Startup Founders' Guide to Software Delivery

Laying the groundwork for velocity and quality at every stage of growth

BY ROB ZUBER, CIRCLECI CTO

circleci

Software delivery on a team of 2 people is vastly different from software delivery on a team of 200. Over the growth of a startup, processes and tool choices will evolve naturally — but either optimizing too early or letting them evolve without a picture of where you're headed can cost you in time and agility later. That's why I've written this guide on how to evolve your delivery process with purpose.

The optimal approach to software delivery is tied to your software architecture, which, as we know from Conway's Law, is in turn related to your organizational structure. Throughout this ebook, I'll be offering insights on how each of these factors plays a role in setting you up for either turbo-charged growth or mounting roadblocks as you scale, all depending on the decisions you make at key inflection points.

Full disclosure: I'm a huge proponent of CI/CD, and also the CTO of a CI/CD company, so don't be surprised when I encourage you to use CI and CD throughout this guide. Among many other strategies I'll lay out, I will explain why it's never too early to start using CI/CD, though the areas it can impact will change and evolve over the course of your startup's growth.

Why should you listen to me?

As a 20 year software industry veteran, four-time startup founder, and three-time CTO, I know that time is one of the most valuable resources a startup has on its side. Make the wrong choices early on, and you'll lose precious runway trying to fix what could have been avoidable problems. I've seen how the right choices can pay huge dividends, as well as where to be hyper-vigilant against over-architecting to account for every eventuality. For the past 7 years I've led the engineering team at CircleCI, and from the tens of thousands of organizations who use our platform to streamline and expedite their software delivery, I've seen without a doubt what works, and what doesn't: for companies at every stage, and in every industry.

Founding stage

1–10 engineers

This is my favorite stage of company development. You're excited. You've got an idea. And the last thing you want to work on is tooling. At a certain point, you're deploying your 20th change to your initial environment by hand. You might be thinking "this can't be the right way to do this." And you'd be right. Even though you technically can operate like this, this is the time to put CI and CD in place. At this early point in a company, you might not even be in the right business (I've seen, and led, companies who made huge early pivots before they landed product-market fit). But rapid delivery and confidence in your releases will be instrumental in helping you get there.

This exploratory phase demands simplicity. Coordination costs are low at this size so use that to your advantage: a single team, a monolithic codebase, a basic (automated) deploy.

You don't want to stop and ask your cofounder: "Hey, remind me the Capistrano command?" or "Did I just push on top of your push?"

You don't want your laptop to be the "build laptop" where when someone wants to push code, they have to come to you.

With the click of a button, you could have CI/CD. Don't make it harder for yourself. CircleCI has a free plan. Use it. Just this one single move, even if you don't "need" it right now (and I'm going to try to convince you that you do), will set you up to be leagues ahead later down the road. When things get real and you've got a security problem that's blindsided you, or you actually have customers and are tackling issues of scale, you can just ship a fix.

And that is way better than "Rob's at lunch, we can't deploy."

What to do

Put CI/CD in place *now*.

Your application is changing a lot and you want to learn as fast as you can. Don't burn time trying to remember how to safely deploy. When your systems do start to necessitate more complex (and I will explain later why you should defer complexity as long as you can) you will have the practices and tooling in place to handle complex systems.

Keep it simple.

It's so easy to have CI and CD in place. Get Google Analytics and set up CircleCI. At this stage, that could be all you need. But not having them is a huge problem. By not having these basic tools, you give your competitors a massive edge. Building a company is all about execution; acquire the tools you need to execute reliably, and put off the rest.

BONUS FOUNDER TIP

Keep the architecture and tooling as simple as possible.

Complexity is the killer: it kills speed, culture, and product velocity.

