuff you've been wanting to do but keep putting off.
- Think through the **external relationships** that have been important for you in your current role. What kinds of folks have been important, and who are the strategic partners that someone needs to know?

After exploring each of these avenues, you'll have quite a long list of things. Test the list on a few folks whom you work closely with and see if you've missed anything. Congratulations, now you know what your job is!

2.6.2 Close the gaps

Take your list, and for each item try to identify the individuals who could readily take on that work. Good job, cross those out.

For items without someone who is ready today, identify a handful of individuals who could potentially take it over. (Depending on the size of your list, it may be helpful to cluster similar items into groups to reduce the toil of running this exercise.)

If you're working at a well-established company, you may find that there aren't too many gaps that couldn't be readily filled by someone else. However, if you're at a company going through hypergrowth,[18] it's common to find that everyone is already working in the most complex role of their career, and you'll uncover gaps, gaping and cavernous.

Filter the gaps down to two lists:

1. The first should cover the *easiest gaps to close*. Maybe it'll require a written document or a quick introduction. You should be able to close one of these in less than four hours.

2. The latter will be the *riskiest gaps*. These are the areas where you're uniquely valuable to the company, where other folks are missing skills, and where getting the tasks done is truly important. You'd expect closing one of these gaps to require ongoing effort over several months.

Write up a plan to close *all* of the easy gaps and *one or two* of the riskiest gaps. Add it to your personal goals, and then, congrats, you've completed a round of succession planning!

This isn't a one-time tool, but rather a great exercise to run once a year to identify things you could be delegating. This helps nurture an enduring organization, and also frees up time for you to continue growing into a larger role as well. You can even get a sense of how well you're doing by taking a two- or three-week vacation and seeing what slips through the cracks.

Those items can be the start of next year's list!

3

CHAPTER 3

Tools

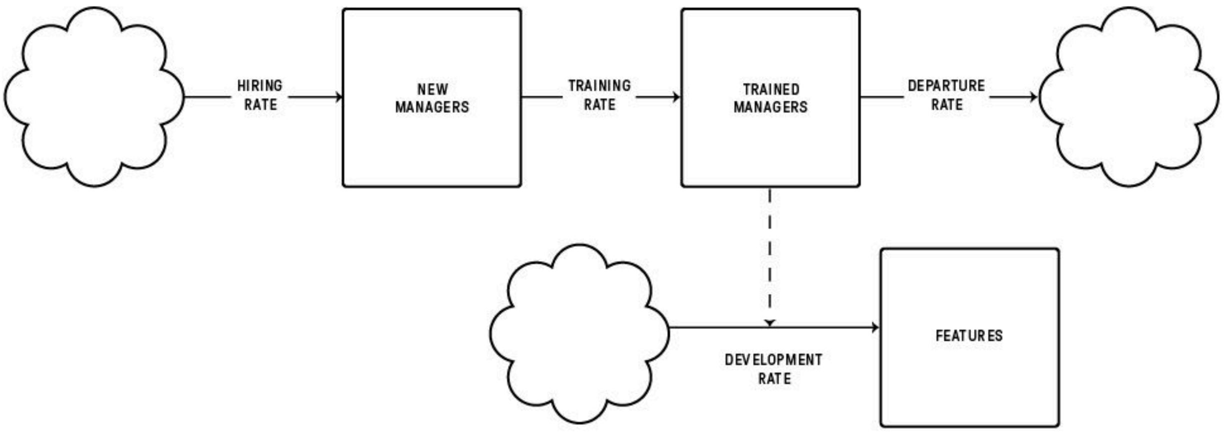


Figure 3.1
System diagram for hiring and training new managers.

Tools

If you ask a manager about their proudest moments, they will probably tell you a story about helping someone grow. If you ask that same manager about their most challenging experience, they will probably talk about a layoff, a reorganization, a shift in company direction, or the time they weathered an economic downturn. In management, change is the catalyst of complexity.

The best changes often go unnoticed, moving from one moment of stability to another, with teams and organizations feeling stable at every step. The key tools for leading efficient change are systems thinking, metrics, and vision. When the steps of change are too wide, teams get destabilized, and gaps open within them. In those moments, managers create stability by becoming glue. We step in as product managers, program managers, recruiters, or salespeople to hold the bits together until an expert relieves us.

This chapter provides a box of tools for managing change, both from the abstract chair of guiding change and from the more visceral role of serving as glue during periods of transition.

3.1 Introduction to systems thinking

Many effective leaders I've worked with have the uncanny knack for working on leveraged[1] problems. In some problem domains, the product management skill set[2] is extraordinarily effective for identifying useful problems, but systems thinking is the most universally useful tool kit I've found.

If you really want a solid grasp on systems thinking fundamentals, you should read *Thinking in Systems: A Primer*[3] by Donella H. Meadows, but I'll do my best to describe some of the basics and to work through a recent scenario in which I found the systems thinking approach to be exceptionally useful.

3.1.1 Stocks and flows

The fundamental observation of systems thinking is that the links between events are often more subtle than they appear. We want to describe events causally—our managers are too busy because we're trying to ship our current project—but few events occur in a vacuum.

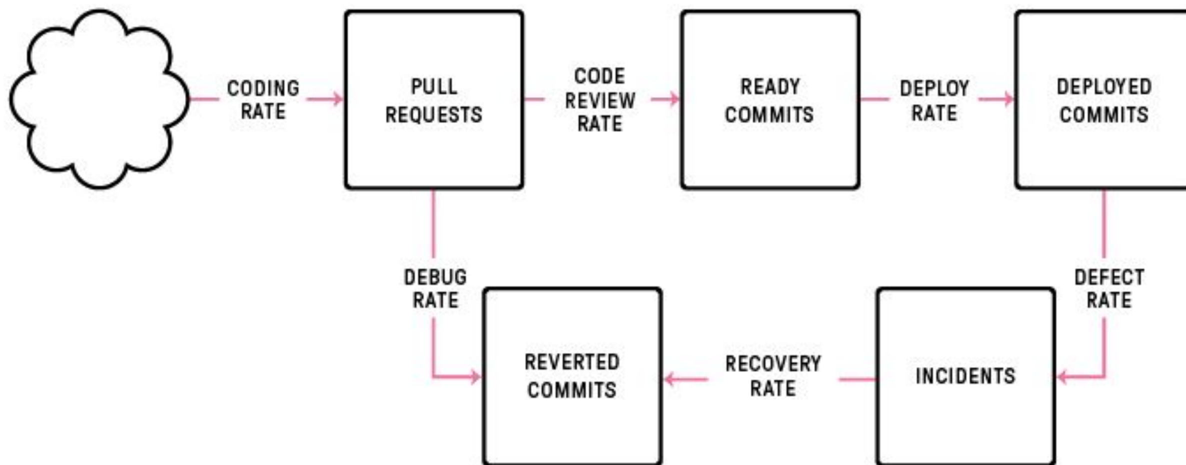


Figure 3.2
System diagram for developer productivity.

Big changes appear to happen in a moment, but if you look closely underneath the big change, there is usually a slow accumulation of small changes. In this example, perhaps the managers are busy because no one hired and trained the managers required to support this year's project deadlines. These accumulations are called *stocks*, and are the memory of changes over time. A stock might be the number of trained managers at your company.

Changes to stocks are called *flows*. These can be either *inflows* or *outflows*. Training a new manager is an inflow, and a trained manager who departs the company is an outflow. Diagrams in this chapter represent flows with solid dark lines.

The other relationship, represented in figure 3.1 by a dashed line, is an *information link*. This indicates that the value of a stock is a factor in the size of a flow. The link here shows that the time

available for developing features depends on the number of trained managers.

Often, a stock outside of a diagram's scope will be represented as a cloud, indicating that something complex happened there that we're not currently exploring. It's best practice to label every flow, and to keep in mind that every flow is a rate, whereas every stock is a quantity.

3.1.2 Developer velocity

When I started thinking of an example of the usefulness of systems thinking, one came to mind immediately. Since reading *Accelerate: The Science of Lean Software and DevOp*, by Nicole Forsgren, Gene Kim, and Jez Humble,[4] I've spent a lot of time pondering the authors' definition of velocity.

They focus on four measures of developer velocity:

1. **Delivery lead time** is the time from the creation of code to its use in production.
2. **Deployment frequency** is how often you deploy code.
3. **Change fail rate** is how frequently changes fail.
4. **Time to restore service** is the time spent recovering from defects.

The book uses surveys from tens of thousands of organizations to assess each one's overall productivity and show how that

correlates to the organization's performance on those four dimensions.

These dimensions kind of intuitively make sense as measures of productivity, but let's see if we can model them into a system that we can use to reason about developer productivity:

- **Pull requests** are converted into **ready commits** based on our *code review rate*.
- **Ready commits** convert into **deployed commits** at *deploy rate*.
- **Deployed commits** convert into **incidents** at *defect rate*.
- **Incidents** are remediated into **reverted commits** at *recovery rate*.
- **Reverted commits** are debugged into new **pull requests** at *debug rate*.

Linking these pieces together, we see a *feedback loop*, in which the system's downstream behavior impacts its upstream behavior. With a sufficiently high *defect rate* or slow *recovery rate*, you could easily see a world where each deploy leaves you even further behind.

If your model is a good one, opportunities for improvement should be immediately obvious, which I believe is true in this case. However, to truly identify where to invest, you need to identify the true values of these stocks and flows! For example, if you don't have a backlog of **ready commits**, then speeding up your *deploy rate* may not be valuable. Likewise, if your *defect rate* is very low,

then reducing your *recovery time* will have little impact on the system.

Creating an arena for quickly testing hypotheses about how things work, without having to do the underlying work beforehand, is the aspect of systems thinking that I appreciate most.

3.1.3 Model away

Once you start thinking about systems, you'll find that it's hard to stop. Pretty much any difficult problem is worth trying to represent as a system, and even without numbers plugged in I find them powerful thinking aids.

If you do want the full experience, there are relatively few tools out there to support you. Stella[5] is the gold standard, but the price is quite steep, with a nonacademic license costing more than a new laptop. The best cheap alternative that I've found is Insight Maker,[6] which has some UI quirks but features a donation-based payment model.

3.2 Product management: exploration, selection, validation

Most engineering organizations separate engineering and product leadership into distinct roles. This is usually ideal, not only because these roles benefit from distinct skills but also because they thrive on different perspectives and priorities. It's quite hard to do both well at the same time.

I've met many product managers who are excellent operators, but few product managers who can operate at a high degree while also getting deep into their users' needs. Likewise, I've worked with many engineering managers who ground their work in their users' needs, but I've known few who can fix their attention on those users when things start getting rocky within their team.

