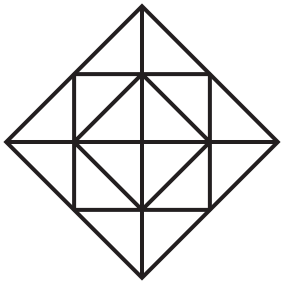
The **Superstruct** *Manifesto*

The **Superstruct** *Manifesto*



A Survival Guide for Founders Who Depend on Devs to Get Things Done

DAVID GUTTMAN

SUPERSTRUCT INTERNATIONAL, INC. | CHEYENNE, WY

David Guttman is a developer, author, and consultant obsessed with building repeatable systems for recruiting, onboarding, and managing remote software engineers.

Over the course of his 20+ years in software development, David has led teams of engineers to ship projects like Project Matterhorn, the system that would eventually become Disney.com, an LMS that *TIME* magazine listed as one of the Best Inventions of the Year 2020, and video ad servers that handle over 10 billion requests per day.

David is a leader in the tech community and has helped thousands of devs level-up as the organizer of the js.la monthly event series, the host of the *Junior To Senior* podcast, and a champion in the Node.js mentorship initiative.

David is the author of two popular JavaScript books, has over 90 open-source packages on npm, and has given talks at tech events and conferences like JSConf and JSFest.

Printed in the United States of America

First Edition, 2023

30 29 28 27 26 25 24 23 1 2 3 4

Library of Congress Control Number: 2023905802

ISBN 13: 979-8-9878466-0-5 (cloth)
ISBN 13: 979-8-9878466-1-2 (paperback)

Dedication

"It's not because things are
difficult that we dare not venture.
It's because we dare not
venture that they are difficult."

Seneca

CONTENTS

INTRODUCTION

Too many founders fail for avoidable reasons. This is why I wrote this book. There are enough things that can go wrong that are out of our control. The last thing your company needs are complications, headaches, and disasters on top of what you'd have to deal with naturally.

This is your survival guide. This is not a cookbook. There are no recipes or step-by-step instructions. Instead, I will show you how to avoid the worst and most common mistakes I see founders make when hiring and working with software engineers.

Many readers will be alarmed by my warnings. They will have heard of big companies or unicorn start-ups doing exactly what I caution against. If you are a big company or a unicorn startup that can afford to keep doing what you're doing, by all means, don't let me stop you. This book is not for you.

In nature, there are plenty of plants and mushrooms that are poisonous or deadly if not handled correctly. The fanciest Japanese restaurants in the world may serve pufferfish, but that's not an invitation for you to try making it at home. Many founders blindly copy famous tech company behaviors and needlessly suffer for it. I'll point these out.

Above all else, one thing has become clear to me: Founders struggle to hire and retain senior engineers. Many seem to think that their troubles will disappear if they can make that happen. The truth is that engineers can behave as seniors or as juniors, and this is largely unrelated to their technical skills. It's up to you to create an environment that encourages the behaviors that will make your company successful and eliminates those that will derail all meaningful progress.

If you try to measure an engineer by their years of experience with a programming language, database technology, or famous tech company, be careful. Those engineers may be capable of solving problems that you don't have and may not want.

To counteract this tendency, I'll show you what to look for and cultivate instead. Ultimately, if you want your engineers to behave like seniors, it's up to you to provide and enforce the appropriate structure. Some engineers may bristle at imposed structure. They want to eat ice cream for dinner and stay up late. When I was in the development trenches, I was like that too. However, engineers like my old self can cause too many distractions and deliver too little value if not channeled appropriately—no matter how "talented" they are.

WE WILL NOT
Inflict Daily Standups on Our Devs

Daily standups are as ubiquitous as they are stupid. I'm amazed that such an expensive and disruptive activity has become commonplace.

Daily standups guarantee that your devs will never have a single full day of meaningful work. This is a steep price to pay.

Yes, yes, I know standups help devs be productive. They increase transparency, highlight blockers, improve alignment, etc. … Even if I were to accept those things as true, standups are an irresponsibly expensive way to achieve those goals.

Why are standups expensive? Presumably, you didn't hire engineers because you had a bunch of empty meeting rooms. Your customers and clients needed features and fixes. You hired devs to build. Any time spent in standups is time not spent building.

If you have four devs standing around for 15 minutes every day, you are losing more than 20 hours of building time every month, *at a minimum*. Every single month, your product could have a new feature that pulls in revenue or reduces customer churn, but instead you just throw that opportunity in the trash.

If you have four devs standing around for 15 minutes every day, you are losing more than 20 hours of building time every month, *at a minimum*.

The sad truth is that it's way worse than this. Standups are supposed to only be 15 minutes long. In fact, that's why they're called standups: Their primary selling point is that they are short. Unlike other meetings, devs have to remain standing

so that they don't get too comfortable and drag the meeting out.

I'd like to pause here to reflect on how bonkers this is. We've all agreed that by its very nature, this type of meeting is inefficient, and our solution is to make it *even more uncomfortable*. I'm surprised nobody has suggested standing on hot coals to make them even quicker.

Unfortunately, even if we make standups uncomfortable enough to limit them to 15 minutes, their damage will always exceed that time. Why? Your devs can't and won't be productive right up until the standup starts and immediately after it ends. There are two main reasons for this: First, context-switching takes time; and second, devs will avoid working on anything complicated if they know they're about to be interrupted.

Think of software development as working at the bottom of the ocean. It takes the dev time to put on their scuba gear, check their oxygen tanks and seals, and slowly descend. Once they're down there, they can be highly productive. However, any interruption that causes them to surface, swim to the boat, and take all their gear off to have a conversation is hugely disruptive. If you knew you were going to get interrupted in 10 minutes, would you dive down or just wait in the boat? I'd wait in the boat.

So what does this mean? It means that even if you used next-gen torture techniques to pare the meeting down to a single minute, you'd have at least 30 minutes of disruption on either side. In other words, the absolute minimum cost of a single standup is more than an hour per dev. If you stick to 15 minutes per meeting, you're losing more than 100 hours a month of productive time with a team of four.

What do standups give us in exchange for all that lost productivity? There's an argument to be made that work is useless if it's on the wrong things. Meetings and planning are essential to make sure that devs are working on the correct priorities. Is it possible that engineers lose 15% of their working hours to standups but become 20% more effective? It's possible but unlikely.