Shomik Ghosh, Principal, Boldstart VC

> *"A good build tool helps the team automate processes, practice good hygiene, etc. While a tool like CircleCI does scale, startups at this stage should be focused on using the best tools that they know that can integrate easily in developer workflows to make life easier. Shipping and hiring velocity are the best predictors of early success and so using CircleCI helps with this because it is one of the most widely used CI/CD platforms for dev tools, and people understand it and love using it. And it helps ship faster.*
>
> *Automate core building blocks as much as possible. Rolling your own CI/CD tool is undifferentiated for the goal of your startup, which is the business logic that will be built on top.*
> *Use as much of an automated process as possible for things like that. Scaling later will mean needing to have a source of truth for everything to help with coordination across teams. CircleCI can help there, too. And then as you become a later stage company, you need something that integrates easily with all different tooling and removes friction for onboarding new devs, so any product that helps that is useful (again CircleCI)."*

- **Think, don't act.**
  Knowing what your future constraints are or might be doesn't mean you have to build to them. Building is expensive, thinking is not. Think through your scaling roadmap before you make decisions so you can make conscious tradeoffs about what you're deferring to the future.

- **Create placeholder implementations.**
  This basically means to defer complexity, but jumpstart processes that support complex parts of an engineering org early. Traceability is a great example of a process that's relatively easy to set up in a monolith, but much harder to put in once your application has scaled to include services. Why bother? Having traceability drives certain behaviors and decisions. Because there is even a simple implementation, you'll think about traceability in everything you build. Putting these practices in place early will actually change the way you code. You will end up with fewer asynchronous handoffs in your codebase, or you'll design them in a way that you'll be more likely to understand later.

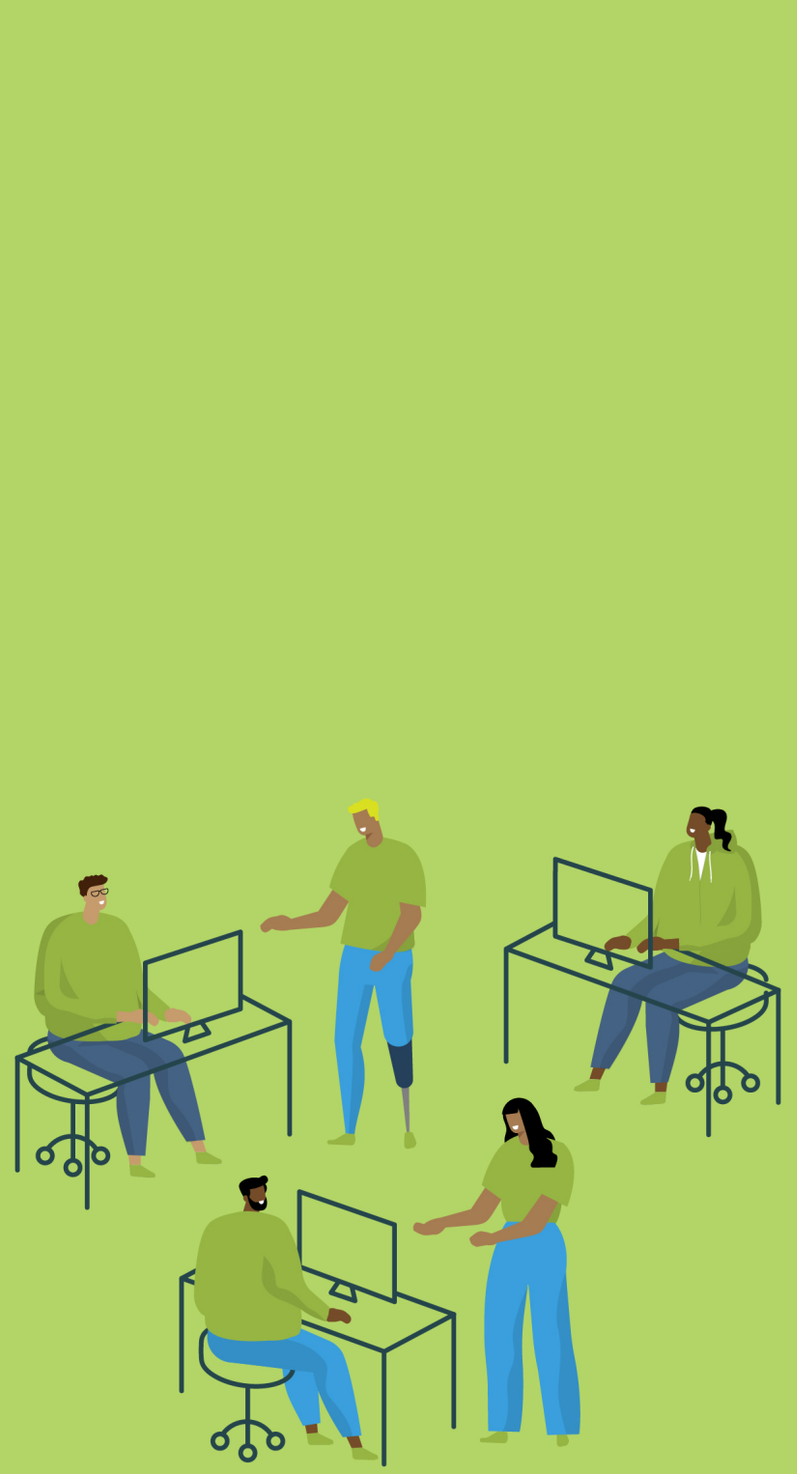
- **Prioritize operability from the beginning.**
  The first developer in a startup should already take responsibility for thinking about the operation of the software. It should be apparent in the first line of code. For example, the effort to select a logging library that can redirect output to a central system is negligible. While you're at it, structure your logging and either remove or structure the log points you were using to debug as you built.

