Choosing a Tech Stack You Won't Regret

Secrets of focused technical leadership for fast-growing startups

BY ROB ZUBER, CIRCLECI CTO

A couple of years ago, I was at a holiday party. One of the hosts told me about another guest at that party who was starting a company, and mentioned that I should talk to him.

I love startups. I love anyone willing to go on that journey. So from the gate, I already want to do anything I can to help him out. When I eventually ran into him, we started chatting.

He started pitching me his idea and spoke for 30, maybe 45, minutes. Finally he paused to take his first breath. I told him I thought his business idea was really cool, which it was. Then I asked him, “How many users do you have?”

> “Oh, we’re still working on the business plan,” he told me.
>
> “All right, how long have you been working on the business plan?”
>
> “About 18 months.”
>
> “Do you have any software developers?”
>
> “Yeah, my friend studied CS and we’re working on this together.”
>
> “What’s he doing?”
>
> “He’s working on the business plan with me.”

I ended up talking to my fellow party guest for over 2 hours. I told him everything I’d learned working in software startups for the last 20+ years. My conversation with this startup founder made it clear to me that while startups are glorified throughout the business world, very little is understood about the day-to-day experience of being inside one, and especially what good technical leadership looks like from the driver’s seat.

> Be hungry for feedback, and be ready to change radically, quickly.

Startups are a wild ride, but they also put a lot of pressure on their leaders to make the right choices, find signal from noise, and move fast. It’s a stressful position to be in. Every decision feels crucial. But there are a few key strategies that will help you focus on what matters, and not waste runway on what doesn’t.

The most important thing I can tell you about how to succeed in a startup boils down to this: *be hungry for feedback, and be ready to change radically, quickly*. How can you do that? Lower what it costs you to make a change. The lower the cost, the more able you will be to seize opportunities, and the less likely you’ll get tied down by your previous bad decisions. The rest of this ebook comprises my best tips for what to do, and what to avoid in order to lower your cost of change.

What is the cost of change?

Grady Booch, one of the godfathers of modern software engineering, said that architecture represents the set of significant design decisions that shape the structure and behavior of a system, where significance is measured by cost of change[1].

Cost of change is one of the most important ideas that you can grapple with as a technology leader. The drivers of that change will evolve with the growth of your business, but in the early days when you're trying to figure out product-market fit, the scope of change is massive.

A podcasting system called Odeo ultimately became Twitter, a giant social network. Tiny Speck, a gaming engine, became Slack, a wildly popular messaging platform. These represent sweeping fundamental changes to business models. Making small and large changes is what enables ultimately finding product-market fit. So cementing the way that your system functions in a way that reduces your ability to change is going to be very expensive for you later on, when you find yourself needing to make big shifts. At the other end of the spectrum, if you're willing to throw out your code and start over, then the cost of change is low (more on this in the section about Erasable Code).

From this moment until you the time ultimately find product-market fit, you could end up making a 90 or 180 degree pivot in your business. For example, if you have locked the structure of your business model into disparate microservices, you've effectively greatly increased the cost of the kind of change that you're likely making at this early stage. It's wise to optimize instead for the ability to make big, sweeping fundamental shifts in how your business operates.

Your first priority: finding product-market fit

Getting feedback from customers is the most important thing you can be doing at the nascent stage of any business. Modern software tools make it possible to get your ideas in front of users on day 1 of your business, and that wasn't always the case.

Today, you are lucky enough to be building a business in the age of AWS, Heroku, and serverless — which means you have an advantage that I didn't have in 1998; in fact, it didn't exist for 99.99% of business history. Modern cloud deployment platforms give you the ability to get your ideas in the hands of users faster than ever before. With tools like this, you can focus entirely on building your product as users are interacting with it, so you can learn from those users as quickly as possible.

With this in mind, there are a few outmoded startup practices I want to caution you against.