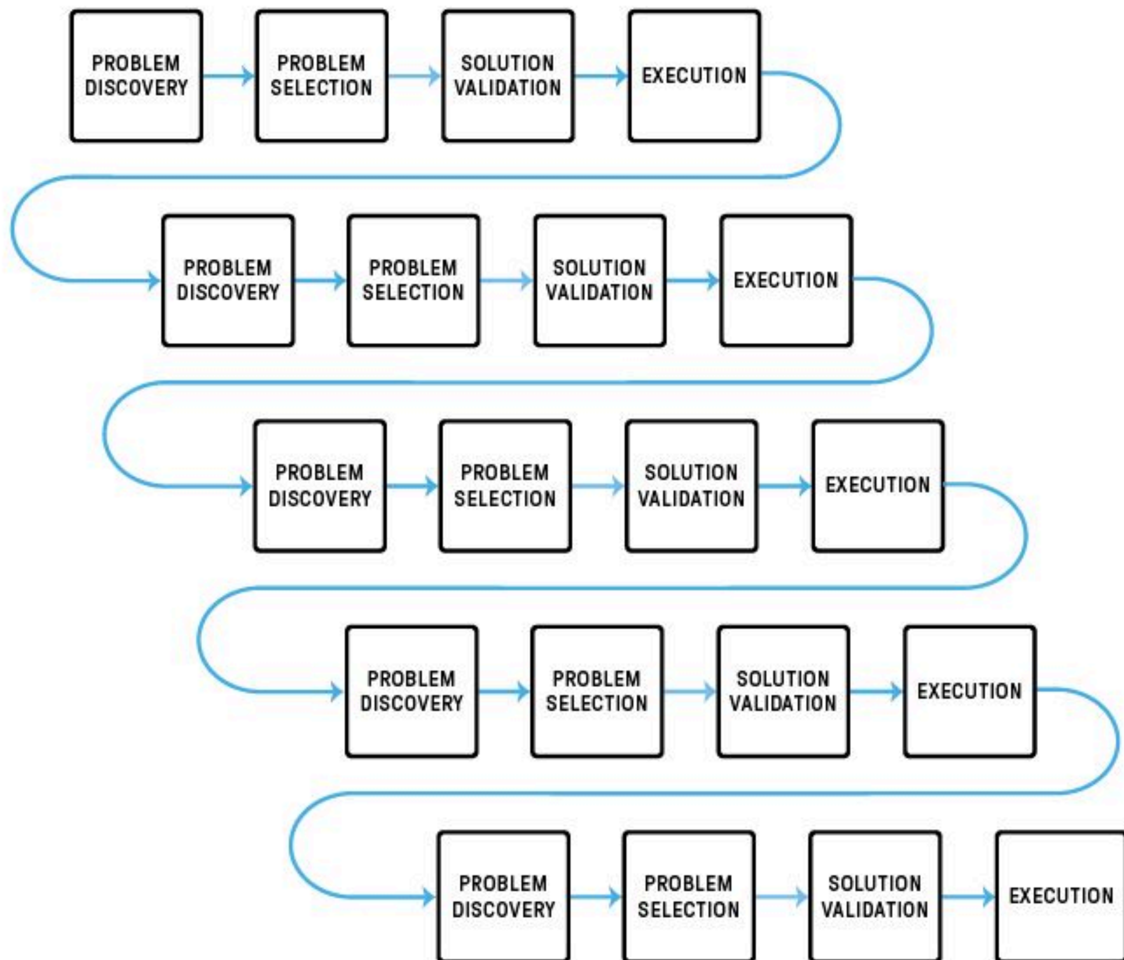


Figure 3.3
Iterative process of product development.

Reality isn't always accommodating of this ideal setup. Maybe your team's product manager leaves or a new team is being formed,[7] and you, as an engineering leader, need to cover both roles for a few months. This can be exciting, and, yes this can be a time when "exciting" rhymes with "terrifying."

Product management is a deep profession, and mastery requires years of practice, but I've developed a simple framework to use

when I've found myself fulfilling product management[8] responsibilities for a team. It's not perfect, but hopefully it'll be useful for you as well.

Product management is an iterative elimination tournament, with each round consisting of *problem discovery*, *problem selection*, and *solution validation*. *Problem discovery* is uncovering possible problems to work on, *problem selection* is filtering those problems down to a viable subset, and *solution validation* is ensuring that your approach to solving those problems works as cheaply as possible.

If you do a good job at all three phases, you win the luxury of doing it all again, this time with more complexity and scope. If you don't do well, you end up forfeiting or being asked to leave the game.[9]

3.2.1 Problem discovery

The first phase of a planning cycle is exploring the different problems that you could pick to solve. It's surprisingly common to skip this phase, but that, unsurprisingly, leads to inertia-driven local optimization. Taking the time to evaluate which problem to solve is one of the best predictors I've found of a team's long-term performance.

The themes that I've found useful for populating the problem space are:

Users' pain. What are the problems that your users experience? It's useful to go broad via survey mechanisms, as well as to go

deep by interviewing a smaller set of interesting individuals across different user segments.

Users' purpose. What motivates your users to engage with your systems? How can you better enable users to accomplish their goals?

Benchmark. Look at how your company compares to competitors in the same and similar industries. Are there areas in which you are quite weak? Those are areas to *consider* investing in. Sometimes folks keep to a narrow lens when benchmarking, but I've found that you learn the most interesting things by considering both fairly similar and rather different companies.

Cohorts. What is hiding behind your clean distributions? Exploring your data for the cohorts hidden behind top-level analysis is an effective way to discover new kinds of users with surprising needs.

Competitive advantages. By understanding the areas you're exceptionally strong in, you can identify opportunities that you're better positioned to fill than other companies.

Competitive moats. Moats are a more extreme version of a competitive advantage. Moats represent a sustaining competitive advantage, which makes it possible for you to pursue offerings that others simply cannot. It's useful to consider moats in three different ways:

- What do your existing moats enable you to do today?
- What are the potential moats you could build for the future?

- What moats are your competitors luxuriating behind?

Compounding leverage. What are the composable blocks you could start building today that would compound into major product or technical leverage[10] over time? I think of this category of work as finding ways to get the benefit at least twice. These are potentially tasks that initially don't seem important enough to prioritize, but whose compounding value makes the work possible to prioritize.

- A design example might be introducing to an application a new navigation scheme that better supports the expanded set of actions and modes you have today, and that will support future proliferation as well. (Bonus points if it manages to prevent future arguments about the positioning of new actions relative to existing ones!)
- An infrastructure example might be moving a failing piece of technology to a new standard. This addresses a reliability issue, lowers maintenance costs, and also reduces the costs of future migrations.[11]

3.2.2 Problem selection

Once you've identified enough possible problems, the next challenge is to narrow down to a specific problem portfolio. Some of the aspects that I've found useful to consider during this phase are:

Surviving the round. Thinking back to the iterative elimination tournament, what do you need to do to survive the current round?

This might be the revenue that the product will need to generate to avoid getting canceled, adoption, etc.

Surviving the next round. Where do you need to be when the next round in order to avoid getting eliminated then? There are a number of ways (many of them revolving around quality trade-offs) to reduce long-term throughput in favor of short-term velocity. (Conversely, winning leads to significantly more resources later, so that trade-off is appropriate sometimes!)

Winning rounds. It's important to survive every round, but it's also important to eventually win a round! What work would ensure that you're trending toward winning a round?

Consider different time frames. When folks disagree about which problems to work on, I find that the conflict is most frequently rooted in different assumptions about the correct time frame to optimize for. What would you do if your company was going to run out of money in six months? What if there were no external factors forcing you to show results until two years out? Five years out?

Industry trends. Where do you think the industry is moving, and what work will position you to take advantage of those trends, or to at least avoid having to redo the work in the near future?

Return on investment. Personally, I think people often under-prioritize quick, easy wins. If you're in the uncommon position of understanding both the impact and costs of doing small projects, then take time to try ordering problems by expected return on investment. At this phase, you're unlikely to know the exact solution, so figuring out cost is tricky, but for categories of

problems that you've seen before you can probably make a solid guess. (If you don't personally have relevant experience, ask around.) Particularly in cases where wins are compounding, they may be surprisingly valuable over the medium and long term.

Experiments to learn. What could you learn now that would make problem selection in the future much easier?

3.2.3 Solution validation

Once you've narrowed down the problem you want to solve, it's easy to jump directly into execution, but that can make it easy to fall in love with a difficult approach. Instead, I've found that it's well worth it to take the risk out of your approach with an explicit solution validation phase.

The elements that I've found effective for solution validation are:

Write a customer letter. Write the launch announcement that you would send after finishing the solution. Are you able to write something exciting, useful, and real? It's much more useful to test it against your actual users than to rely on your intuition.

Identify prior art. How do peers across the industry approach this problem? The fact that others have solved a problem in a certain way doesn't mean that it's a great way, but it does at least mean it's possible. A mild caveat: it's better to rely on people you have some connection to instead of on conference talks and such, since there is a surprisingly large amount of misinformation out there.

Find reference users. Can you find users who are willing to be the first users for the solution? If you can't, you should be skeptical whether what you're building is worthwhile.

Prefer experimentation over analysis. It's far more reliable to get good at cheap validation than it is to get great at consistently picking the right solution. Even if you're brilliant, you are almost always missing essential information when you begin designing. Analysis can often uncover missing information, but it depends on knowing where to look, whereas experimentation allows you to find problems that you didn't anticipate.

Find the path more quickly traveled. The most expensive way to validate a solution is to build it in its entirety. The upside of that approach is that you've lost no time if you picked a good solution. The downside is that you've sacrificed a huge amount of time if it's not. Try to find the cheapest way to validate.

Justify switching costs. What will the costs of switching be for users who move to your solution? Even if folks want to use it, high switching costs may mean that they simply won't be able to. Test with your potential users if they'd be willing to pay the full cost of migrating to your solution instead of their existing planned work.

As an aside, I've found that most aspects of running a successful technology migration[12] overlap with good solution validation! This is a very general skill that will repay many times over the time you invest in learning it.

Putting these three elements in place today—exploration, selection, and validation—won't make you an exceptional product manager overnight, but they will provide a solid starting place to

develop those skills and perspective for the next time you find yourself donning the product manager hat.

3.3 Visions and strategies

As an organization grows beyond 50 people or so, you'll feel a building pressure to add a third layer of management, and eventually you will. This ought to be a benign event: What's the difference between supporting some managers and supporting their managers? It shouldn't be too different, but for me it was when my previous mechanisms of alignment stopped working very well.

Where I was once partnering with teams on their project roadmaps, now I found myself increasingly surprised by the projects that they were working on. Where I was once debating different approaches with the teams, now that conversation was happening in rooms I didn't have time to join.

My first instinct was to dive in and understand each instance, but that, unsurprisingly, wasn't a very scalable solution. My second instinct was to design a series of "operating reviews" in which we periodically reviewed metrics and major projects. While those were useful in their own right, they proved more effective for learning and fine-tuning than for broad alignment. Being out of alignment for a quarter is just so much uncaptured potential.

What I needed was a way to coordinate my approach across teams, both in terms of very specific challenges and in terms of our long-term direction. After experimenting with a handful of different approaches, agreeing on strategy and vision has been the most effective approach that I've found to alignment at scale.

3.3.1 Strategies and visions

Strategies are grounded documents which explain the trade-offs and actions that will be taken to address a specific challenge. *Visions* are aspirational documents that enable individuals who don't work closely together to make decisions that fit together cleanly.

Picking the right format for your needs is important, but the most important thing is probably to give both a try and get a feel for them! These are peculiar genres of literature that take some practice to master. It's taken me years to get comfortable writing visions, and it was only when I started to support teams with seemingly incompatible ideologies that these documents value became clear. The same goes for writing strategies: it took a long time and skeptical study of several strategy books before what I wrote started to come together into a useful artifact.

With that said, it's time to dig into how to write strategies first, and then visions.

	STRATEGY	VISION
PURPOSE	APPROACH TO A SPECIFIC CHALLENGE	A GENTLE, ALIGNING PRESSURE
CHARACTER	PRACTICAL	ASPIRATIONAL
TIME FRAME	VARIABLE	LONG-TERM
SPECIFICITY	ACCURATE, DETAILED	ILLUSTRATIVE, DIRECTIONAL
QUANTITY	AS MANY AS USEFUL	AS FEW AS POSSIBLE

Figure 3.4
Table of differences between strategy and vision.