What not to do

- **Copy the big players.**
  You are not solving the same problems as Netflix, Google, or Facebook. The advancements they are focused on are unlikely to even translate into your context, let alone be useful as a direct copy. Keep it as simple and flexible as possible. Hopefully you'll get there someday, but today is not that day.

Very early stage

- 10–20 engineers
- Finding product-market fit



At this point in your company, you probably don't have a person dedicated to developer efficiency or tooling. You might, but it's not common. It's more likely everyone is contributing to the cause. And your team will benefit from deferring creating a dedicated Dev Experience or Tooling team.

Why? Because at this stage you will be building this mindset into your culture precisely by deferring the dedicated team. You get further by having everyone be responsible than by relying on a single person. Then people are creating tools, processes, and code that can be shared across teams.

At 10–20 engineers, you have some practices in place, you're keeping your software simple. You're finding product-market fit. That's the benchmark. Once you get there, you can invest.

Before product-market fit, everything you build has a high probability of being thrown out — the goal is to build as simply as possible so you can pivot or throw things out easily. Once you have fit, you can start making investments.

This is the awkward stage where you are trying to manage your ability to work independently without creating unnecessary complexity in your operating environment. Should you break into teams? There are too many people for one team, but your priorities are still shifting. But it does feel cumbersome to be working on the same thing.

You'll be tempted to build services, but it's too early. You'd be adding unneeded deployment complexity that you'll pay for later.

Rather, create boundaries and stability with components and libraries. Whether you end up using libraries or just well-defined boundaries between your monolith (i.e. a modular monolith) — you can still pull it all together and deploy in a monolithic fashion This will give you room to operate with more independence, while keeping your deployment simple.

Breaking your codebase into services will add a whole new layer of decisions: What kind of inter-service communication should you use? How should you do service discovery? Or retries? There is a massive amount of cost and complexity that comes with your first service.

So put it off.

You will want to change your continuous integration (CI) process without changing your continuous delivery (CD), or in other words: change your build model without changing your deploy model.

What to do

Keep deployment simple.

Keep your build and deploy as simple as you can. Resist the urge to create services or overly complicate anything you can defer instead.

Maintain efficiency and productivity.

This is crucial as you grow past the 2-pizza mark. Look for more fluid ways of creating independent work streams without concrete team definitions.

BONUS FOUNDER TIP

Have comprehensive and thoughtful test coverage from day one.

Pay down tech debt from day one.

Andy Vitus, Partner, Scale Venture Partners

What not to do

Microservices.

Building services where you could use components or libraries instead will create deployment overhead that a team of this size shouldn't have to deal with. Defer.

BONUS FOUNDER TIP

Let gross margin deteriorate and plan to 'fix it later'.

Add unnecessary complexity.

Andy Vitus, Partner, Scale Venture Partners

Early stage

20–50 engineers

At this point in your growth, you likely have product teams. You are starting to find ways to break up the product so teams can operate independently. When you've reached this stage, you'll see a return on investment from sharing components and practices across your tooling. Being able to share patterns across teams doesn't just save up front implementation time, it will reduce the downstream maintenance burden. (This is why we built CircleCI orbs — because sharing configuration across teams is such a huge value-add at this stage of company and up)

Right about now, you may be considering breaking up your team to accommodate different engineers' language preference. You may be wondering what the harm is in letting each developer work in the language they think is best.