I had my first job in software in 1998. Things were pretty different then. Our first server lived under my desk. When we raised a seed round, I flew to Herndon, Virginia to build out our data center myself. I flew in a plane with a suitcase full of disk drives. That was how you got software onto the internet in 1998.

Don’t do “stealth mode”

Stealth mode is the period in which startups build their product in secret, and won’t talk about it or show it to anyone for fear that someone else will steal their idea.

Before I explain why this is a bad strategy, let me first say that the fear around which stealth mode is constructed is generally unfounded. People are not sitting around, just waiting for a great idea they can steal and then build. If someone had the time and the energy to steal your idea, they probably already have their own idea, and they’re working on that.

But more importantly, for anything that you’re currently working on, there are at least three other teams somewhere, right now, also working on that idea.

The best thing you can do isn’t to work on it in secret, but to build it, and get it in front of customers as fast as possible. The notion that keeping it a secret will extend the time you have to build it is false — you’re merely increasing the chance someone will beat you to market.

Don’t build in secret; build in public. Don’t take your time making it perfect; build something imperfect and get feedback on it as quickly as possible.

What’s the opposite of stealth mode? Landing pages. Landing pages are the first, smallest test of a viable concept. I can put up 5 business ideas before breakfast and see which one pulls the most interest by lunch. I didn’t waste any time, and I got valuable signals about my market. I spent an hour, and wrote no lines of code, and it’s already a better idea than stealth mode.

I will say this many more times in this ebook, but **finding product-market fit is the only thing that matters for early startups**.

Building your product in secret is not finding product-market fit. If you’re hiding something, you’re not learning from it.

Don’t write a business plan, build an MVP

Before you say “Rob Zuber told me not to have a plan,” I think it’s great and important to have a plan — just be able to articulate the plan in a single sentence. Don’t plan further ahead than you have to, which means: plan your next step, build it, and get feedback. Then amend the plan. Repeat.

The absence of that large scale, upfront, 3-ring-binder type planning is often interpreted as lack of a plan. But really it means that I’m planning to be in an optimal position to seize an opportunity, and that I will evolve the actual plan along the way as I do the work.

The real reason to not write a business plan is that no matter how smart your business plan is, you’re wrong. You’re wrong about something in your business. Think back to those three other teams building your same idea. They’re wrong, too. But none of you understand what you’re wrong about yet. Your only strategy at this point is to be wrong as fast as humanly possible (ideally faster than those other teams) so that you can find out what right is and start building that.

This probably won’t surprise you, but this advice isn’t only for early stage startups. At every stage, keep up this cycle of building, executing, and getting feedback. It’s only in this feedback cycle that you’ll learn where you’re wrong and start to get more right.

Building the world’s most impressive business plan is not finding product-market fit.

Your only strategy is to be wrong as fast as humanly possible so that you can find out what right is and start building the right thing.

So put down your business plan. Find the simplest possible approximation of your idea that will give you any kind of signal. Build that in an afternoon and then go give it to someone and see if they care. Do whatever it takes to find that out — it shouldn’t take 18 months and a 90-page document.

Making the right decisions at the right time: cheap vs. expensive

As a technical leader of a startup that is designed to scale and grow, you will find that your job will quickly shift from writing code to setting the direction for the team. This shift will likely happen much faster than you anticipate, even as soon as you hire your first 2 or 3 engineers.

How do you make good decisions for your team, and build a technical architecture you won't regret?

We already talked about two core principles: the cost of change, and understanding that your first priority in a startup is finding product-market fit.

I want to add a third layer to thinking through decision making, and that layer is timing.

It's so accepted as to be boring to say that timing is everything in a startup. But just because you've heard it a million times before doesn't make it any less true. Your timing, your choice of what to focus on today and what to ignore are crucial decisions. These decisions are so important that they should feel uncomfortable. If you are prioritizing product-market fit ruthlessly to the exclusion of all other aims, you will create messes and issues that you will have to clean up later. You will choose shortcuts or hacks that will help you get product out to users quickly, launch new features or improvements fast, and get signal that you are on or off the right path.