One promise of standups is that devs will take accountability for their work. There's no better pressure than peer pressure, and surely no dev would slack off for a day knowing they'd have to admit that to their team. If you've actually been to a standup before, you'll know that devs can make the simplest tasks sound like epic accomplishments. Of course, that's assuming anyone is even paying attention.

Another benefit of standups is transparency and alignment. When Alice describes what she's working

on and mentions a blocker, Bob can take note and adapt if it affects the project he's responsible for. I'm not going to say that this *never* happens. But I will confidently say that no dev wakes up in the morning excited to listen to another dev talk about their previous workday. If you expect Bob to always give Alice his undivided attention, brace for disappointment.

The biggest standup wins are when one dev can save another dev time and frustration. If Bob mentions that he's going to be adding a new OCR library to the app, and Charlie happens to know that the library is a tire fire and will be a gigantic waste of time, that's a hugely valuable warning. If Charlie saves Bob 25 hours of hassle, he just "paid" for a whole week of standups right there. I wonder if standups are popular for the same reason people like slot machines.

You can get all the benefits of standups without the costs.

If this depresses you, I have good news: You can get all the benefits of standups without the costs. Nothing is stopping you from holding your devs accountable or from sharing what each dev is working on with your team. Nothing stands in the way of devs alerting their teammates to blockers. All of these actions are possible without adding a daily interruption to everyone's calendar.

2

WE WILL NOT
Test Devs with Computer Science Riddles

If you're thinking of using *Cracking the Coding Interview* to come up with interview questions when hiring candidates, don't.

Obviously it works for Google, Amazon, and Facebook—or at the very least it doesn't prevent them from maintaining gigantic profits.

So I suppose, if you have no shortage of incredibly talented candidates that you can incentivize with valuable RSUs, go nuts. On the other hand, if you're a startup or midsize company trying to build features for real customers, it's a dumb approach.

In 2015, Max Howell tweeted "Google: 90% of our engineers use the software you wrote (Homebrew), but you can't invert a binary tree on a whiteboard so fuck off."

Would Google have been better off hiring the author of Homebrew? Maybe, maybe not. But what I do know is that if you have a process that rejects someone who has a record of building wildly valuable and successful software because they can't correctly answer computer-science riddles, you'd better be in the business of answering computer-science riddles and not building valuable and successful software.

Think hard about what your company needs to do to be successful. Chances are it will boil down to making customers' lives better and easier to the point that they're thrilled to give you money in exchange. It's tempting to think that people who have mastered algorithms and data structures will be able to build software that makes customers happy—and to build it faster and better than people who lack such knowledge. It might also be tempting to think that if these hiring processes

work for Google and Facebook, then surely they'll work well for you too. In reality, this is an odd bank shot to take.

Here's the deal: The farther you get from testing what you actually care about, the worse off you are.

Interviewing candidates is an instance of a larger class of problems. Classification, screening, and detection all take the same shape; we interact with these sorts of tasks all the time in daily life.

The farther you get from testing what you actually care about, the worse off you are.

For a dramatic example, let's look at airport security. It's difficult to efficiently predict if any one person passing through a security checkpoint will go on to do something horrible. In the wake of 9/11, some people argued that it would be more efficient to use racial profiling to give more or less scrutiny to particular ethnic groups. The thinking went something like this: Members of group A (Muslims) are more likely to have behavior B (terrorism), and therefore we should focus attention on group A to have a better chance of detecting behavior B.

Even leaving aside cultural values and the harmfulness of stereotypes, this strategy doesn't even make basic security sense. Because it introduces

additional complexity in detecting members of group A (since someone's Muslim beliefs are invisible), it increases the cost and reduces the security of the entire system. Not only that, but by establishing a clear profile that will get less scrutiny, it makes it easier for bad actors to evade detection.

Proponents of the racial-profiling system support it because it's "common sense." Someone who looks like Betty White obviously isn't going to blow up a plane, so it's a complete waste of time to spend time screening her if she makes the X-ray machine beep. Unfortunately, trusting our gut and following "common sense" are bad ideas when it comes to security and detection. They lead us astray.

It might be "common sense" that someone who can invert a binary tree would be more likely to be a member of group A (developers who know CS fundamentals), and group A is very likely to have behavior B (the ability to create value for customers). However, building an interview process around this is asinine.

If you want to predict accurately whether or not a candidate will be able to create value for customers or your company, then that's what you need to test for. This requires creativity. In that sense, it may not be as easy as having someone whiteboard bubble sort or answer a hard logic puzzle—because

you can't plagiarize other companies' superficially clever hiring tests. But unless your company bubble sorts and solves logic puzzles for customers, a precise screening mechanism will give you a much better signal than an off-the-shelf set of tests.

Hopefully, we can agree that if you need work done on your house, it's not a good idea to hire a contractor the Google way. Yes, even though your house is unique and will present special challenges, you don't need to ask hypothetical questions to gauge how well they can think on their feet and adapt. You also don't need to quiz them on fundamentals. You just need to see examples of previous work that is similar to what you need to be done.

You want someone who can knock real features off your real roadmap. Test for that.

Software is different, but it's not *that* different. You might think that you want someone who knows CS fundamentals and who can adapt to any situation, but you don't. You want someone who can knock real features off your real roadmap. Test for that. And if your roadmap is too much in flux for that, you have bigger problems to solve before you start hiring devs.

3

WE WILL NOT

Recruit 10x Developers

It's become conventional wisdom that you should only hire "10x developers." If you try to do this, you're setting yourself up for failure.

If you have an investor or other executive pushing you to hire 10x developers, politely thank them for their input and then ignore everything else they say about recruiting and managing software engineers.

Are 10x developers a myth? Short answer: yes. The longer answer is more complicated, but it doesn't change much.

The concept originates in a study from the 1960s that indicated there are developers who can perform coding tasks 10 times faster than peers with comparable experience and salary. If you take this at face value (and you shouldn't), it's understandable that you'd prefer to get 10x the benefit at no additional cost. But there are several reasons why you shouldn't get your hopes up.

To get a sense of what programming was like at the time, here's a quote from the study: "TSS utilizes an IBM AN/FSQ-32 computer. User programs are stored on magnetic tape or in disk file memory. When a user wishes to operate his program, he goes to one of several teletype consoles; these consoles are direct input/output devices to the Q-32. He instructs the computer, through the teletype, to load and activate his program. The system then loads the program either from the disk file or from magnetic tape into active storage (drum memory)."