3.3.2 Strategy

A *strategy* recommends specific actions that address a given challenge's constraints. A structure that I've found extremely effective[13] is described in *Good Strategy/Bad Strategy* by Richard Rumelt,[14] and has three sections: *diagnosis*, *policies*, and *actions*.

The *diagnosis* is a theory describing the challenge at hand. It calls out the factors and constraints that define the challenge, and at its core is a very thorough problem statement. An example of a simple diagnosis might be "I am too busy to think about long-term goals. I attend 35 hours of meetings each week. I am under pressure to immediately increase team performance. I believe

that if I stop doing my current meetings, short-term team performance will decrease. I am concerned that if my short-term team performance decreases, I may lose face as an effective leader, which will undermine my career opportunities. I believe that if I don't think about long-term goals, our performance will never improve, which will also undermine my career opportunities." Before you've even finished reading a great diagnosis, you'll often have identified several good candidate approaches. That's the power of a well-defined problem statement, and why it's an important foundational element for your strategy.

The second step is to identify *policies* that you will apply to address the challenge. These describe the general approach that you'll take, and are often trade-offs between two competing goals. Continuing the above example, you might choose to allow short-term performance to dip in order to invest into long-term performance, combined with a policy of proactive expectation-setting with your stakeholders. Conversely, you might choose to take a hill-climbing approach to long-term performance, with iterative short-term improvement leading to improved long-term performance. Both are valid guiding policies, and both embrace the reality that you have limited resources to invest. When you read bad guiding policies, you think, "so what?" because its found a way to justify entrenching the status quo. When you read good guiding policies, you think, "Ah, that's really going to annoy Anna, Bill, and Claire," because the approach takes a clear stance on competing goals.

When you apply your guiding policies to your diagnosis, you get your *actions*. Folks are often comfortable with hard decisions in

the abstract, but struggle to translate policies into the specific steps to implement them. This is typically the easiest part to write, but publishing it and following through with it can be a significant test of your commitment. In the example above, your specific actions might be to stop attending weekly team meetings in order to free up time and move to a monthly metrics review, combined with blocked-out focus hours that you unapologetically shelter. When you read good, coherent actions, you think, “This is going to be uncomfortable, but I think it can work.” When you read bad ones, you think, “Ah, we got afraid of the consequences, and we aren’t really changing anything.”

Because strategies are specific to a given problem, it’s okay—and even encouraged—to write quite a few of them. Over the past year, I’ve worked with people on strategies for how we partner with other teams, how we manage end-to-end API latency, and how we manage infrastructure costs.[15] I’ve also peered over others’ shoulders as they worked on quite a few more ideas. The act of writing a strategy leads folks through a systematic analysis, so, even if we don’t share them, writing these documents helps us work through quite a few challenges, both overwhelming and mundane.

People sometimes describe strategy as artful or sophisticated, but I’ve found that the hardest part of writing a good strategy is pretty mundane. You must be honest about the constraints that are making the challenge difficult, which almost always include people and organizational aspects that are uncomfortable to acknowledge. No extent of artistry can solve a problem that you’re unwilling to admit.

3.3.3 Vision

If strategies describe the harsh trade-offs necessary to overcome a particular challenge, then *visions* describe a future in which those trade-offs are no longer mutually exclusive. An effective vision helps folks think beyond the constraints of their local maxima, and lightly aligns progress without requiring tight centralized coordination.

You should be writing from a place far enough out that the error bars of uncertainty are indisputably broad, where you can focus on the concepts and not the particulars. Visions should be detailed, but the details are used to illustrate the dream vividly, not to prescriptively constrain its possibilities.

A good vision is composed of:

1. **Vision statement:** A one- or two-sentence aspirational statement to summarize the rest of the document. This is your core speaking point, which you will repeat at each meeting, planning period, and strategy review. It shouldn't try to capture every detail of your vision, but it does need to memorably evoke your vision effectively.

2. **Value proposition:** How will you be valuable to your users and to your company? What kinds of success will you enable them to achieve? There is a sequencing question of whether you should start with *capabilities* (the next bullet) and reason into *value proposition* or do the opposite, but I've found that starting from your users leads to visions that are both more ambitious and more grounded.

3. **Capabilities:** What capabilities will the product, platform, or team need in order to deliver on your value proposition? Will it need to support multiple independent business lines? Will it need to deliver against the disparate needs of several distinct customer cohorts?

4. **Solved constraints:** What are the constraints that you're limited by today, but that in the future you'll no longer be constrained by? For example, if you're currently "spending into" developer velocity, perhaps in the future you'll be able to achieve significant developer velocity while also maintaining low costs.

5. **Future constraints:** What are the constraints that you expect to encounter in this wonderful future? Hopefully, these new constraints will be things that are easy to adjust, like funding or hiring.

6. **Reference materials:** Link all the existing plans, metrics, updates, references, and documents into an appendix for those who want to understand more of the thinking that went into the vision. This allows you to shed complexity from your vision document without sacrificing context.

7. **Narrative:** Once you've written the previous sections, the last step of writing a compelling vision is to synthesize all those details into a short—maybe one-page—narrative that serves as an easy-to-digest summary. In your final document, this is probably the second section, following the statement.

Put all these pieces together, and you've crafted a document that is a guiding hand to align decisions yet still creates room for teams to make their own choices and trade-offs along the way.

You'll know a vision is succeeding when people reference the document to make their own decisions, and you'll know it's struggling when decisions keep happening that don't fit into its direction.

Compared to strategies, there is more room to play with the vision format. You'll be reading far fewer visions than strategies, so consistency is less important. Play with the content a bit to find what works best for you.

A few additional tips that I've found especially useful:

Test the document! This is a core leadership tool, and your first version will almost certainly be bad. Write a draft, sit down with a few different folks to get their perspectives, then iterate. Keep doing this until you've synthesized feedback. If there is feedback you disagree with, embrace the vision as an opportunity to address conflict by explicitly acknowledging disagreements within the vision text.

Refresh periodically. Take some time every year to refresh the vision, and prefer usefulness over consistency. If your old vision doesn't resonate, it's okay to start over: it's a sign that you've learned a lot over the past year.

Use present tense. This makes the writing impactful and concise, and conveys a sense of confidence about the future.

Write simply. Often, visions are saturated with buzzwords, which turns readers off.

You'll likely want *one vision for every complete distinct area, but no more.* If areas overlap, you get the alignment value from having one unified vision; having two clearly articulated visions in one place is worse than having zero.

Like other leadership tools, a vision or strategy document is a solution to a specific set of problems, and it's not always useful. If your team is aligned and doing good work, time spent writing these probably won't be too valuable.

However, if your team is struggling to align with stakeholders, or if you're struggling to lead a cohesive organization, these documents are exceptionally useful, fairly quick to write as you gain practice, and low risk (at worst, they get ignored).

3.4 Metrics and baselines

There is a moment in every company's growth when top-level planning shifts from discussing specific projects to talking about goals. This happens recursively across each scope of leadership, as areas of accountability become too broad or complex for their leaders to consistently understand every project's details.

This can be a very empowering moment because goals decouple the "what" from the "how," but it can also be a confusing transition for everyone involved: writing clear goals takes a bit of practice.

• Defining goals

Bad goals are indistinguishable from numbers. "Our p50 build time will be below two seconds," or "We'll finish eight large projects." You'll know a goal is just a number when you read it and aren't sure if it's ambitious or whether it matters.

Good goals are a composition of four specific kinds of numbers:

1. A **target** states where you want to reach.
2. A **baseline** identifies where you are today.
3. A **trend** describes the current velocity.

4. A **time frame** sets bounds for the change.

Put these all together, and a well-structured goal takes the form of: "In Q3, we will reduce time to render our frontpage from 600ms (p95) to 300ms (p95). In Q2, render time increased from 500ms to 600ms."

The two tests of an effective goal are whether someone who doesn't know much about an area can get a feel for a goal's degree of difficulty, and whether afterward they can evaluate if it was successfully achieved. If you define all four aspects, typically your goal will fulfill both criteria.

• Investments and baselines

There are two particularly interesting kinds of goals: investments and baselines. Investments describe a future state that you want to reach, and baselines describe aspects of the present that you want to preserve.

Imagine that you wanted to speed up your data pipeline. Your goal might be, "Core batch jobs should finish within three hours (p95) by the end of Q3. They currently take six hours (p95), and over the course of Q2 they got two hours slower." This is a well-structured goal, but it's also incomplete because you could likely reach that goal tomorrow by doubling the size of your cluster, which is probably not a desirable outcome.

The best way to avoid such unintended outcomes is to pair your investment goals with baseline metrics, sometimes referred to as

countervailing metrics. For the data pipeline example, a few of the baseline metrics might be:

- Efficiency of running core batch jobs should not exceed current price of $0.05 per GB.
- Core batch jobs should not increase alert load on teams operating or using the pipeline, which are currently alerting twice per week.

Baseline metrics are useful for narrowing the solution space that you explore in order to accomplish your investment goals. They are also useful for identifying when you should pause pursuing your goals and instead invest in platform quality. For example, if you were making excellent progress toward launching a new feature but site stability has regressed below your baselines, this framework provides a structure to trigger rebalancing your priorities.

Although your baselines will often be about preserving a current property, you can also decide to accept some degradation before you want to trigger reprioritization. Perhaps you're okay with costs increasing by 10 percent as long as your investment goals are accomplished. This kind of upfront clarity around trade-offs can be quite powerful.

• Plans and contracts

The most common way to use goals is during a planning process. By agreeing on the mix of investment and baseline goals for each

team, you're able to set clear expectations for a team while still giving them full ownership of how they'll satisfy the constraints. I've found that you should specify as few investment goals as possible, maybe three, and that those should be the focus of planning discussions.

You'll probably want to identify more baseline goals than investment goals, but it's easiest to separate them out to avoid bogging down the conversation. Ideally, baselines are carried over across planning periods, such that they frame the investment goals but don't require too much active discussion during any given planning cycle.

One potential exception is when you're using a baseline as a contract with a second party, possibly specifying an SLO,[16] at which point you'll probably want to discuss it more explicitly than other baselines: missing an SLO will probably require immediate reprioritization, whereas missing most other baselines can generally be addressed more methodically.

From OKRs[17] onward, there are dozens of different approaches to setting metrics, but I've found this format to be a useful, lightweight structure to start from. If folks have found other approaches easier or more useful, I'd love to hear from you!

3.5 Guiding broad organizational change with metrics

Although people often talk about goals and metrics[18] when they're writing new plans or reflecting on past plans, my fondest memories of metrics are when I've seen them used to drive large organizational change. In particular, I've found metrics to be an extremely effective way to lead change with little or no organizational authority, and I wanted to write up how I've seen that work.

At both Stripe and Uber, I've had the opportunity to manage infrastructure costs. (Let me insert a plug for Ryan Lopopolo's amazing blog post on "Effectively Using AWS Reserved Instances."[19]) Folks who haven't thought about this problem often default to viewing it as boring, but I've found that as you dig into it, it's rich soil for learning about leading organizational change.

It's also a good example of how to lead change with metrics!

Infrastructure cost is a great example of a baseline metric.[20] When you're asked to take responsibility for a company's overall infrastructure costs, you're going to start from a goal along the lines of "Maintain infrastructure costs at their current percentage of net revenue of 30 percent." (That percentage is a fictional number for this example's purposes, since the percentage will depend on your industry and maturity, but I *have* found that tying

it against net revenue is more useful than pinning it at a specific dollar amount.)