Accept the hacks. Accept the mess. This is very hard for people to do. But, ruthlessly prioritizing the problems that are important today and ignoring the rest is the best way to get product-market fit, which is the only way to get to any level of business success.

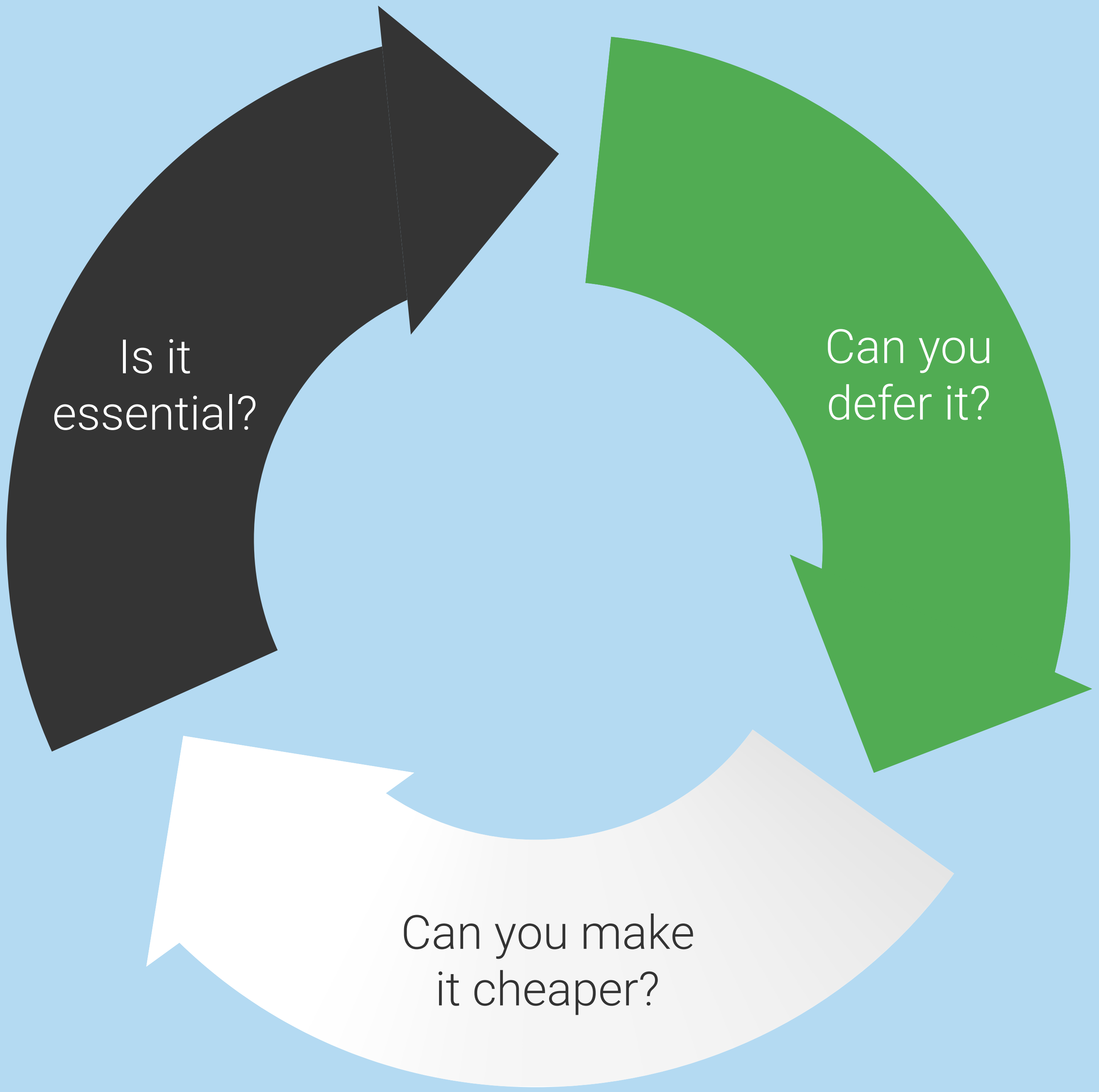
Accept the hacks.
Accept the mess.