And for a sense of scale: This Q-32 computer weighed up to 90 tons and occupied nearly an entire floor of a large office building. All this for a jawdropping-at-the-time 1.5 megabytes of memory. I'll leave you to decide how smart it is to make

hiring assumptions based on a five-decade-old study of 12 programmers working with a computer like this.

When people think of a 10x developer they think of 10x the *average* developer. The fastest developers in this study are an order of magnitude faster than the *worst* developers, not the *average* developer. In fact, average scores in the study were much closer to the best scores than the worst. This means that slow developers have outsized influence on this metric. I'm surprised we don't hear just as many people dispensing sage advice to avoid 0.1x developers.

I'm surprised we don't hear just as many people dispensing sage advice to avoid 0.1x developers.

Second, this whole concept rests on the premise that these more productive developers are no more costly to hire than average developers. If we think about this for more than a minute, it can't be true if the performance difference is obvious to the business.

Let's pretend that there are engineers out there who can complete any given task 10 times faster than their teammates. How long could that continue? Any manager in their right mind would not let this miracle engineer sit idle for 90% of the week.

However, if they didn't, this 10x dev would be doing 10x the work of everyone else. This would be very noticeable.

Any manager who sees this productivity would worry about losing this superstar. How long would that dev be making the same salary? Considering how productive they are, the most rational thing to do would be to fire the weakest engineer and give their salary as a raise to the 10x dev. Paying double would be a bargain for such an effective pair of golden handcuffs. Sadly, this wrecks the foundation of the concept. If 10x developers don't make more than their peers, it's because the business doesn't notice. And if the business doesn't notice, are they really 10x developers?

Any manager who sees this productivity would worry about losing this superstar.

At this point we could probably stop, but I'd like to share the simple reason why that study's large differences in programmer ability doesn't translate to working teams. Studies evaluate multiple programmers on the same task and then draw conclusions. If, instead of attributing the increased performance to the developer, we attribute it to the *combination* of developer and task, things make more sense.

A developer who has worked on UI rendering problems before will solve a similar UI rendering problem way faster than a developer who hasn't. This doesn't mean that the same speedy developer will knock a database query optimization problem out of the park. In other words, the 10x developer mantle is situational, and it will only show up when the stars of talent and task align. Therefore, it's no surprise that over the course of a year, you don't have devs that are doing 10x the work of everyone else. They will crush some projects, but not all of them.

Am I telling you this because I think that one dev can't do the work of 10 in the same amount of time? Of course not. I know that it's possible. In fact, I'm a 20x dev.

As a test, I recently showed a project manager a video editing app that I built, and I asked him to estimate how long it would take a team of his devs to re-create it. He came back with an estimate of 200 hours (5 devs × 40 hours each). I built that app in less than 10 hours. Therefore, in this case I'm a 20x developer.

So why not just hire a bunch of developers like me? I'm not going to be nearly that fast on 99% of the work teams need to accomplish. If I cherry-pick projects that particularly suit my skills, I'm going to

be way faster than most teams. However, even that app wasn't a fully-fledged money-making product. If I were to race a team to build out user accounts and payment and terms of service and logging and backups and admin tools and everything else that real products need, I would lose. Badly.

In fact, most of my speed came from knowing what I wanted to build and how ahead of time. If I had written down how I wanted to build the app and given that to the team, they too would have been able to build the app quickly. We'd get a much better result if I spent an hour documenting how the project should be built and letting the team do the actual coding.

In other words, my true value here was not being a developer. It was being a commando.

One of my favorite models for software development is Commandos, Infantry, and Police. When people think of 10x developers, they're thinking of commandos. Commandos love coming up with novel approaches to hard problems. When people fawn over the idea of the 10x developer, they recount stories of clever solutions that normal developers couldn't duplicate even if they were given 1,000x more people or time, let alone 10x.

This is what commandos do. They can establish a beachhead with astounding speed. Other engineers would never even think to parachute behind enemy lines or swim ashore at night.

If your business is R&D, constantly creating new proofs of concept, or rapid prototyping, commandos are valuable. Outside these types of organizations, you'll want creativity and novel approaches, and even in more traditional environments, you'll want to innovate. However, 99.9% of software development does not need commandos.

Having too many commandos is a problem. For each clever idea, there are countless hours of boring work. Most software engineering is executing proven designs. You don't want all your bridges to be wacky one-of-a-kind designs.

Having too many commandos is a problem. For each clever idea, there are countless hours of boring work.

Commandos don't want to do the routine work. They prefer to only do what only they can do. Using them for routine building creates more problems than it solves. Even the most novel products in Al, AAA games, still need "boring" build tools, admin dashboards, and basic sign-up forms. Commandos won't do any of those things more quickly. They will just cause drama.

Commandos get bored and chew the furniture. They want to write their own frameworks. They want to try brand-new databases, and they want to rewrite entire parts of your app. If you don't have big enough problems for them, they will create them for you.

If you want someone like that, you don't need them to write your production code. Nor do you need many of them. Each commando will create mountains of work—work that should be done by engineers who can get things done.

You want solid team players. You want infantry.

Optimize for predictability, not the extremes. The devs you hire should be able to work with product to understand the desired outcome. Whether it's a feature or a fix, they can plan out multiple approaches to make it happen, evaluate tradeoffs, document the changes, and do it all again. Sometimes when planning they'll find something that can benefit from refactoring or optimization, and they'll either build that time into the estimate or handle it after the feature is shipped. Sometimes they'll tell product that if the outcome is tweaked, they can ship faster (relaxing constraints).

Optimize for predictability, not the extremes.

Depending on what you're doing, you may even want police instead of infantry. They're adept at routine maintenance and tasks that follow a checklist. This is the absolute worst fit for a 10x engineer. Applying security patches and going through to update all the places it affects is not creative or exciting. The commando's tendency to improvise is not helpful.

If you have a single thing that is difficult, teams may not help. Teams are good with problems that are parallelizable. In business, one-off, never-been-done-before problems are rare. Don't optimize for or have too many people for that purpose.

Enlist a good manager, CTO, architect, or consultant when issues arise. Ask this person to create a novel proof of concept, write up a design doc based on their previous experience, or lead conversations with the devs that force them to think creatively and to be proactive about likely issues.

4

WE WILL NOT
Let Devs Start without an Estimate

Software estimates are notoriously difficult. Devs hate giving them. They're rarely correct. Many organizations abandon them entirely. Are they worth the trouble?