From there, the approach that I've found effective is:

1. **Explore:** The first step is to get data in an explorable format in your data warehouse, an SQL database, or even an Excel spreadsheet. Once there, spend time looking through it and getting a feel for it. Your goal in this phase is to identify where the levers for change are. For example, you might find that your batch pipeline is the majority of your costs and that your data warehouse is surprisingly cheap, which will allow you to focus further efforts.

2. **Dive:** Once you know the three or four major contributors, go deep on understanding those areas and the levers that drive them. Batch costs might be sensitive to number of jobs, total data stored, or new product development, *or* it might be entirely driven by a couple of expensive jobs. Diving deep helps you build a mental model, and it also kicks off a relationship between you and the teams who you'll want to partner with most closely.

3. **Attribute:** For most company-level metrics (cost, latency, development velocity, etc.), the first step of diving will uncover one team who are nominally accountable for the metric's performance, but they are typically a cloak. When you pull that cloak aside, that team's performance is actually driven by dozens of other teams. For example, you might have a Cloud engineering team who are accountable for provisioning VMs, but they're not the folks writing the code that runs on those VMs. It's easy to simply pass the cost metric on to that Cloud

team, but that's just abdicating responsibility to them. What's more useful is to help them build a system of second-degree attribution, allowing you to build data around the teams using the platform. This second degree of attribution is going to allow you to target the folks who can make an impact in the next step.

4. **Contextualize:** Armed with the attribution data, start to build context around each team's performance. The most general and self-managing tool for this is benchmarking. It's one thing for a team to know that they're spending $100,000 a month, and it's something entirely different for them to know that they're spending $100,000 a month *and* that their team spends the second-highest amount out of 47 teams. Benchmarking is particularly powerful because it automatically adapts to changes in behavior. In some cases, benchmarking against all teams might be too coarse, and it may be useful to benchmark against a small handful of cohorts. For example, you might want to define cohorts for front-end, back-end, and infrastructure teams, given that they'll have very different cost profiles.

5. **Nudge:** Once you've built context around the data so that folks can interpret it, the next step is to start nudging them to action! Dashboards are very powerful for analysis, but the challenge for baseline metrics is that folks shouldn't need to think about them the vast majority of the time, and that can lead to them forgetting about the baselines entirely. What I've found effective is to send push notifications, typically email, to teams whose metric has changed recently, both in terms of absolute change and in terms of their benchmarked performance against their cohort. This ensures that each time

you push information to a team, it includes important information that they should act on! What's so powerful about nudges is that simply letting folks know their behavior has changed will typically stir them to action, and it doesn't require any sort of organizational authority to do so. (For more on this topic, take a look at *Nudge* by Richard H. Thaler and Cass R. Sunstein.)[21]

6. **Baseline:** In the best case, you'll be able to drive the organizational impact you need with contextualized nudges, but in some cases that isn't quite enough. The next step is to work with the key teams to agree on baseline metrics for their performance. This is useful because it ensures that the baselines are top-of-mind, and it also gives them a powerful tool for negotiating priorities with their stakeholders. In some cases, this does require some organizational authority, but I've found that folks universally want to be responsible. As long as you can find time to sit down with the key teams and explain why the goal is important, it typically doesn't require much organizational authority.

7. **Review:** The final phase, which hopefully you won't need to reach, is running a monthly or quarterly review that looks at each team's performance, and reaching out to teams to advocate for prioritization if they aren't sustaining their agreed-upon baselines. This typically requires an executive sponsor, because teams who aren't hitting their baselines are almost always being prioritized against other goals, and they your help explaining to their stakeholders why the change is important.

I've seen this approach work and, more importantly, I've found it to be very scalable. It enables a company to concurrently

maintain many baseline metrics without overloading its teams. This is largely because this approach focuses on driving targeted change within the key drivers, only requiring involvement from a small subset of teams for any given metric. The approach is also effective because it tries to minimize top-down orchestration in favor of providing information to encourage teams themselves to adjust priorities.

3.6 Migrations: the sole scalable fix to tech debt

The most interesting migration I ever participated in was Uber's migration from Puppet-managed services to a fully self-service provisioning model in which any engineer at the company could spin up a new service in two clicks. Not only could they, they did, provisioning multiple services each day by the time the service was complete, and every newly hired engineer could spin up a service from scratch on their first day.

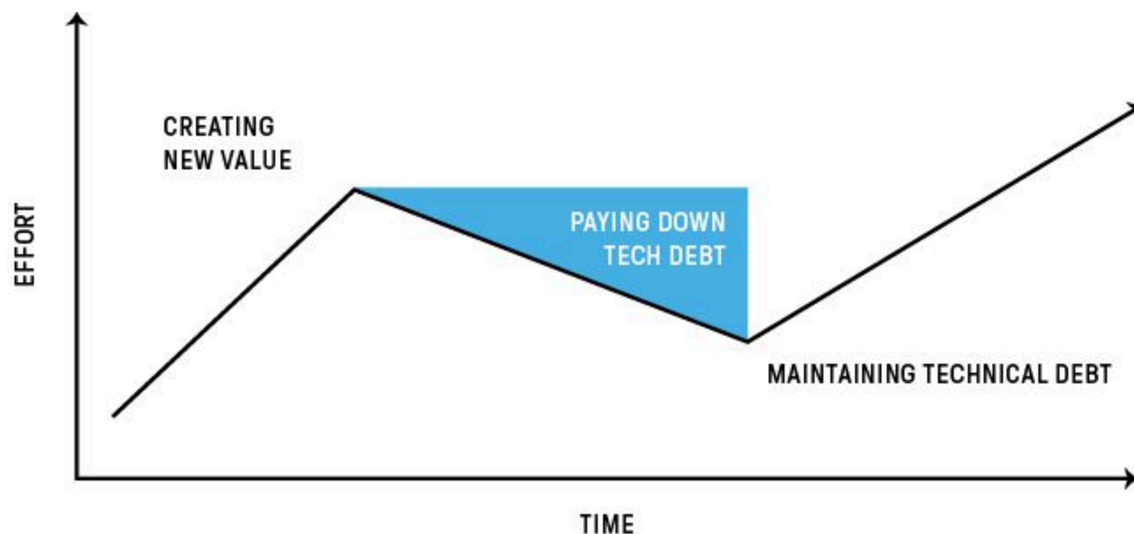


Figure 3.5
Stages of a technical migration.

What made this migration so interesting was the volume. When we started, provisioning a new service took about two weeks of clock time and about two days of engineering time, and we were

falling further behind each day. At the time, this was more than just a little stressful, but it was also a perfect laboratory to learn how to run large-scale software migrations: the transition was large enough to see even small shifts, and long enough that we got to experiment with a number of approaches.

Migrations are both essential and frustratingly frequent as your codebase ages and your business grows: most tools and processes only support about one order of magnitude of growth[22] before becoming ineffective, so rapid growth makes migrations a way of life. This isn't because you have bad processes or poor tools—quite the opposite. The fact that something stops working at significantly increased scale is a sign that it was designed appropriately to the previous constraints rather than being over-designed.[23]

As a result, you switch tools a lot, and your ability to migrate to new software can easily become the defining constraint for your overall velocity. Given their importance, we don't talk about running migrations very often; let's remedy that!

3.6.1 Why migrations matter

Migrations matter because they are usually the only available avenue to make meaningful progress on technical debt.

Engineers hate technical debt. If there is an easy project that they can personally do to reduce tech debt, they'll take it on themselves. Engineering managers hate technical debt, too. If there is an easy project that their team can execute in isolation,

they'll get it scheduled. In aggregate, this leads to a dynamic in which there is very little low-hanging fruit to reduce technical debt, and most remaining options require many teams working together to implement them. The result: migrations.

Each migration aims to create technical leverage (Your indexes no longer have to fit on a single server!) or reduce technical debt (Your acknowledged writes are guaranteed to persist a master failover!). They occupy the awkward territory of reduced immediate contribution today in exchange for more capacity tomorrow. This makes migrations controversial to schedule, and as your systems become larger, they become more expensive. Lore tells us that Googlers have a phrase, "running to stand still," to describe a team whose entire capacity is consumed in upgrading dependencies and patterns, such that the group can't make forward progress on the product/system they own. Spending *all* your time on migrations is extreme, but every midsize company has a long queue of migrations that it can't staff: moving from VMs to containers, rolling out circuit-breaking, moving to the new build tool . . . the list extends effortlessly into the sunset.

Migrations are the only mechanism to effectively manage technical debt as your company and code grow. If you don't get effective at software and system migrations, you'll end up languishing in technical debt. (And you'll still have to do one later anyway, it's just that it'll probably be a full rewrite.)

3.6.2 Running good migrations

The good news is that while migrations are hard, there is a pretty standard playbook that works remarkably well: de-risk, enable, then finish.

• De-risk

The first phase of a migration is **de-risking** it, and to do so as quickly and cheaply as possible. Write a **design document** and shop it with the teams that you believe will have the hardest time migrating. Iterate. Shop it with teams who have atypical patterns and edge cases. Iterate. Test it against the next six to twelve months of roadmap. Iterate.

After you've evolved the design, the next step is to **embed into the most challenging one or two teams**, and work side by side with those teams to build, evolve, and migrate to the new system. Don't start with the easiest migrations, which can lead to a false sense of security.

Effective de-risking is essential, because **each team who endorses a migration is making a bet on you** that you're going to get this damn thing done, and not leave them with a migration to an abandoned system that they have to revert to. If you leave one migration partially finished, people will be exceedingly suspicious of participating in the next.

• Enable

Once you've validated the solution that solves the intended problem, it's time to start sharpening your tools. Many folks start migrations by generating tracking tickets for teams to implement, but it's better to slow down and build tooling to programmatically migrate the easy 90 percent.[24] This radically reduces the migration's cost to the broader organization, which increases the organization's success rate and creates more future opportunities to migrate.

Once you've handled as much of the migration programmatically as possible, figure out the **self-service tooling and documentation** that you can provide to allow teams to make the necessary changes without getting stuck. The best migration tools are incremental and reversible: folks should be able to immediately return to previous behavior if something goes wrong, and they should have the necessary expressiveness to de-risk their particular migration path.

Documentation and self-service tooling are products, and they thrive under the same regime: sit down with some teams and watch them follow your instructions, then improve them. Find another team. Repeat. Spending an extra two days intentionally making your documentation clean and your tools intuitive can save years in large migrations. Do it!

• Finish

The last phase of a migration is deprecating the legacy system that you've replaced. This requires getting to 100 percent adoption, and that can be quite challenging.

Start by **stopping the bleeding**, which is ensuring that all newly written code uses the new approach. That can be installing a ratchet in your linters,[25] or updating your documentation and self-service tooling. This is always the first step, because it turns time into your friend. Instead of falling behind by default, you're now making progress by default.

Okay, now you should start **generating tracking tickets**, and set in place a mechanism which **pushes migration status** to teams that need to migrate and to the general management structure. It's important to give wider management context around migrations because the managers are the people who need to prioritize the migrations: if a team isn't working on a migration, it's typically because their leadership has not prioritized it.

At this point, you're pretty close to complete, but you have the long tail of weird or unstaffed work. Your tool now is: **finish it yourself**. It's not necessarily fun, but getting to 100 percent is going to require the team leading the migration to dig into the nooks and crannies themselves.

My final tip for finishing migrations centers around recognition. It's important to celebrate migrations while they're ongoing, but the majority of the celebration and **recognition should be reserved for their successful completion**. In particular, starting but not finishing migrations often incurs significant technical debt, so your incentives and recognition structure should be careful to avoid perverse incentives.