If you succeed, you will have the money and resources to go back and optimize your stack.

If you don't succeed, it never mattered anyway.

I want to acknowledge that this is very hard for engineers. You are likely hiring extremely smart, detail-focused individuals who take deep pride in their work.

Your job is to reinforce, loudly and often, that an engineer's best work is not the work that is the coolest, most elegant, or most technically advanced. The best work of engineering is the solution that is the best choice for the business.


Is it essential?
Can you defer it?
Can you make it cheaper?

Expensive now, cheap later: back away from shiny tech

Defer, defer, defer

I hope by now I've convinced you that certain decisions (like how long you should stay in stealth mode for) shouldn't be made at all. But as a technology leader of a hopefully-fast-growing startup, there are other complex, interesting, and challenging decisions that you probably see in your future, and that perhaps you're eager to make correctly. Should you use microservices? What's the right way to orchestrate Kubernetes clusters? I am here to tell you to put off these decisions as long as you can. Deferring big decisions is not just a way of avoiding difficult tasks, but it can (and I'll argue, should) be a core strategy. Don't make any decision you don't have to, and try and put off any decision you do have to make for as long as possible.

Whether you like it or not, the name of the game for an early startup technology leader is ambiguity. You don't know what business you're in yet, whether it will succeed or fail, whether or how you'll have to scale, or answers to a host of other existential questions.

> Don't make any decision you don't have to, and try and put off any decision you do have to make for as long as possible.

An ambiguous position demands that you:

1. Avoid any decision that you could also make later.
2. Use as much existing material as possible.

Let's dig into what that looks like in practice.

Let time do the work

In the early days, the CircleCI application was a monolith that took a customer's build with its associated data and pushed it into one of several LXC containers. Every container that we spun up was instantiated from the same image that contained everything that we thought anyone would ever want in their test environment. In hindsight, this sounds like a terrible idea, but at the time, it was fantastic. It was simple to maintain and supported the needs of our early customers, many of whom were building Rails monoliths.

As time passed and our customer base grew, so did the diversity of their needs in terms of test environments. Upgraded versions of their underlying databases, novel new development frameworks, even new operating system versions became necessary.

The original container management in CircleCI wasn't designed in a way that allowed us to adapt easily to these changing needs. But it was fairly simple. So when we set out to solve these problems, we knew where to splice in a new approach and it took minimal work to enable that splicing. We also didn't have to unravel a poor generalization that didn't support our new problem.

Our original approach sounds pretty simple, and it was. It also saw us through five years of successful growth as a company. We met the needs of our customers on that simple system for five years before the first of them ever tested its replacement. In those five years, Docker was created, as was HashiCorp's Nomad. Combined, those tools eliminated huge portions of the work necessary to get the flexible and scalable environments that we support for customers today.

As we retooled the system to adapt to the changed market we were in a position to ask "How do we do this in a way that better positions us for incremental change?" It would be difficult to overstate how much value that five years of experience provided when designing a solution.

Most of the time we don't even know that we're making a monumental design decision at the time we make it because the alternate path hasn't shown up yet. At the same time, correct solutions have a habit of revealing themselves if you can find a way to wait long enough. Technology gains traction or dies. If you chose a container orchestration engine in 2016, there's a roughly 20% chance that you would have chosen Kubernetes. That means by the beginning of 2018, almost 80% of companies were switching. Those are not great odds.

So deferring can be good. Deferring to the point of creating a crisis is not.

Beware the esoteric tech stack

I'm going to crush someone's dreams right now who just decided to invent their own programming language as a precursor to building their e-commerce site. But **nobody in the history of startups has won because they out tech-stacked their competition**. There is no tech stack that will give you a leg up because it's new and different from what everybody else is using. The only thing that will give you a leg up is something that everybody already knows how to use.