Absolutely. But not for the reason you think.

The most common reason to produce an estimate is for planning and coordinating with other initiatives. Sometimes this is necessary, and sometimes it isn't. But that's not what I'm talking about.

Developer estimates are one of the most powerful management tools you have at your disposal. If you're not using them, you're going to experience some of the most common and frustrating problems when trying to get your projects out the door.

One of the wildest things you'll notice is that simply requiring estimates will immediately improve the quality of your projects. It doesn't even matter if you use that estimate for planning. You could ask a developer for an estimate in a sealed envelope and throw it in a fire before even looking at it. This would still be a huge improvement over not requiring estimates. How is this possible?

Simply requiring estimates will immediately improve the quality of your projects.

Estimates force a developer to think about the entire project end to end. If a developer can just jump right in without considering the whole timeline, they'll often do that. Why is this a problem? The first solution that comes to mind is seldom the best one. If your dev has been reading a trendy company's blog post about switching to a fancy

new database, don't be surprised that their immediate step is to shoehorn that into the requirements and immediately start playing with it.

This is a good first line of defense against shiny toys and time-sucks, but it's not the only one that comes from requiring estimates. Starting projects without an estimate is like shopping without a budget. This might be fine for routine trips to the grocery store, but you should have a price range in mind before you buy a car or a house. Even if you can't know exactly how long a project will take, you don't want to be surprised by an extra order of magnitude beyond what you had expected. It's better to get that surprise before the project starts, not after.

In fact, it's common for the estimate to vary from what you were expecting. And this is the perfect opportunity to understand why. Many times you'll have missed some critical complexity. Other times you'll realize your dev is out of their depth. The most common opportunity here is to get clear on the scope. If the estimate is much longer than you were expecting, it might be because the developer is unsure of the desired outcome. They may be putting too much effort into a part of the project that doesn't matter. Everyone wins when you focus on building the most important pieces first.

When you're responsible for your company's success, you're acutely aware of the constraints you are operating within. There's never enough time or money. Constraints help us focus and prioritize. By default, devs are insulated from these realities. Estimates overcome this.

When devs are on salary or charge by the hour, there's little direct incentive to ship. They get paid equally for every hour they spend on the project. This means that when they have the idea to switch frameworks, redo the architecture, or experiment with new build systems, they don't have competing financial pressure. They can move a project forward, backward, or sideways and be financially rewarded the same. On the other hand, you and your customers only benefit once the project is launched.

You do not want $100 solutions to $5 problems.

Scope creep doesn't just come from above. Over-engineering and "gold plating" are common dev behaviors. You want $5 solutions to $5 problems and $100 solutions to $100 problems. You do not want $100 solutions to $5 problems.

You'll encounter this over-delivery behavior when you need to account for some one-off special case. For example, you might have a single customer with grandfathered terms that no longer exist for

other users. Instead of just handling that account directly, a dev will want to build a whole system for handling arbitrary special terms with an admin interface and email alerts. This might be useful or worth the time if more than one of your thousands of customers has or is likely to have special terms—and a complete waste of time and money if not. You don't want to give your devs a blank check to build things like that.

Estimates help align these incentives.

Estimates are also valuable midway through the project. They give you a good timeline of when to check in on progress. When a project goes off the rails, the sooner you know about it, the better.

Without an estimate, you can only do periodic or milestone progress updates. With an estimate, you can check in at the 25%, 50%, 75%, and 90% marks. At these checkpoints, you should be testing the dev's confidence that everything is going according to plan. They should be growing more certain that they will hit the target ship date as time goes on. If they are less confident, that's a sign that you should dig in further or possibly intervene.

Pay particular attention to how the dev is acting at the 75% and 90% checkpoints. There's a reason why the last 10% of most projects takes 90% of the time.

This is where you'll find out that your dev has been avoiding all the uncomfortable unknowns, a product manager has been moving the goalposts, or that your dev has to drive their hamster to the psychiatrist an hour away each day and doesn't have as much time as they thought. The sooner you uncover these issues, the better—and you'll want to cut scope, add someone else to the project, and/or remove someone from the project ASAP.

To get the most out of these benefits, you'll want your dev's estimates to be at least somewhat aligned with reality. To be clear, you do not need estimates to be exact to be useful. Estimates are just a type of measurement. Measurements are observations that reduce uncertainty. We can't eliminate uncertainty, and there are diminishing returns to precision.

The best way to get good estimates is to make this clear to your devs. They are neither expected to provide estimates down to the millisecond nor are they expected to ship right on the dot. In my experience, you're doing well if most estimates are within 15%. You're going to have some projects be off by 50% or more. That's life, but don't let those become common.

I've seen managers try to "reduce developer stress" by making it clear that wrong estimates were not the fault of any particular dev, but instead are the

fault of the entire team. Managers like this will run planning poker sessions, take an average of the estimates, and when that estimate is invariably wrong, shrug it off and tell the dev they did a great job. This provides no incentive to any dev to improve their estimates or to try to hit the targets. Diffusing responsibility over the whole team is a bad idea.

When should you come down hard on a dev? Should they get a warning any time they miss a target ship date by more than 15%? No. It's difficult to predict the future, and we don't need perfect precision. We're looking for three things: consistency, communication, and conscientiousness.

If a particular developer consistently misses their estimates by 25% or more, that's a problem.

If any particular project misses its estimate, that's not a big deal. If a particular developer consistently misses their estimates by 25% or more, that's a problem. This means that this developer is not learning from their past projects and there's likely to be a deeper issue. The dev may be overconfident in their abilities or be highly susceptible to Parkinson's Law (the time required to perform a task tends to extend to all the time available to perform it).

There's an infinite number of ways for the universe to thwart our best-laid plans. You don't need to be upset with a dev if they miss an estimate. These aren't deadlines, after all. A dev should only worry if they fail to do one of two things: provide sufficient warning when they realize their estimate is wrong or put in the required amount of effort.

It's common for devs to avoid committing to estimates. Anything can happen. Unexpected issues come up all the time. This is true in all aspects of life, but it doesn't stop us from making promises. If you ask me to pick you up from the airport, it would be ridiculous for me to say, "I can't know for sure. I could get a flat tire or be sick that day. How about we see how it goes, and I let you know after it happens?"