3.7 Running an engineering reorg

I believe that, at quickly growing companies, there are two managerial skills that have a disproportionate impact on your organization's success: making technical migrations cheap, and running clean reorganizations. Do both well, and you can skip that lovely running-to-stand-still sensation, and invest your attention more fruitfully.

Of the two, managing organizational change is more general, so let's work through a lightly structured framework for (re)designing an engineering organization.

Caveat: this is more of a thinking tool than a recipe!

My approach for planning organization change:

1. Validate that organizational change is the right tool.
2. Project head count a year out.
3. Set target ratio of management to individual contributors.
4. Identify logical teams and groups of teams.
5. Plan staffing for the teams and groups.
6. Commit to moving forward.
7. Roll out the change.

Now, let’s drill into each of those a bit.

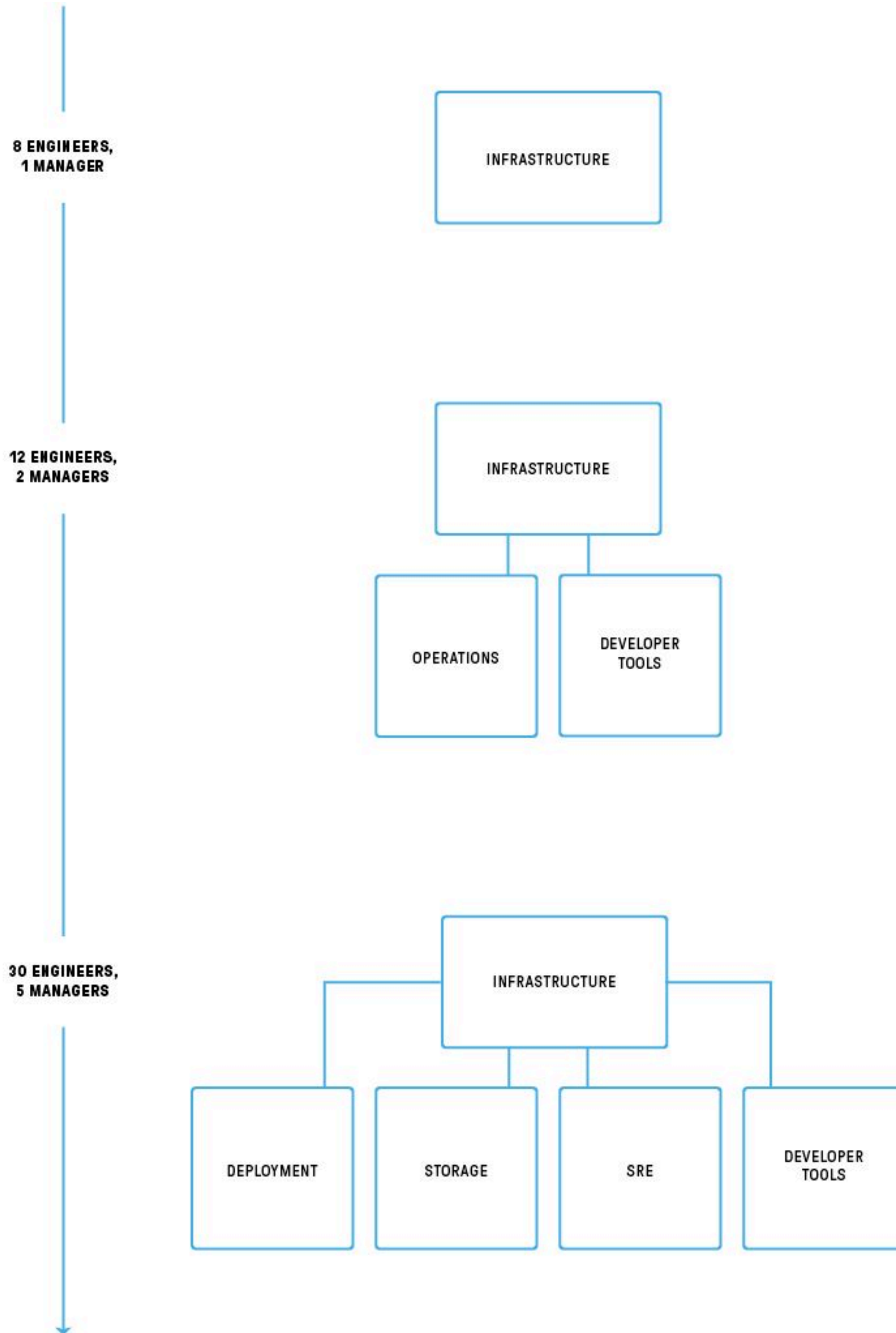


Figure 3.6
Refactoring organizations as they grow.

3.7.1 Is a reorg the right tool?

There are two best kinds of reorganizations:

- The one that solves a structural problem.
- The one that you don't do.

There is only one worst kind of reorg: the one you do because you're avoiding a people management issue.

My checklist for ensuring that a reorganization is appropriate:

1. Is the problem structural? Organization change offers the opportunity to increase communication, reduce decision friction, and focus attention; if you're looking for a different change, consider if there's a more direct approach.

2. Are you reorganizing to work around a broken relationship? Management is a profession where karma *always* comes due, and you'll be better off addressing the underlying issue than continuing to work around it.

3. Does the problem already exist? It's better to wait until a problem actively exists before solving it, because it's remarkably hard to predict future problems. Even if you're right that the problem will occur, you may end up hitting a different problem first.

4. Are the conditions temporary? Are you in a major crunch period or otherwise doing something you don't anticipate

doing again? If so, then it may be easier to patch through and rethink on the other side, and avoid optimizing for a transient failure mode.

All right, so you're still thinking that you want a reorg.

3.7.2 Project head count a year out

The first step of designing the organization is determining its approximate total size. I recommend reasoning through this number from three or four different directions:

1. An optimistic number based on what's barely possible.

2. A number based on the "natural size" of your organization, if every team and role was fully staffed.

3. A realistic number based on historical hiring rates.

Then merge those into a single number.

Unless you've changed something meaningful in your process, it's likely that the historical trend will hold accurate, and you should weight that figure the most heavily (and my sense is that the list of easy changes that significantly after hiring outcomes is short).

One of the goals of using the year-out head count number is to avoid optimizing too heavily for your exact current situation and the current individuals you're working with. Organizational change is *so disruptive* to *so many people* that I've increasingly

come to believe you should drive organizational design from the boxes and not from the key individuals.

3.7.3 Manager-to-engineer ratio

Once you have your head count projection, you need to identify how many individuals you want each manager to support. This number particularly depends on your company's working definition of an engineering manager's role. If engineering managers are expected to do hands-on technical work, then their teams should likely be three to five engineers (unless the team has been working together well for a long time, in which case things get very specific and hard to generalize about).

Otherwise, targeting five to eight engineers, depending on experience level, is pretty typical. If you're targeting more than eight engineers per manager, then it's worth reflecting on why you believe your managers can support a significantly higher load than industry average: Are they exceptionally experienced? Are your expectations lower than typical?

In any case, pick your target, probably in the six-to-eight range.

3.7.4 Defining teams and groups

Now that you have your target organization size and target ratio of managers to engineers, it's time to figure out the general shape of your organization!

Suppose that you have 35 engineers and 7 engineers per manager.

35 / 7 = 5 managers

$Log^{7}(35) \approx 1.8$ managers of managers, or second-degree managers

In a growing company, you should generally round up the number of managers, as this is a calculation “at rest,” and your organization will be a living, evolving thing.

Once you have the numbers, these are useful to *ground* you in the general number of teams and groups of teams you should have.

In the first case, with 35 engineers, you’re going to want between one and three groups, containing a total of five or six teams. In the latter, with 74 engineers, you’ll want two to four groups, comprised of 12 to 15 teams.

Once you’ve grounded yourself, here are some additional considerations:

1. Can you write a crisp mission statement for each team?

2. Would you personally be excited to be a member of each of the teams, as well as to be the manager of each of those teams?

3. Put teams that work together (especially poorly) as close together as possible. This minimizes the distance for escalations during disagreements, allowing arbiters to have sufficient context. Also, most poor working relationships are

the by-product of information gaps, and nothing fills those faster than proximity.

4. Can you define clear interfaces for each team?
5. Can you list the areas of ownership for each team?
6. Have you created a gap-less map of ownership, such that each responsibility is owned by a team? Try to avoid implicitly creating holes of ownership. If you need to create explicit holes of ownership, that's a better solution (essentially, defining unstaffed teams).
7. Are there compelling candidate pitches for each of those teams?
8. As always, are you over-optimizing on individuals versus establishing a sensible structure?

This is the least formulaic aspect of organizational design, and, if possible it's a good time to lean on your network and similar organizations for ideas.

3.7.5 Staffing the teams and groups

With your organization design and head count planning, you can roughly determine when you'll need to fill each of the technical and management leadership positions.

From there, you have four sources of candidates to staff them:

1. Team members who are ready to fill the roles now.
2. Team members who can grow into the roles in the time frame.

3. Internal transfers from within your company.

4. External hires who already have the skills.

That is probably an *ordered* list of how you should try to fill the key roles. This is true both because you want people who already know your culture and because reorganizations that depend on yet-to-be-hired individuals are much harder to pull off successfully.

Specifically, I'd recommend having a spreadsheet listing every single person's name, their current team, and their future team. Accidentally missing someone is the cardinal sin of reorganization.

3.7.6 Commit to moving forward

Now it's time to make a go decision. A few questions to ask yourself before you decide to fully commit:

1. Are the changes meaningful net positive?

2. Will the new structure last at least six months?

3. What problems did you discover during design?

4. What will trigger the reorg *after* this one?

5. Who is going to be impacted most?

After you've answered those questions, make sure to get not only your own buy-in but also buy-ins from your peers and leadership. Organizational change is rather resistant to rollback, so you have

to be collectively committed to moving forward with it, even if it runs into challenges along the way (which, if history holds, it almost certainly will).

3.7.7 Roll out the change

The final, and oftentimes most awkward, phase of a reorganization is its rollout. There are three key elements to a good rollout:

1. Explanation of reasoning driving the reorganization.
2. Documentation of how each person and team will be impacted.
3. Availability and empathy to help bleed off frustration from impacted individuals.

In general, the actual tactics for doing this are:

1. Discuss with heavily impacted individuals in private first.
2. Ensure that managers and other key individuals are prepared to explain the reasoning behind the changes.
3. Send an email out documenting the changes.
4. Be available for discussion.
5. If necessary, hold an organization all-hands, but probably try not to. People don’t process well in large groups, and the best discussions take place in small rooms.

6. Double down on doing skip-level one-on-ones.

And with that, you're done! You've worked through an engineering reorganization. Hopefully, you won't need to do that again for a while.

As a closing thought, an organization is both (1) a collection of people and (2) a manifestation of an idea separate from the individuals comprising it. You can't reason about organizations purely from either direction. There are many, exceedingly valid, different ways to think about any given reorganization, and you should use these ideas as *one* model for thinking through changes, not as a definitive roadmap.

3.8 Identify your controls

When I transitioned from directly supporting teams to instead partnering their managers, I struggled to remain effective without understanding their day-to-day tasks. My first instincts were to retain the same fidelity of context over a much wider area, and for the individuals working with me this was probably indistinguishable from micromanagement. Maybe it even *was* micromanagement.

Thanks to a great deal of feedback and reflection, I've gotten more deliberate at identifying where to engage and where to hang back, a process that I call *identifying your controls*.