Don't do it.

Yes, it may seem like you'll move fast today, but you'll surely be slowed down tomorrow. The definition of speed gets so easily connected by engineers to volume of code written. We may feel we're moving fast ("I know Rust! I'm writing a lot of code!") But if you or your engineers are writing boilerplate, then it's a sign you should have used a library.

Delivering value must, at this point, be the foremost priority, not lines of code written.

The 20–50 engineer stage is the point at which you'll start to fracture. Invest in countering the ability for the team to fracture. There's an expression in DevOps known as "paved roads." This means that sure, you could go up over the mountain by foot and chop down trees, or you could drive over the freeway.The tooling you provide (including CI/CD tooling) should make it easy for your engineers to choose the well-paved paths. If the folks in your org aren't, you probably aren't setting the right goals.

[Paved Roads]

Offer pre-provisioned tools and shared components to make it easy for your engineers to do the right thing.

Put another way, me trying to use Rust because I saw a cool YouTube video should be painful for me.

How do you pave a road? Build shared components. If it's language stacks, just work in a monolith. At CircleCI, we have a pre-canned Docker container that's already tested. If it's a requirement that you are up to date on CVEs, we have that built into a Clojure pipeline. We have backend shared pieces that folks can use to build services in Clojure. All the things everyone does at the start of a project, we have them and you can just get what you're working on into production. If there are security updates, you get them for free. But: if you want to develop in Go, your team is building all that from scratch.

What you signed up for is to deliver value to your customers. No matter how much code you wrote and in what language, this is still the bottom line. Make it easy for your team to make it happen.

What to do

Standards. And linters.

Coding standards are not interesting. And debating approaches to writing code (everything from debating indentations to parentheses) is one more thing that people can spin their wheels on that adds no value. Choose a standard, enforce it. Move on to delivering value.

Automate your culture.

Everything you can say as a statement about your software, put it into CI/CD. Your decisions about how you do things are now embedded, and automatically enforced. That's why you want to start with CI/CD right out of the gate — it gives you a home to put your rules.

Create placeholders where necessary.

As an example, there are free vulnerability scanners; just use them. Build the culture of coding standards and practices. Add them all in your CI/CD. Later, when you upgrade the services you're using, you have a place to plug in those paid services. It's easy to do at this stage, but hard to add in later.

BONUS FOUNDER TIP

Expose engineering to customers' feedback.

Eric Anderson, Principal, Scale Venture Partners

> *"You want an engineering team that understands why customers want a given feature and what it is solving. There is a time and place for large product management teams that curate information and focus the team around core principles and maintain consistency across a product portfolio, but this isn't now [early in GTM]. As you hone product-market fit, you can best take advantage of your small team by optimizing for direct channels of communication between customers and engineering and maintaining tight feedback loops."*