So what happens in real life? What if I promise to get you, but I'm actually too sick to get you? Will you be upset with me because I'm breaking my commitment? That depends. Did I give you advance notice or were you waiting on me for hours? Was this sickness actually outside of my control or did I stay up all night drinking to earn myself a bad hangover? You're likely to be forgiving if I give you a heads-up with enough time to make alternative plans and if my behavior shows that I took the commitment seriously.

Devs don't need to be in trouble just because they don't deliver before the estimate. However, you should expect them to let you know as soon as things aren't looking good, and you should ensure that they aren't procrastinating, getting distracted by lower-priority tasks, or otherwise poorly managing their time.

The final thing you should beware of: Do not tell your dev what the estimate should be. If a dev does not feel like they made the estimate, they won't own it. They'll always have the escape hatch that they tried their best to do what you wanted, but it was never their idea. If you don't agree with an estimate or the timeline is unacceptable, you should either adjust the scope to eliminate any unimportant, time-consuming parts of the project or find a different developer.

If a dev does not feel like they made the estimate, they won't own it.

It's crucial that your devs take responsibility for their estimates and that you hold them accountable. Estimates will quickly become useless if devs have no skin in the game. If there's no reason to stick to an estimate, there's no defense against rewrites, over-engineering, or surprise disasters.

5

WE WILL NOT

Sprint

Sprints used to be controversial.

Now they are commonplace in engineering organizations—so much so that it's controversial to say that sprints are a bad idea.

Sprints are a bad idea.

I suppose if you had to decide between using two-week sprints or an annual release cadence, sprints are the winner. But that's not what you are deciding between.

In a nutshell, sprints are (usually) a two-week block of time to undertake a number of features, fixes, and other dev work. The work is planned before the sprint begins, and the results are evaluated afterward.

In theory, this fixed block of time forces the team to break work down into smaller, shippable parts. Any project or task that is too large to be completed within that time gets split up. Having short-term deadlines for shipping adds some urgency to overcome decision paralysis and mitigate scope creep. Likewise, any work that isn't a priority will be pushed into the future, when it won't be crowded out by higher-priority tasks. Additionally, having work planned out and visible for stakeholders and other interested parties is a bonus for transparency, and holding retrospective meetings to complete the learning feedback loop is great.

Those are all nice things. We can and should have all of them. However, sticking to an arbitrary two-week schedule is not necessary. I'm sure it was helpful in the past, but in today's world, it causes more harm than good.

A good metaphor for a sprint is a backpack. The backpack has a fixed size. You can fit a lot of small items in there, or you can fit a few large items. You can also do a mix. If someone wants you to carry a piano for them, it's easy for you to say, "Sorry, my backpack doesn't have room for that. Is there a particular part of the piano that would be helpful for me to carry?"

Of course, backpacks can also be frustrating depending on the size and shape of the objects. It's rare that you'll have the exact combination of items that will completely fill the backpack without going over. Often you'll have a bunch of space left over, or worse, it'll be packed so tightly it breaks open during your travels and dumps your possessions all over the ground.

Stakeholders only care about estimates for their particular features, not the sprint as a whole.

It's dumb to play this guessing game every other week. If you don't plan enough work for the sprint, devs will either be idle or break the spirit of the sprint by taking on future work. If you plan too much work, your devs will miss the deadline. There isn't even any benefit to guessing correctly. Stakeholders only care about estimates for their particular features, not the sprint as a whole.

Does it make sense to use the size of your backpack as an excuse for not being able to carry someone's piano? No. You don't need a two-week sprint to avoid taking on large projects that haven't been broken down into shippable pieces. Just make a rule that you don't take on large projects that haven't been broken down into shippable pieces.

You don't need a two-week sprint to avoid taking on large projects that haven't been broken down into shippable pieces.

Similarly, if you were relying on the fixed size of sprints to push non-priority work into the future, you can still do that. If someone wants you to work on something immediately, just show them your current list of projects and ask which one they want to replace.

If you want to have a retrospective, status, or planning meeting every other week, fantastic. Any project that has been planned or shipped since the last meeting is a great topic for discussion, but artificially constraining projects to this arbitrary cadence is unnecessary.

Here's the thing: You can break work down into small, manageable parts. You can push unimportant, non-urgent tasks into the future. You can be transparent about what your team is working on

and why. You can regularly sync up for a feedback loop on what's working and what isn't. I recommend doing all of these things. However, none require you to shoehorn all your projects into two-week periods.

6

WE WILL NOT
Allow Our Devs to Multitask

Until a project is shipped, its value is zero.

Having 10 almost-complete projects is a lot worse than having one shipped project.

We all know this—and yet we allow our devs to pile up more and more unfinished work.

Let's say you have five projects that need to get done, and it takes two days to complete each one. Would you rather have them completed all at once on the 10th day or would you prefer to have the first completed on the second day, the next on the fourth day, and so on? This should be a no-brainer. You want the first project out in the world as soon as possible. Until it's complete, it's not making money, saving you time, or making customers happy.

Here's a simple model: An engineer costs $1/day and you have five projects that will each generate $1/day when completed. It takes a day of work for the engineer to complete a project. If the engineer spreads their time evenly over the five projects, all will be complete at the end of day five. At the end of day five, you'll have made $0 from the projects and will have paid $5 to the engineer.

If, instead of doing all the projects in parallel, the engineer does them serially, the first project will be making money on day two, the second project on day three, the third on day four, and the fourth on day five. The engineer still costs $5, but that's $10 of income by the end of day five.

We've gone from losing $5 in the parallel example to making a profit of $5 in the serial version. That's a big swing.

In reality, most released features don't translate into immediate revenue. However, the sooner a feature is shipped, the sooner we get user feedback and analytics to know if it's achieving our goal. It could be direct revenue, or it could be preventing churn, removing friction, saving time, or just adding more visibility. There's truth to the saying that time is money. A feature shipped today is worth a lot more than the same feature shipped next year.

A feature shipped today is worth a lot more than the same feature shipped next year.

We like our devs to be busy. We do not want our devs sitting idle; we want them to be fully utilized. Unfortunately, this focus on being busy causes problems. While a developer is waiting on code review, they may as well start on the next project, right? Nope. They should be doing everything in their power to get their changes through code review, through QA, and into production.

We don't want developers hiding behind others to excuse why their projects aren't shipped. We want to be aligned. If a developer thinks that they are "done" as soon as they submit their code for review, they aren't going to be incentivized to make sure any ROI ever comes from their work. It's not their fault if the review is slow. It's not a big deal if QA

finds bugs; they can fix them later, after they finish what they're working on.