Controls are the mechanisms that you use to align with other leaders you work with, and they can range from defining metrics to sprint planning (although I wouldn't recommend the latter). There is no universal set of controls—depending on the size of team and your relationships with its leaders, you'll want to mix and match—but the controls structure itself is universally applicable.

Some of the most common controls that I've seen and used:

Metrics[26] align on outcomes while leaving flexibility around how the outcomes are achieved.

Visions[27] ensure that you agree on long-term direction while preserving short-term flexibility.

Strategies[28] confirm you have a shared understanding of the current constraints and how to address them.

Organization design allows you to coordinate the evolution of a wider organization within the context of sub-organizations.

Head count and transfers are the ultimate form of prioritization, and a good forum for validating how organizational priorities align across individual teams.

Roadmaps align on problem selection and solution validation.

Performance reviews coordinate culture and recognition.

Etc. There are an infinite number of other possibilities, many of which are specific to your company's particular meetings and forums. Start with this list, but don't stick to it!

For whatever controls you pick, the second step is to agree on the *degree of alignment* for each one. Some of the levels that I've found useful are:

I'll do it. Stuff that I will personally be responsible for doing. When you're going to do something, it's better to be explicit and avoid confusion on responsibilities. Best used sparingly.

Preview. I'd like to be involved early and often. This is probably an area where we aren't quite on the same page and this will help us avoid redoing work.

Review. I'd like to weigh in before it gets published or fully rolled out, but we're pretty aligned on the topic.

Notes. Projects I'd like to follow but don't have much I can add to. Often used for wide-reaching initiatives on which we are well aligned, and I want to be able to represent my colleagues' work correctly.

No surprises. The work that we're currently aligned on but requires updates to keep my mental model in tact. If I'm asked about a related problem, I want to be able to answer it correctly. This is particularly important for me, as my effectiveness is evaluated based on my ability to stay on top of new problems.

Let me know. We're well aligned on this, since my colleagues have done it before and done it well. I want them to let me know if something comes up that I can help with, but otherwise I'm totally confident it'll go well, so we don't need to talk about this much.

Combine your controls and the degree of alignment for each, and you've established the interface between you and the folks you support. This reduces the ambiguity of how you work together and allows everyone to focus. It's useful for agreeing on performance goals, and is also very useful for exposing alignment gaps between you and individuals you work with (For example, it's a worrisome sign if you want to preview every bit of someone's work, unless you just started working together.)

Finally, this is a useful diagnostic for you as a leader to identify if you are micromanaging. If you simply can't imagine a world where you don't preview everyone's work, it's probably time to reflect a bit on what's holding you back from letting the team thrive.

3.9 Career narratives

A peculiar challenge of management is trying to invest in someone's career development when they themselves are uncertain about their goals. As a manager, you may have more experience and more access to opportunities within the company, but that represents a small slice of someone's career possibilities. Our schooling often rewards us for being methodical, linear thinkers, but that approach is less effective outside the intentionally constrained possibility spaces.

The options for approaching a career, particularly for those of us privileged to work in technology, are so extraordinarily vast that exploring them effectively requires a different approach. This vastness also means that you, as a manager, can't simply give folks a career path: you'll inevitably steer them toward the most obvious avenues and through avoidable competition.

Flipping perspectives, it's also quite challenging to plan your own career. I sometimes find myself walking from one meeting where I'm coaching someone on their career goals, straight into a second meeting where I struggle to string together words to articulate my own. The hardest bit is that most folks are always at the furthest point in their career, each change a step into the unknown, with limited visibility into the upcoming opportunities that their company can provide.

The intersection that I've found between the individual's and their manager's perspective is the *career narrative*. I've explained

these fabled documents a few times before, in “Roles over Rocket Ships”[29] and “Partnering with Your Manager,”[30] but given how useful they can be, it’s useful to expand a bit on the process of maintaining one.

3.9.1 Artificial competition

If you took 10 minutes to ask a dozen people about their immediate career goals, I suspect that for 11 of them it would center on either getting promoted or switching companies to reach the next evolution of their current job. This doesn’t mean that climbing the career ladder is bad—that’s what it’s designed for—but it has the side effect of funneling most folks toward a constrained pool of opportunity.

What I’ve slowly but increasingly come to believe is that there is much more opportunity on career ladders than off of them, and by including those opportunities you’ll make and *feel* more progress. Better yet, you’ll find far more opportunities to partner with your peers, no longer competing for limited promotion slots.

For example, if your long-term goal is to be the head of engineering at a mid size company, you could approach that linearly by trying to expand your role bit by bit at your current company on the track to becoming its head of engineering. That’ll work for roughly one person at the company, but for everyone else pursuing that same path it will probably be suboptimal.

A different approach would be to instead work on identifying the gaps that would keep you from being a strong head of engineering, and then start using your current role to help fill those gaps. A prototypical head of engineering will be skilled at organizational design, process design, business strategy, recruiting, mentoring, coaching, public speaking, and written communication. They'll also have a broad personal network and a broad foundation from product engineering to infrastructure engineering. That's not even a particularly complete list of relevant skills! There are so many different aspects to build out, and you can find opportunities to practice all of them in your current role. There's no need to convince yourself that your current role is holding you back—everything you need is here.

Importantly, while generalized career paths won't necessarily align cleanly with your goals, they are also unlikely to take full advantage of your strengths. An important part of setting your goals is developing areas you're less experienced in to maximize your global success, but it's equally important to succeed locally within your current environment by prioritizing doing what you do well.

With all of this in mind, take an hour and write up as many goals as you can for what you'd like to accomplish in the next one to five years. Then prioritize the list, pick a few that you'd like to focus on for the next three to six months, and share it with your manager at your next one-on-one.

3.9.2 Translating goals

Once you've identified goals to pursue, the next step is to translate those goals into actions, and this is where your manager can be a leveraged partner in iterating on your career narrative.

Managers tend to have a strong sense of the business's needs, and that gives them the superpower of finding the intersection of your interests and the business's priorities. That translation is a creative pursuit, so don't leave this entirely to your manager: participate as well! Brainstorm projects, research how folks at other companies have pursued similar goals, and educate your manager on aspects of your goals that they don't know much about. (For example, engineers often have more conference speaking experience than engineering managers do.)

Bringing your list of goals to this discussion helps ensure that it's successful. If you don't bring a rough draft, by default you'll get steered toward the contested commons, and your career narrative will be a dull instrument for progress.

This refined list of goals, aligned to your company's priorities, then becomes a central artifact for how you and your manager collaborate on your career evolution. Every quarter or so, take some time to refresh the document and review it together.

If you're unconvinced that it's worth your time to write a career narrative, you certainly don't have to write one. Most folks get away with not writing one for their entire career and have deeply fulfilling careers despite the absence.

That said, if you don't, then there is probably no one guiding your career. Chasing the next promotion is at best a marker on a mass-produced treasure map, with every shovel and metal detector re-

covering the same patch. Don’t go there. Go somewhere that’s disproportionately valuable to you because of who you are and what you want.

3.10 The briefest of media trainings

When I was working at Digg,[31] I was fortunate enough to get five minutes of media training from my colleague Christine. As a testament to her, that brief training lodged deeply in my head, and I've found myself repeating it frequently ever since. Eventually, I realized that I should probably just write it up!

The three rules for speaking with the media:

1. **Answer the question you want to be asked.** If someone asks a very difficult or challenging question, reframe it into one that you're comfortable answering. Don't accept a question's implicit framing, but instead take the opportunity to frame it yourself. *Don't Think of An Elephant* by George Lakoff [32] is a phenomenal, compact guide to framing issues.

2. **Stay positive.** Negative stories can be very compelling. They are quite risky, too! As an interviewee, find a positive framing and stick to it. This is especially true when it comes to competitors and controversy.

3. **Speak in threes.** Narrow your message down to three concise points, make them your refrain, and continue to refer back to your three speaking points.

That was it! Concise, compact, and I'm still using and learning from that advice a decade later.

3.11 Model, document, and share

Early on in my career, I spent a lot of time trying to find *my* leadership style. Recently, I think it's more useful to think about growing yourself as a leader by developing a range of styles and applying them thoughtfully to your circumstances. Confining yourself to one style is just too hard.

One of the trickiest, and most common, leadership scenarios is leading without authority, and I've written about one of the styles that I've found surprisingly effective in those conditions. I call it **Model, Document, Share.**

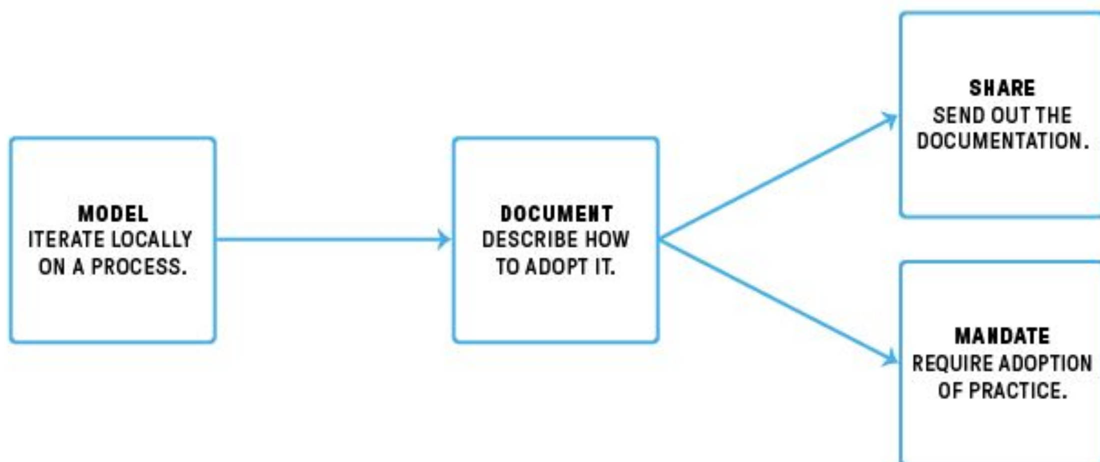


Figure 3.7
The Model, Document, Share-Mandate approach.

3.11.1 How it works

Imagine that you've started a new job as an engineering manager, and the teams around you are too busy to use a planning process. You've mentioned to your peers a few times that you've seen kanban[33] work effectively, but folks tried it two years ago and are still upset whenever the word is mentioned: it just doesn't work here.

Your first reaction might be to confront this head-on, but it takes a while to build credibility after starting a new job. Sure, you've been hired for your experience, so they respect your judgment, but it's a hard sell to convince someone that your personal experience should invalidate their personal experience.

I've been trying something different.

Model. Start measuring your team's health (maybe using short, monthly surveys) and your team's throughput (do some lightweight form of task sizing, even if you just do it informally with a senior engineer on the team or with yourself), which will allow you to establish the baseline before your change.

Then just start running kanban. Don't publicize it, don't make a big deal about it, just start doing it with your team. Frame it as a short experiment with the team, and start trying it. Keep iterating on it until you're confident it works. Have the courage to keep at it for a while, and also the courage to stop doing it if it doesn't work after a month or two. Use the team's health and throughput metrics to ground your decision about whether it's working.

Document. After you've discovered an effective approach, document the problem you set out to solve, the learning process you went through, and the details of how another team would

adopt the practice for themselves. Be as detailed as possible: make a canonical document, and even get a few folks on other teams to check that it's readable from their perspective.

Share. The final step is to share your documented approach, along with your experience doing the rollout, in a short email. Don't ask everyone to adopt the practice, don't lobby for change, just present the approach and your experience following it.