What not to do

✖ **A complete rewrite.**

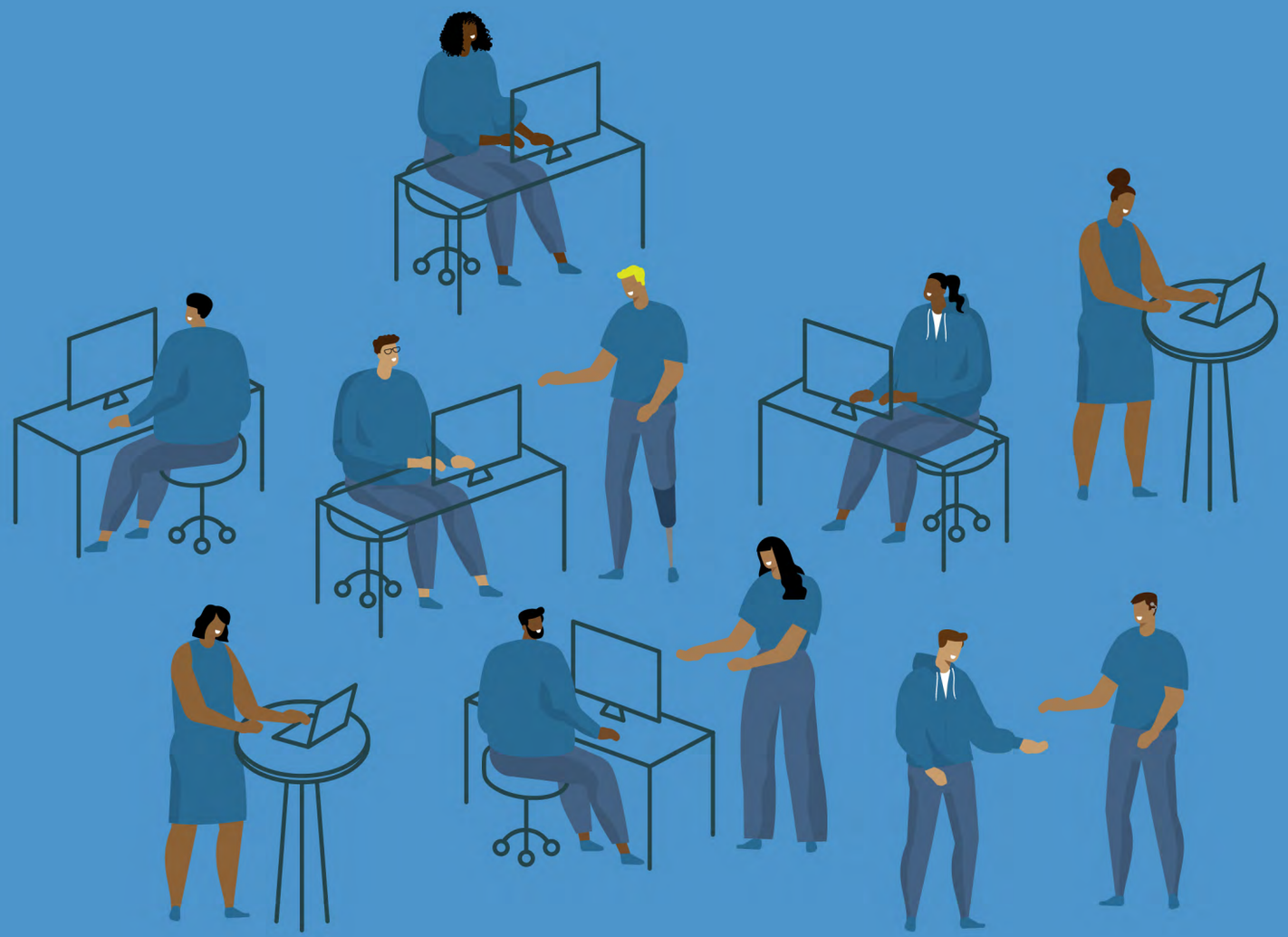
You may be reflecting on how much more you know as a team now than when you first wrote your codebase, and be tempted to start from scratch and rewrite your codebase with all your hard-won knowledge. Resist the urge. Things are going to be messy but push on. You're just hitting your stride as a business. Contain the mess, clean as you go, but don't halt your progress with a massive rewrite.

✖ **Assume shared context.**

Up to about 20 people, you haven't broken into teams, so you can assume that people have context (that's a hack — not a strategy). Any larger than that, and you can't assume that everyone shares the same context. Invest in shared context. Enforce that with tooling, for example, making a build fail in CI if you didn't comment correctly. Make sure people understand why it's enforced. It will save you from writing infinite documents that no one will read. Using tests as an example of shared contexts, a test is an expression of expected behavior. But a lot of the time you read old tests and don't understand why they would behave like that. If you broke a test, but there's no documentation or error message, how can you know if it's trustworthy?

Mid stage

50–150 engineers



Now you're big. You can't fight it anymore: you're probably going to have some services.

Your software has gotten complex enough, your number of pipelines has increased in order to increase delivery velocity. You probably want to decouple units of work. Before you go splinter off completely, focus on pulling out the patterns in your delivery pipeline, and in your approach, so that you're not creating chaos.

As you do move into services, continue to manage the fracturing as much as possible. Pick one thing and get it right. Figure out what a build and deploy pipeline looks like for a single service, not for ten services at the same time.

Consistency is the name of the game. This is not a place to let a thousand flowers bloom. Instead: make a standard, then replicate.