You don't want a developer's task to become someone else's problem. There's a tendency to let devs off the hook as soon as they "finish" writing the code. The truth is that code itself isn't valuable. It's only valuable when it's being used in production. That code needs to be reviewed, go through QA, and be deployed. If the dev thinks they're off the hook before any of that happens, you'll hear a lot of "not my problem." You want your dev to do absolutely everything in their power to make sure their code gets through review and QA as quickly as possible. Therefore, it must be their problem if it doesn't.

You want your dev to do absolutely everything in their power to make sure their code gets through review and QA as quickly as possible.

It's common for devs to mark their code changes as "needs review" and then move on to the next thing. If their changes stall here for some reason or another, they won't notice because they'll be distracted by the new ticket.

At this point, you may hear things like, "It's not my fault nobody reviewed it." This is exactly what you'd expect to hear from someone who is not invested

in the outcome. They did their job already; it's now someone else's problem. This is not what you want.

The situation gets even worse when, after review, their feature gets bounced back for failing QA. If the dev considers their only responsibility to write code that works on their own machine, they're not incentivized to do any serious testing to ensure that it passes QA on the first try. This additional loop adds a ton more time. To make matters even worse, if there's a long delay between when the developer last worked on the code and when they start on it again after failing QA, it's going to take them twice as long to get their bearings and fix the issue. Context switching is costly for everyone, but this goes double for engineers.

The more projects a dev has "in progress," the more these issues get exacerbated. When a dev is focused on a new project, they're not ensuring their last feature is being reviewed. If there are issues that come up in review, they're going to be less responsive in addressing them because they're busy with the next thing.

Devs may think all of this is unfair. They can write the code, but everything else is outside their control. This is a dangerous way of thinking. Life is uncertain for all of us. We can't predict the future or control all outside forces. What we can do is prepare for

different scenarios and adapt. It all comes down to motivation and responsibility.

Maybe you're not excited about meeting an acquaintance for dinner. If your car doesn't start, you can send an apologetic text message and spend the rest of the evening ordering in and playing video games. You said you were going to do something and you were thwarted by the universe; oh well.

What about a different scenario? This isn't just any acquaintance. This is a VC who is interested in investing, and there's a good chance you'll get a term sheet after this meeting. I'd bet that earlier that day you'd double-check that your car is ready to rock. I'd also wager that even if it didn't start, you'd pay for a rideshare, convince a friend to drive you, borrow a car, or take public transportation. In other words, with sufficient motivation, you'd prepare ahead of time and not let any inconvenience get in your way.

You don't want your devs ordering takeout and playing video games every time they encounter a minor obstacle. You need them committed to getting to the destination.

Do not reward your devs for being busy. Reward them for delivering finished products.

7

WE WILL NOT
Accept the First Solution a Dev Thinks Up

"Go with your gut," "trust your instincts," and "follow your intuition" have something in common: They're terrible strategies when you're making a decision.

Why?

There are too many ways for an engineer's instincts to get hijacked.

It's too easy for an engineer to think that they're choosing a technology or approach because it's the best for the situation when something else is going on.

Here's a typical scenario: You need to create a simple documentation site. You tell the engineer. The engineer has been reading about how great Hugo/Gatsby.js/Next.js/whatever is. The engineer says you should use Hugo/Gatsby.js/Next.js/whatever and you say, OK. Now instead of spending time on the actual documentation, your engineer is spending time getting Hugo/Gatsby.js/Next.js/whatever to work properly.

This simple documentation site could be a single HTML file. The v1 could be created and deployed within an hour. Instead, the engineer is fiddling with React server-side rendering, webpack nonsense, and a specialized build pipeline for days. Why does this happen?

The important thing to remember is that your engineers have different priorities than you do.

The important thing to remember is that your engineers have different priorities than you do. You want new features that help your customers and coworkers—as quickly and as bug-free as possible. Your engineers

want to solve puzzles in new and interesting ways. Sometimes these priorities align. Often, they don't.

Is the single HTML file the "best" approach? Not always. However, the HTML file solution is probably the simplest, and that was the stated goal up-front. This is a scenario where it's easy to spot misaligned priorities.

The answer depends entirely on the purpose of the site. Is this needed just to check a box for a potential client and it'll never actually be used by a human? Is this site going to be a major selling point that needs to be feature-rich and slick as hell? Or was this solution needed yesterday, and anything later than a week from now (no matter how amazing it is) is useless?

Devs who always want to do the most feature-rich, slick-as-hell version are just as problematic as the devs who always want to do the quickest, dirtiest, most hacky version. However, the problems don't actually start until they get attached to an approach and start fighting alternatives. So how do you prevent disaster?

I'll tell you what *not* to do. Do not tell your engineers how to solve problems. You are the "what"; they are the "how." If you take that away from them, they won't have any stake in the outcome. Any bugs or

issues will just be chalked up to doing it differently than they wanted to.

So how do you both have the engineers take ownership of the approach and avoid them choosing problematic approaches?

Don't accept the first solution they come up with. Make sure that they know that single approaches will not be accepted. They must come up with at least three alternatives to avoid digging in and becoming emotionally attached.

When a dev only presents a single approach, it becomes theirs. They'll interpret your rejection of the idea as a rejection of them personally. This is why you often see irrational arguments when a dev pushes back on an idea.

If we take the example above, you might say that the site needs to go up quickly and you don't want the team to have to learn a new framework. The engineer might respond with how popular Gatsby.js is, how much faster the site will load because of server-side rendering, or that it's "modern" or "best practice."

There are infinite ways to achieve any desired outcome. They certainly won't all be viable, but it's a fallacy to think that there's only one right answer

and it's always going to be the first thing your dev thinks up. The last thing you want is to have to make an "all-or-nothing" decision.

There are infinite ways to achieve any desired outcome.

If the goal is to get to the other side of a river, your engineer probably wants to build a bridge. If this is their only idea and you suggest that it might take too long, they'll dig in and say that Google and Apple build bridges, bridges are the safest way to cross a river, and users love bridges and constantly take photos of them. It won't matter if any of these things are relevant to your situation.

A different thing happens when your dev presents multiple approaches. When you tell them that a bridge will take too long, instead of fighting you like their ego depends on it, they'll present a different solution. They'll tell you that instead of building a bridge, you can just pay to rent a helicopter. Too expensive? How about a rope and simple raft?