You'll spend the majority of your time refining approaches that work effectively for your team, a bit of your time documenting how you did it, and almost no time trying to convince folks to change their approach.

Strangely, in my experience, this has often led to more adoption than top-down mandates have.

3.11.2 Where it works

When considering how the **Model, Document, Share** approach works, it's interesting to compare it with the top-down mandate.

Mandates assume:

It's better to adopt a good-enough approach quickly.

- Teams have the bandwidth to adopt a new approach.
- The organization has available resources to coordinate a rollout.
- You want to centralize decision-making on this topic.

- Consistency is important; all teams need to approach this problem in the same way.
- It's important to make this change quickly.

Model, Document, Share assumes:

It's better to adopt a great approach slowly.

- Some teams don't have the bandwidth to adopt a new approach.
- The organization may not have resources to coordinate a rollout.
- You want to decentralize decision-making on this topic.
- Teams have agency to adopt the appropriate practices for themselves.
- It's okay to approach change gradually.

If your circumstances and your organization's values align with the second list, then this approach may be *more* effective for you than making mandates. If you have the time, you can slowly flock toward great practice, without needing organizational authority. (You'll still need some natural authority, the respect of your peers.)

Although I've seen this approach work remarkably well, I've also seen it go nowhere. It's a particular tool for certain circumstances, and it does fail. It may be an inexpensive failure—folks simply don't adopt—as you haven't spent much authority on

it, but nonetheless you still haven't accomplished your goal. It's particularly important not to try to use this as a strategy to circumvent organizational authority. Operating in direct conflict with authority usually doesn't end very well.

3.12 Scaling consistency: designing centralized decision-making groups

In small organizations, it's easy for individuals to be aware of what others are doing and to remember how they've previously approached similar problems. This hive mind and memory create decision-making whose consistency correlates strongly with quality. As organizations grow, there is a subtle slide into inconsistency, which is often one of the most challenging aspects of evolving from a small team into a much larger one.

There are many different approaches to try to manage inconsistency creep. Some of the solutions I've seen are formalized sprints, training, shadowing, documentation, code linters,[34] process automation (particularly deploys), and incident reviews. However, when the problem becomes truly acute, folks eventually reach for the same tool: adding a centralized, accountable group.

The two most common flavors of this I've seen are "product reviews" to standardize product decisions and the "architecture group" to encourage consistent technical design. There are hundreds of varieties, and they crop up wherever decisions are made.

Some of these groups take on an authoritative bent, becoming rigid gatekeepers, and others become more advisory, with a focus on educating folks toward consistency. Depending on your culture and how you value consistency, there are an infinite

number of approaches, and designing an effective decision-making group depends on a handful of core decisions.

3.12.1 Positive and negative freedoms

Before jumping into design, a few words on the framing that I use for reasoning about when to create a new centralized authority.

These groups typically consolidate significant authority from the broader community into the hands of a few. Many folks will feel a significant loss of freedom when you create these groups, as their zone of decision-making will be newly limited. Less obviously, many folks find the creation of centralized groups to be extremely empowering. The difference? One group is largely populated by individuals comfortable with self-authorization, and the latter typically have a higher threshold for self-authorization.

This is just one example of the dynamics that play out across many dimensions when you're considering introducing a new authority, and the most useful framework I've found for thinking through this involves *positive* and *negative freedoms*. A *positive freedom* is the freedom to do something, for example the freedom to pick a programming language you prefer. A *negative freedom* is the freedom from things happening to you, for example the freedom not to be obligated to support additional programming languages, even if others would greatly prefer them.

How are you shifting freedoms, and who are you shifting them from? Particularly in cases where ownership is extremely diffuse, I believe that cautiously authoritative groups do increase net positive freedom for individuals without greatly reducing negative

freedom. That also happens to be my goal when designing a new group!

3.12.2 Group design

Now that you've decided to create a decision-making group, it's time to get into the choices!

Influence. How do you expect this group to influence results? Will they be an authoritative group that makes binding decisions? Will you rely on the natural authority of the members you select? Will they primarily work through advocacy? The answers to these questions along with the particular folks in the group, will be the primary factor in how your group impacts the positive and negative freedoms of those they work with.

Interface. How will other teams interact with this team? Will they submit tickets, send emails, attend a weekly review session? Will you be reviewing work before it launches, or previewing designs before they're staffed? Depending on the kind of influence they're exerting and how your company works, you'll want to play around with different approaches.

Size. How large should the group be? If it's six or fewer individuals, it's possible for them to gel into a true team, one whose members know each other well, work together closely, and shift a significant portion of their individual identities into the team. If the group has more than ten members, you'll find it hard to even have a good discussion, and it'll need to be structured into sub-groups to function well (rotation that spreads members over time, working groups of some members to focus on

particular topics, and so on). The larger the group, the more responsibility becomes diffuse, and the more you'll need to have specified roles within the group, for example a member responsible for coordinating members' focus.

Time commitment. How much time will members spend working in this group? Will this be their top priority, or will they still primarily be working on other projects? Higher time commitment correlates with more actions and decisions. Consequently, my sense is that you want time commitment to be higher for areas where folks are directly impacted by the consequences of their decisions, and to be lower for scenarios with weaker feedback loops.

Identity. Should members view their role in the group as their primary identity? Should they continue to identify primarily as members of their existing team? You'll need a small team and high time commitment to support individuals shifting their identity.

Selection process. How will you select members? I've found the best method to be a structured selection process,[35] in which you identify the requirements to be a member and the skills that you believe will be valuable, and then allow folks to apply. Membership in these groups often becomes an important signal of organizational status, which makes having a consistent process for selecting membership especially important.

Length of term. How long will members serve? Are these permanent assignments, or are they fixed terms for, say, six months? If they are fixed terms, are folks eligible for subsequent

elections? Is there a term limit? I've tried most combinations, and my sense is that the best default is fixed terms, while allowing current individuals to remain eligible, and without enacting term limits.

Representation. How representative will this group be? Will you explicitly select folks based on their teams, tenure, or seniority, or will you allow clusters? Attention to this can help you avoid architecture reviews that are missing front-end and product engineers, as well as product reviews without infrastructure perspective.

Predictably, each of these decisions will impact the effectiveness of the others, which can make designing the group you want quite tricky. Some formats will require particular kinds of individuals to staff them, and you have to design groups that will work with the people available to participate and the culture they are participating within.

3.12.3 Failure modes

Before you release that email you're writing to spin up a new centralized decision-making group, it's worth talking about the four ways these groups consistently fail. They tend to be domineering, bottlenecked, status-oriented, or inert.

Domineering groups significantly reduce individuals' negative and positive freedoms, and become churn factories for members. This is most common when those making decisions are abstracted from the consequences of the decisions, e.g., architecture groups in which the members write little code.

Bottlenecked groups tend to be very helpful, but are trying to do more then they're actually able to do. These groups get worn down, and either burn out their members or create a structured backlog to avoid burning themselves out, which ends up seriously slowing down folks who need their authority.

Status-oriented groups place more emphasis on being a member of the group than on the group's nominal purpose. The value of the group revolves around recognition rather than contribution, leading to folks who try to join the group for status, and the diffusion of whatever original mission the group set out to resolve.

Inert groups just don't do much of anything. Typically, these are groups whose members have not gelled or are too busy. On the plus side, this is by far the most benign of the four failure modes—but you're also missing out on a great deal of opportunity!

Having experienced each of these a few times, I try to ensure that there is a manager embedded into every centralized group, and that the manager is responsible for iterating on the format to dodge these pitfalls.

3.13 Presenting to senior leadership

Yahoo! BOSS[36] was partially powered by an internal Yahoo! search technology named Vespa. We'd run into a bunch of challenges, and I'd decided to convince my team we should migrate to SOLR.[37] My manager asked me to put together a presentation for our next team meeting. The meeting came, I started to present, and within two slides things fell apart.

"No, no, this isn't the way to put this together. This is like an academic presentation. You have to start from the conclusion first," my manager lamented as he abruptly terminated my presentation. Pausing to brush my deck's fragments from his boots, he offered some final wisdom: "And don't use curved lines in your diagrams. Those *never* make sense."

It took me a few years to glean a lesson from that experience, but it comes to mind frequently now when I work with folks who are just starting to present to executives. Giving a presentation to senior leadership is a bit of a dark art: it takes a while to master, and most people who do it well can't quite articulate how they do it. Worse yet, many people who are excellent rely on advantages that resist replication: charisma, quick wit, deep subject matter expertise, or years of experience.

That said, few people watching me bomb my Yahoo! presentation would have bet I'd ever figure this one out, so you should know that it is a learnable skill. Along the way, I've picked up some tips that I hope will help you prepare for your next presentation:

Communication is company-specific. Every company has different communication styles and patterns. Successful presenters probably can't tell you what they do to succeed, but if you watch them and take notes, you'll identify some consistent patterns. Each time you watch individuals present to leadership, study their approach.

Start with the conclusion. Particularly in written communication, folks skim until they get bored and then stop reading. Accommodate this behavior by starting with what's important, instead of building toward it gradually.

Frame why the topic matters. Typically, you'll be presenting on an area that you're intimately familiar with, and it's probably very obvious to you why the work matters. This will be much less obvious to folks who don't think about the area as often. Start by explaining why your work matters to the company.

Everyone loves a narrative. Another aspect of framing the topic is providing a narrative of where things are, how you got here, and where you're going now. This should be a sentence or two along the lines of, "Last year, we had trouble closing several important customers due to concerns about our scalability. We identified our databases as our constraints to scaling, and since then our focus has been moving to a new sharding model that enables horizontal scaling. That's going well, and we expect to finish in Q3."

Prepare for detours. Many forums will allow you to lead your presentation according to plan, but that is an unreliable prediction when presenting to senior leadership. Instead, you need to be prepared to lead the entire presentation yourself, while

being equally ready for the discussion to derail toward a thread of unexpected questions.

Answer directly. Senior leaders tend to be indirectly responsible for wide areas, and frequently pierce into areas to debug problems. Their experience "debug piercing" tunes their radar for evasive answers, and you don't want to be a blip on that screen. Instead of hiding problems, use them as an opportunity to explain your plans to address them.

Deep in the data. You should be deep enough in your data that you can use it to answer unexpected questions. This means spending time exploring the data, and the most common way to do that is to run a thorough goal-setting exercise.[38]

Derive actions from principles. One of your aims is to provide a mental model of how you view the topic, allowing folks to get familiar with how you make decisions. Showing you are "in the data" is part of this. The other aspect is defining the guiding principles you're using to approach decisions.

Discuss the details. Some executives test presenters by diving into the details, trying to uncover an area the presenter is uncomfortable speaking on. You should be familiar with the details, e.g., project statuses, but I think that it's usually best to reframe the discussion when you get too far into the details. Try to pop up to either the data or the principles, which tend to be more useful conversations.

Prepare a lot; practice a little. If you're presenting to your entire company, practicing your presentation is time well spent. Leadership presentations tend to quickly detour, so practice isn't

quite as useful. Practice until you're comfortable, but prefer to prepare instead getting deeper into the data, details, and principles. As a corollary, if you're knowledgeable in the area you're responsible for, and have spent time getting comfortable with the format, over time you'll find that you won't need to prepare much for these specifically. Rather, whether you're able to present effectively without much preparation will become a signal for whether you're keeping up with your span of responsibility.

Make a clear ask. If you don't go into a meeting with leadership with a clear goal, your meeting will either go nowhere or go poorly. Start the meeting by explicitly framing your goal!

That's a lot to remember, so I've synthesized these ideas into a loose template. There absolutely is not a single right way to present to senior leaders, but hopefully the template is a useful starting point.