Treat your pipeline design the same way you'd treat any software design. Build it once, try it out, build it again, do some copy-paste. Figure out where it's breaking down. Use the Rule of Three: at the third case, you can build an abstraction. Before that, trying to build a "perfect" abstraction is a waste of time because you don't understand it yet. You'll burn cycles and be wrong anyway.

Time spent on building perfect abstraction on the first try is time wasted.

Another thing: the random social interactions that reinforce structure and consistency start to fail at this size. So folks instead just focus on the effectiveness of their team because they don't have the perspective of what other teams are doing.

[Rule of Three]

Two instances of similar code don't need to be refactored, but at the third instance, the code should be extracted into a new procedure.
(popularized by Martin Fowler in Refactoring and attributed to Don Roberts.)

You might be on your way to having a dedicated Developer Experience and Tooling team. In the meantime, having one team come up with a solution and then distributing it across all other teams can be a helpful way to bridge the gap. This approach also avoids the risk of building a culture where delivery is someone else's problem — sharing these responsibilities keeps the entire team engaged in the outcome.

What to do

✔ **Invest in consistency across delivery pipelines.**

The trick here is finding the balance of assigning real ownership without the rest of the team thinking it's someone else's responsibility. Maintain a culture where everyone believes effective software delivery is part of their job, and then give someone the ownership to make that a reality. Otherwise everyone solves their own problems in their own team, duplicating work and losing out on shared lessons.