Dan McKinley has a great blog post about **choosing boring technology**. He writes: "If the choices of software were truly without baggage, you could indeed pick a whole mess of locally-the-best tools for your assortment of problems. But of course, the baggage exists. We call the baggage 'operations' and to a lesser extent 'cognitive overhead.' The problem with 'best tool for the job' thinking is that it takes a myopic view of the words 'best' and 'job.' Your job is keeping the company in business. And the 'best' tool is the one that occupies the 'least worst' position for as many of your problems as possible."

CircleCI's founders chose Clojure as our programming language. I love Clojure. It's a beautiful language for a team of two developers. Nine years later, however, with a team of 130 engineers, and hiring more every day, I am forced to make the choice between a tiny pool of candidates who understand Clojure and are lacking entire other areas of engineering discipline or tripling my onboarding time. Because with every single engineer that I hire, I first have to teach them a programming language that they know nothing about.

If you are trying to build a business, a cost-effective delivery business, then teaching new hires obscure programming languages, or managing the operational overhead that accompanies shiny new technologies is not where you want to be. This is not finding product-market fit.

Put another way, stack-related decisions are technical and they don't matter to your users. Product does. Build great products to put in users' hands.

What about microservices?

Let's look at product-market fit through the lens of microservices: when you build out a microservices architecture, you're taking your code and splitting it across API and network boundaries, which creates much more concrete differentiation. It also greatly increases the cost of then making a change when you realize several parts of the system need to be swapped out.

So don't use microservices yet.

If you're trying to figure out what your business is, the last thing you need is microservices. In fact, trying to build them right now might kill your business. Your job as a technology leader in your organization is to maximize the ability of technology to impact value delivery to your customer. And one of your key points of leverage as a technology leader is the architectural decisions that you make, or don't make. At this point, that means keeping them simple, avoiding anything that might add needless complication to your ability to ship quickly and get feedback.

Product-market fit will paper over bad technical decisions every single time

Don't worry about whether you're choosing the right technology for your business long-term. Product-market fit is what matters right now. Once you find product-market fit, the drivers of change will be different, and at that point you can reconsider some of your earlier decisions.

If you need more encouragement, remember that product-market fit will paper over bad technical decisions every single time. If you have product-market fit, you will be dragged faster than you could humanly believe towards success. And if you have great technical decisions and no product-market fit, it's completely irrelevant.

Pragmatism always wins the day – build things that are simple, build things that are boring and build things that are battle-tested.

Depend on uncertainty

At any point, something crucial about your business could change. But you don't know what or where yet.

In "97 Things Every Software Architect Should Know" Kevlin Henney describes an important approach to thinking about uncertainty:

> "The presence of two options is an indicator that you need to consider uncertainty in the design. Use the uncertainty as a driver to determine where you can defer commitment to details and where you can partition and abstract to reduce the significance of design decisions. If you hardwire the first thing that comes to mind, you're more likely to be stuck with it – incidental decisions become significant and the softness of the software hardens."

This framing is great for explicit decisions, but what if you don't know you're making a choice?

Many times, the choice isn't even visible yet, but will reveal itself later. In these situations, the answer is not to overgeneralize, building abstractions everywhere "just in case." Implement abstractions in only the places you have identified that have multiple implementations, i.e. items with high cost of change, where you also have high confidence that there will be a change. Building perfect abstractions all over the place will certainly protect you against inevitable change, but will also be impractically costly. Instead, keep things as simple as possible so you can understand them later if you have to make a change. By merely separating concerns and keeping like with like, you still have work to do when it comes to making changes, but the cost of doing the work is low.

Build in a way that minimizes the cost of being wrong, and then watch change closely.