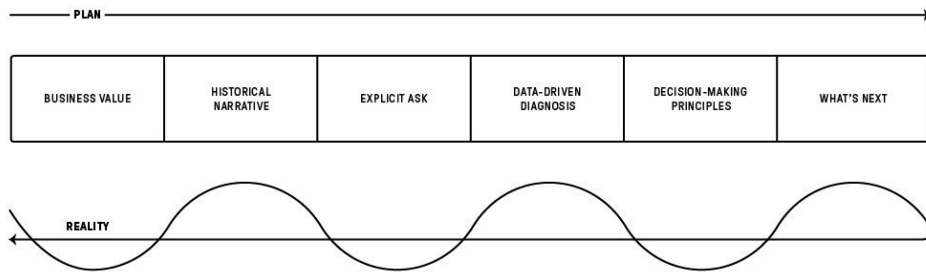


Figure 3.8
The expected and actual experience of presenting to executives.

My general approach to presenting to senior leaders is:

1. **Tie topic to business value.** One or two sentences to answer the question "Why should anyone care?"
2. **Establish historical narrative.** Two to four sentences to help folks understand how things are going, how we got here, and what the next planned step is.
3. **Explicit ask.** What are you looking for from the audience? One or two sentences.
4. **Data-driven diagnosis.** Along the lines of a strategy's diagnosis phase,[39] explain the current constraints and context, primarily through data. Try to provide enough raw data to allow people to follow your analysis. If you only provide analysis, then you're asking folks to take you on trust, which can come across as evasive. This should be a few paragraphs, up to a page.
5. **Decision-making principles.** Explain the principles that you're applying against the diagnosis, articulating the mental model you are using to make decisions.
6. **What's next and when it'll be done.** Apply your principles to the diagnosis to generate the next steps. It should be clear to folks reading along how your actions derive from your principles and the data. If it's not, then either rework your principles or your actions!
7. **Return to explicit ask.** The final step is to return to your explicit ask and ensure that you get the information or

guidance you need.

I’ve had a lot of luck with this format in general, and I think you’ll find it pretty useful as a starting point. That said, the first rule remains true: communication is company-specific. If things don’t quite work for this format in your company, then watch how other folks present. Given a few examples, you’ll be able to reverse engineer the discussions that go well into a workable template.

3.14 Time management

When you sit down for coffee with a manager, you can probably guess the biggest challenge on their mind: time management. Sure, time management isn't *always* everyone's biggest challenge, but once the crises of the day recede, it comes to the fore.

Time management is the enduring meta-problem of leadership. For most other aspects of leadership, you can look to more experienced managers and be reassured that things will get better, but in this dimension it appears that the most tenured folks are the ones most underwater. Yes, their degree of difficulty is certainly higher, but it's intimidating to consider that there's little evidence most folks ever get a solid grasp of their time.

Does that make this a lost cause? Nah.

I'm still pretty busy on a day-to-day basis, but I've gotten much, much better at getting things done, not by getting faster but by getting more logical about solving problems. The most impactful changes I've made to how I manage time are:

Quarterly time retrospective. Every quarter, I spend a few hours categorizing my calendar from the past three months to figure out how I've invested my time. This is useful for me to reflect on the major projects I've done, and also to get a sense of my general allocation of time. I then use this analysis to shuffle my goal time allocation for the next quarter.

Most folks are skeptical of whether this is time well spent, but I've found it particularly helpful, and it's the cornerstone of my efforts to be mindful of my time.

Prioritize long-term success over short-term quality. As your scope increases, the important work that you're responsible for may simply not be possible to finish. Worse, the work that you believe is most important, perhaps high-quality one-on-ones, is often competing with work that's essential to long-term success, like hiring for a critical role. Ultimately, you have to prioritize long-term success, even if it's personally unrewarding to do so in the short term. It's not that I like this approach, it's that nothing else works.

Finish small, leveraged things. If you're doing leveraged work,[40] then each thing that you finish[41] should create more bandwidth to do more future work. It's also very rewarding to finish things. Together, these factors allow large volumes of quick things to build into crescendoing momentum.

Stop doing things. When you're quite underwater, a surprisingly underutilized technique is to stop doing things. If you drop things in an unstructured way, this goes very poorly, but done with structure this works every time. Identify some critical work that you won't do, recategorize that newly unstaffed work as organizational risk,[42] and then alert your team and management chain that you won't be doing it. This last bit is essential: it's fine to drop things, but it's quite bad to silently drop them.

Size backward, not forward. A good example of this is scheduling skip-levels.[43] When you start managing a multi-tier

team, say 20 individuals, you can specify a frequency for skip-levels and reason forward to figure out how many hours of skip-levels you'll do in a given week. Say you have 16 indirect reports, and you want to see them once a month for 30 minutes, so you end up doing two hours per week.

This stops working as your team grows, because there is simply no reasonable frequency that won't end up consuming an unsustainable number of hours. Instead, specify the number of hours you're able to dedicate to the activity, perhaps two per week, and perform as many skip-levels as possible within that amount of time. This method keeps you in control of your time allocation, and it scales as your team grows.

Delegate working "in the system." Wherever you're working "in the system,"[44] design a path that will enable someone else to take on that work. It might be that this plan will take a year to come together, and that's fine, but what's not all right is if it's going to take a year and you haven't even started.

Trust the system you build. Once you've built the system, at some point you have to learn to trust it. The most important case of this is handing off the responsibility to handle exceptions. Many managers hold onto the authority to handle exceptions for too long, and at that point you lose much of the system's leverage. Handling exceptions can easily consume all of your energy, and either delegating them or designing them out of the system is essential to scaling your time.

Decouple participation from productivity. As you grow more senior, you'll be invited to more meetings, and many of those

meetings will come with significant status. Attending those meetings can make you feel powerful, but you have to keep perspective about whether you're accomplishing much by attending. Sometimes, being able to convey important context to your team is super valuable, and in those cases you should keep attending, but don't fall into the trap of assuming that attendance is valuable.

Hire until you are slightly ahead of growth. The best gift of time management that you can give yourself is hiring capable folks, and hiring them before you get overwhelmed. By having a clear organizational design, you can hire folks into roles before their absence becomes crippling.

Calendar blocking. Creating blocks of time on your calendar is the perennial trick of time management: add three or four two-hour blocks scattered across your week to support more focused work. It's not especially effective, but it does work to some extent and is quick to set up, which has made me a devoted user.

Getting administrative support. Once you've exhausted all the above tools and approaches, the final thing to consider is getting administrative support. I was once quite skeptical of whether admin support is necessary—and, until your organization and commitments reach a certain level of complexity, it isn't—but at some point having someone else handling the dozens of little interruptions is a remarkable improvement.

As you start using more of these approaches, you won't immediately find yourself less busy, but you will gradually start to finish more work. Over a longer period of time, though, you can get less busy by prioritizing finishing things with the goal of

reducing load. If you’re creative and consequent, and if you don’t fall into the trap of believing that being busy is being productive, you’ll find a way to get the workload under control.

MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY	SATURDAY	SUNDAY
PREP		PLANNING OFFSITE	INCIDENT REVIEW	FOCUS BLOCK	NOPE	NOPE
STAFF MEETING	INTERVIEW					
	1:1		1:1	INTERVIEW		
LUNCH	LUNCH	LUNCH	LUNCH	LUNCH		
HEADCOUNT PLANNING	FOCUS BLOCK	INTERVIEW	FOCUS BLOCK	USER CHAT		
		USER CHAT		1:1		
	1:1	1:1	INTERVIEW			
	SKIP LEVEL 1:1	1:1	SKIP LEVEL 1:1			

Figure 3.9
Calendar time-blocking for an engineering manager.

3.15 Communities of learning

I’ve always preferred learning in private. Got something difficult? Sure, leave me alone for a few hours and I can probably figure it out. If you want me to figure it out with you watching, I’m not even sure how to start. This is partly introversion, but altogether I’m pretty uncomfortable making mistakes in public. Like a lot of folks, I have a brain that still helpfully reminds me of public errors I made decades ago, and they still bother me.

For a very long time, this discomfort prevented me from discovering one of the most rewarding elements of being in a supportive work environment: building a community of learning with your peers. This works especially well in a gelled “first team,”[45] and, recently I’ve been spending more time facilitating a broader learning community of engineering managers.

When I first started facilitating the group, we focused on content-rich presentations. Each slide was dense with important lessons and essential details. It didn’t work well. Folks weren’t engaged. Attendance dropped. Learning was not the order of the day.

Since then, we’ve iterated on format and eventually stumbled on an approach that has worked consistently:

1. **Be a facilitator, not a lecturer.** Folks want to learn from each other more than they want to learn from a single presenter. Step back and facilitate.

2. **Brief presentations, long discussions.** Present a few minutes of content, maybe five, and then move into discussion. Keep the discussions short enough that even if a group doesn't get traction on a given topic, it doesn't become awkward. A good time limit would be 10 minutes.

3. **Small breakout groups.** Giving folks time to discuss in small groups allows them to learn a bit about the topic in a small, safe place. It also gives everyone an opportunity to be part of the discussion, which is a lot more engaging than listening to others for an hour.

4. **Bring learnings to the full group.** After discussions, give each group an opportunity to bring their discussion back to the larger group, to allow the groups to cross-pollinate their learnings.

5. **Choose topics that people already know about.** Successful topics are ones that people have already thought about, typically because these concepts are core to their daily work. Ideally the topic is something that they do but would like to get better at, such as one-on-ones, mentorship, coaching, or career development.

   People find it very hard to discuss content that they've just learned if it's too far away from their previous experience. That also creates an environment where learning has to come from the facilitator instead of from the group at large.

6. **Encourage tenured folks to attend.** For many learning communities, you'll find that the most senior or most tenured folks opt out to focus on other work. This is a shame, because

there is so much for them to teach newer folks, and also because it creates an opportunity for them to learn from and get to know new members.

7. **Optional pre-reads.** Some folks aren't comfortable being introduced to a new topic in public, and for those individuals, providing a list of optional pre-reads can help them prepare for the discussion. I find that most people don't read them (which, surprisingly, I also found true when hosting paper-reading groups[46]), but for some folks they're very helpful.

8. **Checking in.** Depending on the size of the group, it can be powerful to start by checking in with each other, having each person give a 20- or 30-second self-introduction. The format we've been using lately is your name, your team, and one sentence about what's on your mind. This is especially useful in quickly growing communities, as it makes it easier for individuals to meet each other.

The thing I enjoy most about this format is that it gives folks what they really want, spending time learning from each other, and is remarkably quick for the facilitator to prepare. I'm far from a seasoned facilitator, and I've also found this format to be a rewarding and safe opportunity for me to grow as a facilitator.

If your company doesn't have any learning communities, give it a try. I've found this one of the easiest, most rewarding things I've had the opportunity to work on.

	CHECK IN	OPPORTUNITIES	DISCUSSION	REGROUP
COMMUNITIES OF LEARNING	SHARE: YOUR NAME, YOUR TEAM, 1 SENTENCE ON TOP PRIORITY.	A LOT OF LEARNING IS TOP-DOWN. WE CAN LEARN A LOT FROM PEERS. LEARNING AS A HABIT, LEARNING IN A COMMUNITY.	BREAK INTO GROUPS OF 4–5. WHERE DO YOU LEARN TODAY? WHO DO YOU LEARN FROM? REGROUP IN 10 MINUTES.	SUMMARY FROM EACH GROUP.

Figure 3.10
Structuring a meeting to facilitate a community of learning.