Your constraints will invariably be some combination of high-level categories like time, quality, and price. Therefore, an easy way to cover your bases is to make sure that of the three approaches, one is inexpensive, another is quick, and only the final one is full-featured.

Why doesn't this happen naturally? It's uncomfortable to sit with open problems. Engineers like to build. The sooner they have a solution in mind, the sooner they can get to the "real" work.

However, if you want your devs to be creative, you need to train them to resist the urge to take the easy way out. They need to tolerate that uncomfortable feeling while they think through multiple facets of the problem.

There's also the fallacy that quality and quantity are at odds. A better way to think about this is that the truly great ideas are underneath the pile of mediocre ones. You need to give your engineers time to dig through multiple lackluster approaches before they can get to the good stuff.

Attachment is the enemy. The goal is to think deeply about the problem itself. So align your devs' incentives with yours and your company's.

WE WILL NOT
Allow Our Devs to Talk in Private

How much information do your key employees keep in their heads?

If your answer is "a boat-load," then I've got some very bad news for you: Anything not written down and shared becomes inaccessible when they leave your company.

Any conversations they've had, any decisions they've made, and any special knowledge about projects just disappear. To use a computing metaphor, a lot of your company's data is stored exclusively in RAM.

A truly valuable company can continue to operate and thrive independently from its employees. On the flip side, if a company would fail with the loss of a single key employee, it's hardly a business at all. That "company" can only survive until that person leaves. And no matter what, that person won't stay forever. There will be a new opportunity, a legal issue, a family emergency, or a health problem. Unless there is a radical breakthrough in life expectancy, this is guaranteed.

Will your company outlast you? Will it outlast your engineers? Is your company more than just the people who work for you? If all your engineers quit or disappeared in the Rapture, what would be left?

Hopefully, all your servers and databases would still be up and running. All your data would be safe on hard drives. All your code would be available in repos. Maybe you have a well-maintained wiki. Is this enough for a brand-new team to come in and operate? Would they be able to build off what's left?

If there was critical information that only existed in the minds of your team, success is unlikely. Onboarding is difficult enough for newbies when an existing team is in place, and it would be next to impossible if all the essential people and knowledge were gone. The new team would waste countless dev cycles reinventing the wheel or relearning old lessons. There would be clues in the codebase, but we could also expect the new team to relitigate old decisions because nobody would know why things were built in a particular way.

Now imagine this: What if all your team's conversations were recorded and searchable? What if new team members could just read the transcripts of a previous newcomer's onboarding process? What if an entirely new team could find and read all discussions related to a project's decisions? The future of your product—and your business—wouldn't be so dire.

What if an entirely new team could find and read all discussions related to a project's decisions?

In fact, even if your current team doesn't disappear in the Rapture, capturing relevant conversations has huge benefits. Anyone you onboard will have infinitely more context at their disposal to figure things out. Anyone who quits won't be taking institutional knowledge with them. Even within your

company, devs can move between projects much more easily and get up to speed faster.

It's a daunting task to force your teams to take detailed notes of every meeting (recording video and transcribing the dialogue would also work), upload them to a centralized place, and make them searchable. The good news is that you don't need to go to these lengths.

Instead of having voice and video calls, just default to having meetings and conversations in chat.

Instead of having voice and video calls, just default to having meetings and conversations in chat.

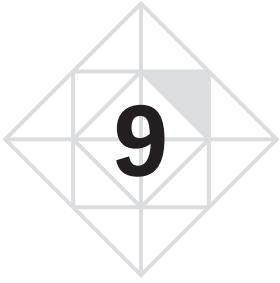
To anyone who enjoys meetings because they're fun and social, this might sound awful. Typing messages back and forth can make it harder to laugh and joke around. Others may not like it because when writing, you need to put more thought into how you communicate.

Some act as if it's impossible to inject levity and fun into written communication. The written word has made people laugh and connect emotionally for centuries. This power did not arrive with the telephone or Webex. As a bonus, you can still share funny links, images, and video clips.

I have less sympathy for people who prefer to talk on the phone because it's "easier" or "quicker." I agree that writing can be more difficult. But that's only because it takes more effort to structure your thoughts clearly and concisely.

It's a mistake to avoid communicating clearly just because it takes more effort. Writing forces you to think about what you are saying before you say it. You can't just wing it, and you can't easily hand-wave over incomplete thoughts. Even if it takes more effort, this is not a bad thing. The effort pays for itself.

When people work for a company, their effort should be directed toward building the value of the company. The company pays money to the employee, and the employee generates value for the company. If your devs are generating institutional knowledge that only exists inside their heads, the company is only getting a fraction of the value. Worse, that value can disappear at any time. If your company's value can disappear at any time, you don't have a durable company.



WE WILL NOT
Allow Our Devs to Wander Off

Even if you hire the most capable and motivated devs in the universe, it's naive to think they'll naturally stay that way without any intervention.

Why?

Without strong leadership, they're going to wander off.

They'll either start spending more time on their own projects, or they'll find another leader to follow.

People are not static. Motivation, enthusiasm, and engagement will naturally decrease over time. A new company or project will provide a boost, but this doesn't last forever. If you're not careful, productivity will slow. Even small changes will start to take forever. You'll see careless mistakes that affect users in production.

If a dev doesn't physically wander off to a different company, you still want to prevent their mind from wandering off. This causes them to work on things that don't matter or to introduce costly bugs.

Every dev needs a recurring one-on-one meeting with their manager to pull them back to the center.

How can you do this? Every dev needs a recurring one-on-one meeting with their manager to pull them back to the center. Devs wander off for specific reasons, and as long as you can consistently and directly address those reasons, you'll keep your devs and their attention much longer.

First, you need your dev to *want* to help you. It's difficult to follow someone you don't like—or who doesn't like or care about you. You might be able to fake it for money, but that's hard to keep up. The

same goes for your devs. If they feel that you don't care about them as people, don't count on them giving you 100% of their effort and attention. You don't need to be best friends or have them give a toast at your wedding. A little bit of care and attention goes a long way toward building their loyalty.

Second, keep in mind that even if a dev loves you and is committed to helping you and your company succeed, that's not enough. Not only do they need to *want* to help you, but they also need to know *how* to help you. Sadly, it's common for devs to have a distorted (or missing) view of what the company's priorities are. This makes it difficult (or impossible) for them to align their work with your goals.