This often looks like leveraging every resource available to you that helps you avoid building things yourself. When you can't use something off the shelf, invest in the highest level of abstraction possible and continue looking for ways to leverage existing tools and frameworks. For example, when I'm still trying to find my business, if I can use AWS Lambda with 10 lines of original code, then I'm going to use that even though it's more expensive than doing it myself. This frees me to only improve the parts that are failing, which I can wait to see as I go along. When the time comes to build something custom because what you're trying to do just can't be supported by something in the market, then you'll know. And you can focus on solving that problem. As technologists we often feel we are not doing our job if we are merely connecting other peoples' tools and libraries together. But more accurately, your job is to make decisions that drive value for the business, even when that means deciding against building exciting new tools.

Remember: code is dead. Once it's written, it becomes a liability. There are no perfect designs at this stage. The less code you have, the better, so keep it small and simple, and be comfortable throwing it out.

Erasable code

Your architecture is going to evolve and it's going to be messy. Spoken as someone who's been a shining embodiment of the "oh my goodness, how do I get my site back up?" model of software design, there's not a lot of whiteboarding involved. It's closer to, "what can we do as of yesterday to get this thing fixed?" The truth of software evolution is it's going to look pretty dicey when you look back on it a little bit later.

But there are some things you can do upfront to get yourself in a better position once you're ready to scale. And there are plenty of things that will slow you down from getting to that place at all. One of my favorite writings on this topic is a blog post from the blog Programming is Terrible, about **writing code that is easy to delete**, not easy to extend:

> "If we see 'lines of code' [not as 'lines produced' but] as 'lines spent', then when we delete lines of code, we are lowering the cost of maintenance. Instead of building reusable software, we should try to build disposable software." -tef

What this post points out is that for much less effort than it takes to make something exceptional, you could focus on making it simple enough that you could throw it away and replace it down the road. This type of pragmatism now will be enormously helpful in preparing you to scale successfully down the road.

There is often a very this-or-that perspective in engineering: you either have a monolith or you have microservices. You have really well-crafted, abstracted, encapsulated code or you have spaghetti. But the truth is you're going to end up somewhere in the middle. Getting comfortable with that, being pragmatic about it, is one of the levers you can use to drive your team to focus on things that matter and solve the most critical problems at any given time.

Cheap now, expensive later: instead, start here

One framework I find useful is thinking about decisions that are cheap to make upfront but become expensive to change later. If you're going to make investments (and you are), it's worth considering the cost of change upfront.

We've already decided you're not going to be in microservices (remember: when you find yourself defining Docker and Kubernetes instead of building your business, you're making bad decisions). Instead, I like to think about isolation. We talk a lot about encapsulation in software and the rule of three. We can ponder what the perfect abstraction is forever. In keeping things simple and pragmatic, though, isolate things that are specifically tied to one area of your business and make sure all of that stuff is together. Later, when you understand it well enough to have to make decisions about encapsulating it or abstracting over it, at least it's all contained.

Then you can define boundaries.

Make sure you define those boundaries within a simple code base. At this point, you're probably building a monolith, which is fine. Great, even. Monoliths get a bad rap, but in reality, everything is in one place, easy to find, and easy to pull out when you need to.

When we think about this in terms of cheap vs expensive, it's actually cheap to stick to those boundaries, understand your boundaries, be clean with them and adjust them as you're evolving. Because the cost of reshaping a function name, or its position in a code base, is extremely low relative to the cost of moving things between services. Look at things like ports and adapters or hexagonal architecture: there are very well-established patterns for defining well-isolated, well-bounded code within a single code base. Then, when you scale to the point where you realize you need to pull a piece out, you know exactly where that piece is.

In contrast, the microservices transition that a lot of people go through is wanting to pull out a piece and realizing they need to clean up their model into something that's not a dumpster fire first. That's where the cost is really high.

Domain-driven design

Domain-driven design is another process that is cheap early, and expensive later. If you happen to be integrating with other platforms, something that many businesses being built these days do, you'll find yourself pulling in data from those other platforms. And you'll translate that data, at a boundary, into the state that you want in your own system.

Here's a personal example: at CircleCI, we've been working with GitHub since 2011. We still have data in our system that looks like data that was given to us by GitHub in 2011. And we have huge amounts of code that deal with that in multiple places. Being able to translate that at a boundary would make things much, much easier.

Knowing what you own and making sure that you have control over it is very simple to do early and very expensive to add later.

Invest early in CI/CD

If you've ever built a CI and CD pipeline for three lines of code for your `hello world`, you know that it's fairly easy. Automating your entire infrastructure when you have a massive code base and something very big that you're trying to get out is much, much harder. Setting up CI/CD pipelines for your software is perhaps the most impactful decision you can make early on that will make your life and your team's life much easier as you hit your stride and start to scale. CircleCI is a perfect choice for startups because it's fast and easy to implement as you're building out your application. It runs automatically without manual overhead or devoting precious developer time to managing it — CircleCI is the CI/CD expert on your team. And CircleCI scales as your team grows and your codebase changes, helping make those scaling and architectural transitions seamless.

The ability to move quickly and adapt to change is going to be tied very tightly to your ability to deliver. CircleCI is designed to help teams deliver — staying nimble and staying ahead of their competition.

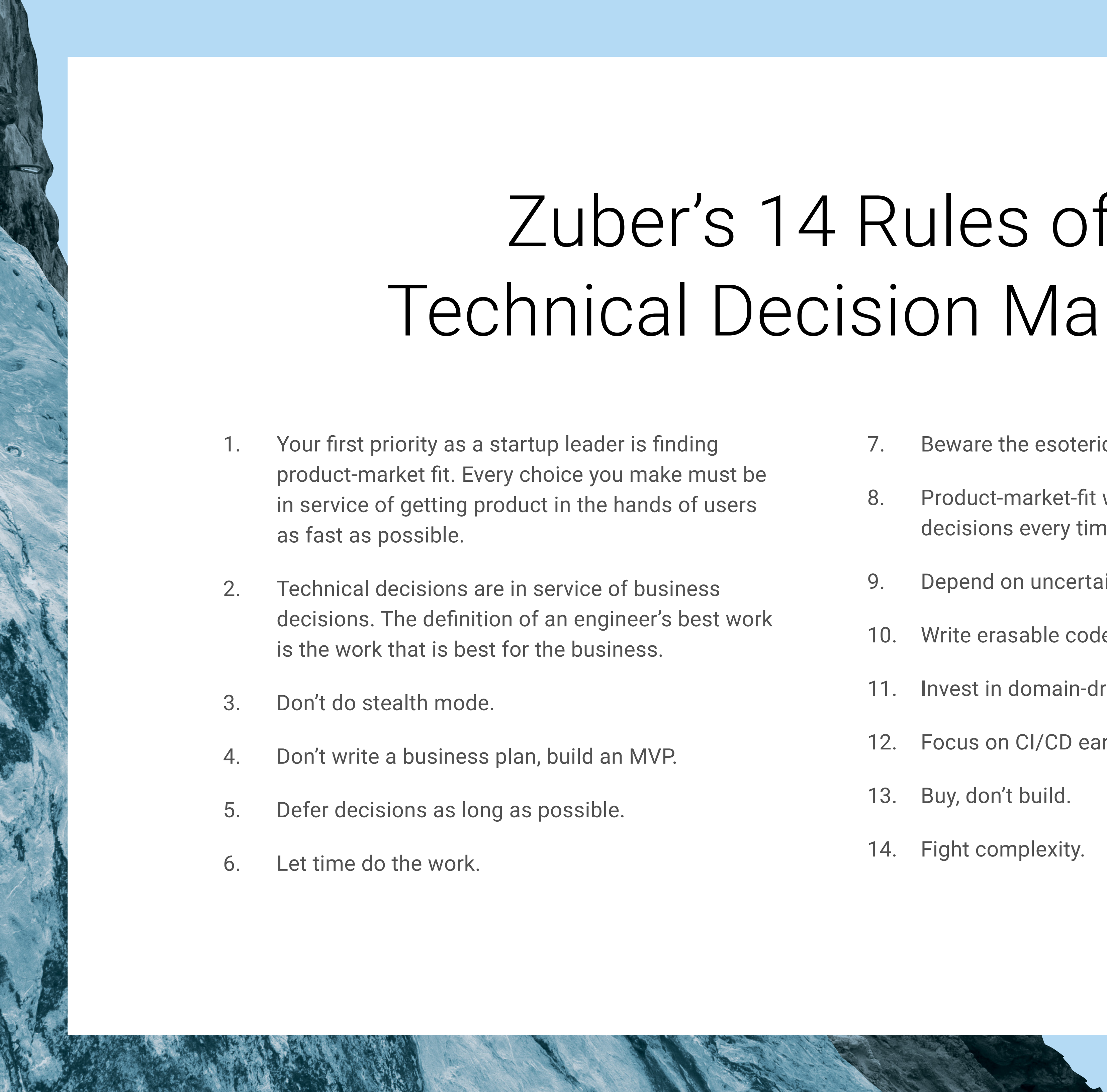
Time, money, distraction: Some decisions are always expensive