Here's a thought experiment: If you were to ask all of your devs what their top priorities are, would you be able to guess their answers? If you were to tell all your devs to guess your top priorities, would they be correct? If you and your devs are not in sync about what is important, you won't be happy with their output and, ultimately, neither will they.

Sometimes they might be able to build or fix things that are helpful or will be helpful in the future. It's plausible that they might surprise you by building something valuable that you didn't know you needed. Regardless, it's reckless to rely on your devs to guess the best way to help your business. Make

sure that your devs are always clear on what your goals are and that their priorities align with them.

Third, you need to pay attention to when your devs are being blocked or slowed down. How can you tell when this is happening? Your devs will tell you.

Engineers prefer to get work done. They don't like to be impeded. If it's taking too long for their pull requests to get reviewed, they'll complain about it. If they can't figure out how to run your app locally, they'll complain about it. If product specs are too vague, they'll complain about it.

Developer complaints are evidence that something is wrong with the process. You don't need to make a change every time a dev complains, but it's your responsibility to figure out what the underlying problem is.

For example, if a new dev can't get your app running quickly, your docs might be out of date. Or you might need to update the app so that it's less picky about its running environment. Or new devs might need to take a crash course on Docker before they attempt to run the app locally. Or maybe this dev is just careless.

Your job is to find out. It's dangerous to always assume the dev is the problem and your process is

It's dangerous to always assume the dev is the problem and your process is perfect.

perfect. It's just as dangerous to always assume that your process is deficient and the dev is running into an issue that needs to be addressed.

Running engineering teams is like flying a plane. Don't ignore your gauges and instruments, and don't fly blind. You'll want to make adjustments based on the feedback you receive. If you see a warning light, investigate. Make sure it's not a false alarm and then do something about it. Ignore problems at your peril. Your devs will become less productive and less motivated—and, eventually, they'll look for another job.

The feedback you get from devs will often be more subtle than, "I can't get this to work." Instead, you might see a lot of confusion around why your codebase is set up in a particular way or why particular technology choices were made. This can either be genuine confusion or veiled disagreement. It can be limited to a single engineer, or it can be pervasive on the team. This is important for you to figure out. Keep in mind that if one engineer has an issue, others might be silently struggling as well.

Often, the struggle isn't process related. A dev's productivity and motivation can be tanked if they're

working on parts of the stack they don't like. Some devs just don't like working with markup and UI. A task here and there might be novel or interesting, but if that becomes the majority of their work, they'll lose interest. Even if a dev likes the part of the stack they're working with, if the work is too far outside their ability, it can be draining. It can be exciting initially, but over time, work that is too challenging will take its toll. Try to keep tasks aligned with a dev's interests and talents.

The fourth and final thing that causes devs to wander off is lack of meaning. This might sound abstract, but it's actually one of the simplest—and most commonly overlooked—things you can provide to an engineer. In your company, you think about your customers and clients a lot. Your company exists because it saves those customers time, eliminates pain, or helps them be more productive. It's probably easy for you to see how new features or fixes directly affect them. You know why your work matters. It's often very difficult for engineers to see that.

Your engineers won't automatically know that the last feature they shipped was a lifesaver. They won't know that their last fix made a customer's day. If a dev thinks nobody cares about their work, they'll stop caring too.

10

WE WILL NOT
Let Our Devs Boss Us Around

Are you afraid of your developers?

Devs are expensive, hard to find, and harder to replace.

Devs have a lot of control over your product and systems.

There are tons of features on the roadmap that need to get done, but your lead developer tells you that you have too much legacy code to continue working as is. The team won't be able to make any progress unless your app is rewritten with a better framework.

They're the lead dev and they know what they're doing, so you probably shouldn't annoy them with questions. Also, they seem to be pretty certain this is the way forward even though it's going to halt all progress for a while. Even if you could tell them no, they may not take it well and leave. You'd be completely screwed if that happened. It's best to bend over backward to make them happy, right?

Wrong.

Unfortunately, this is really common. In theory, the devs work for you, but because you don't understand the technical details, you feel you need to go along with anything they say, even when it will clearly have a negative impact on your business.

It's true that developers work on complicated problems that can require creativity. What's not true is that they should be given a free pass on making decisions that can dramatically affect your timelines. Your devs must be accountable, and they must understand the business objectives.

We all want to make beautiful software built from well-tested, well-documented, beautifully designed, elegant code. Sometimes, we even get to.

However, this is not what makes a successful business. Successful businesses provide value to their customers. Customers will never complain that the codebase is legacy, that your database is uncool, or that you're not using the newest framework. None of those things have any effect on your company's ability to help them make more money, save them time, or otherwise improve their lives.

Don't allow your devs to equate their wants with your customers' needs.

When your developers complain about these things, they're thinking about themselves and not the customer. They're thinking about saving their own time. They're thinking about technologies they want to put on their own resume. They're thinking about how fun it is to play with new tools.

Don't allow your devs to equate their wants with your customers' needs. Tech debt is real. Productivity gains from new tools are real. Increased user-facing bugs from lack of tests are real. Your job is to make sure that those benefits are real for your situation.

It might actually be the case that the best way forward is a total rewrite. However, that is extremely expensive from a business perspective. Time is the scarcest resource, and you should not halt progress on a whim. Do not allow a dev to bully you into making that decision. Make them prove that it's the best way forward—for the business.

Time is the scarcest resource, and you should not halt progress on a whim.

If they tell you that writing new features is slower with your current legacy approach, ask them how much slower. Does it make sense to have 10% velocity now to get a theoretical 110% velocity later? Is a total rewrite the only way to get an increase in velocity?

If they tell you that a rewrite would reduce the frequency of user-facing bugs, ask them why. They haven't worked with the as-of-yet-unwritten codebase, so what makes them so sure it would have a lower and not higher frequency of bugs? Is a total rewrite the only way to get a decrease in bug frequency?

Remember: Your devs are not irreplaceable. They don't need to be expensive. They don't need to be hard to find. You should not rewrite your app without exploring alternatives. Your devs work for you.

CREDITS

Design: Aespire

Editing: Bryn Mooth

Printing: TBD

Titling Typefaces: Adelle and Adelle Sans,
designed by José Scaglione and Veronika Burian

Body Typeface: Oso Serif,
designed by Robert Slimbach