Buy, don't build

Would you build your own email server? I've done it. But I worked at an email company. For the same reason, don't build your own CI/CD. We do it (and we do it well) so you don't have to do it at all.

We all continue to stand on the shoulders of giants. In every cycle of tech, we are building interesting combinations of what had been novel technology the last time around. First, we built chips. Then we wrote the first software: adding machines. We built on top of those. We built maps. Then someone built turn-by-turn directions on top of mapping. Then when ride sharing came onto the scene, they didn't have to invent turn-by-turn, it already existed. If you find that you are reinventing something that's already been invented, know that your competition is not. They are focusing on the business you're trying to win — you're essentially wasting your time and you're going to lose.

If it's not a competitive differentiator or delivering new, unique value to users, do not build it. (And if your company isn't in the CI/CD business, build on top of the best one there is).



Zuber's 14 Rules of Technical Decision Making

1. Your first priority as a startup leader is finding product-market fit. Every choice you make must be in service of getting product in the hands of users as fast as possible.
2. Technical decisions are in service of business decisions. The definition of an engineer's best work is the work that is best for the business.
3. Don't do stealth mode.
4. Don't write a business plan, build an MVP.
5. Defer decisions as long as possible.
6. Let time do the work.
7. Beware the esoteric tech stack.
8. Product-market-fit will paper over bad technical decisions every time.
9. Depend on uncertainty.
10. Write erasable code.
11. Invest in domain-driven design early.
12. Focus on CI/CD early.
13. Buy, don't build.
14. Fight complexity.

Conclusion: Fighting complexity while scaling

You probably have a deep-seated notion of how you want your technology to work. You know your priorities. But as you build out a team, unless you are very clear and articulate about those things (and say them about 50 times more than you think you have to), everyone you bring onto the team will come with their own perspective. One thing I hear from other CTOs is that after they started their company, they end up hiring more experienced engineers. Suddenly they feel like they're not the most experienced person in the room and doubt their position to influence what those people should be doing. At some point you end up with a bunch of leaders at the table, making divergent decisions about how things are going to function. And the cost of that divergence is extremely high.

Be very clear upfront; as the founder or technical leader, that is your job. You have the context of the business. You know the value you're trying to deliver. Focus on that. Express it in a way that people can look to and know whether they're making good decisions day-to-day or not is critical.

Fight technical complexity as you scale. Always keep driving toward product-market fit and focus on the decisions that will enable you to pivot as quickly, as often, and as cost-effectively as necessary. If you maintain focus on your highest leverage decisions as a technical leader, you will be in the privileged position of having the time and money to later untangle your speedy but strategic early decisions. With strategy, relentless focus, and a little bit of luck, you'll get there.