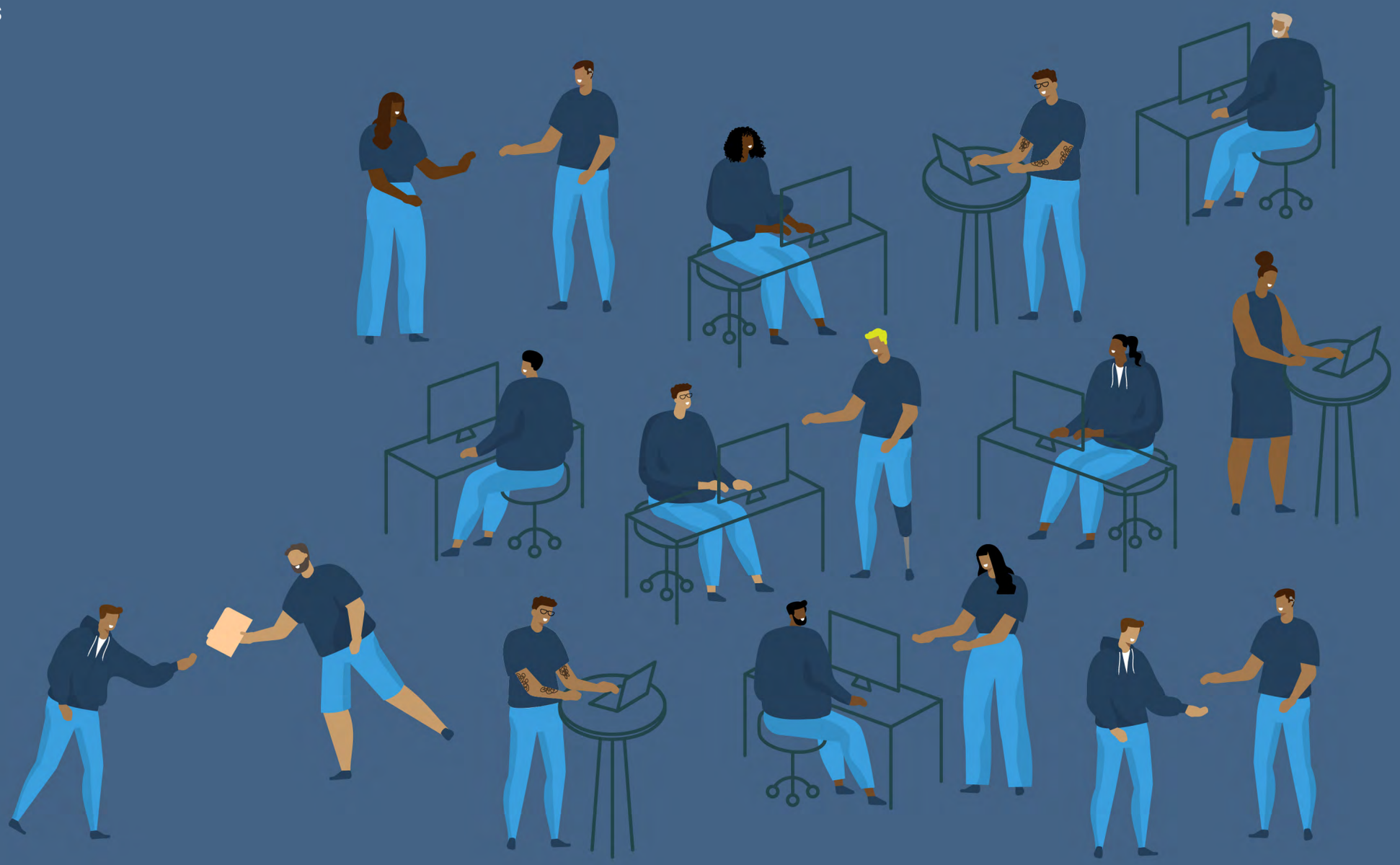
What not to do

✖ **Give teams free reign.**

Don't be lured into the false sense of velocity you feel when letting everyone operate independently — dozens of teams creating hundreds of local efficiencies will come at the cost of long-term complexity.

Growth stage

150–500 engineers



This is the stage at which all the placeholders you created, and culture you enforced through process, starts to really pay off.

You're growing at an alarming rate so anything that comes easily based on your prior decisions will be a huge advantage. At first, you had an inkling that you wanted to do something and left a space for it. Then you replaced that space with a free tool, then a paid tool, then a person, and then a whole team, all because you thoughtfully left a place for that team to plug in. At this point, you may have a team of teams (SRE, Developer Experience, Release Engineering, etc.) responsible for thinking about tooling for the rest of your engineering organization, and that is a real transition point.

> *A note: don't be overly dogmatic about this. Find the places where consistency is driving the most value, and the places where it will create more overhead than value. Balance enforcing consistency with letting folks get things done.*

Put your shared tooling experts on the projects that your team will use the most, not on every single instance of someone doing something twice.

At some point there will be places where the value of consistency is outweighed by the overhead. Knowing where to let that happen is a real challenge — always be aware of your tradeoffs

What to do

Invest in systems and process, not personal relationships.

You're past the point of being able to understand what's happening by just looking at everything. Now systems keep things working, not people.

As you are completing the transition from personal relationships to systems, make it clear how your operational tooling is going to work. In the short term, you'll get better feedback about what's happening. Admittedly, it can be hard to build the business case for setting up this tooling, but again it's one of those things that is super cheap to do early and more expensive as you go. Start small. Do a little every day. Keep going.

Clear team boundaries are essential, such that you can deploy your service irrespective of what's going on elsewhere in the system, or who you are interfacing with. Here is where the real value of autonomy comes to bear: you can focus on getting your work done without interfacing with specific people to smooth the path.

BONUS FOUNDER TIP

Make time to address and work down technical debt.

Jai Das, President & Managing Director, Sapphire Ventures

> *"Technical debt is something you cannot avoid. If you don't address these debts, in time they compound and become an even bigger problem. Technical debt typically shows up in several different ways. Architectural and technological debt shows up in any software that has been written after 4-5 years. Given the rapid change rate of cloud native technologies, software using legacy technology will also have shortcomings in the architecture that make it hard to extend and scale. Over time, software accumulates code debt. Developers don't always write clean code with well-defined interfaces and API's. Software also accumulates testing and documentation debt. Very few developers adequately document their code or write tests with adequate coverage. Given that you cannot avoid technical debt, the best approach is to proactively focus on fixing it.. Most good software team leads budget six months every 4-5 years when they stop adding any new features and just focus on resolving technical debt.*

What not to do

Reward heros.

You can no longer find the expert and get them to fix something. At this point if you don't have a tool to manage something, you're in trouble. This looks like good documentation, clear tools, and clear process for dealing with issues. No more patching the boat; you need robust infrastructure and operational tooling. Don't have your team rely on personal relationships to get things done.

Try to fix everything.

Now you've got 150+ engineers working on a system. Some parts of your codebase will be untouched since the point at which you had 10 engineers. Others will be actively touched every day. Worry about those first. You're more likely to break the legacy (aka legendary) code than to improve it. If folks are working on it every day, fix it first.

Closing thoughts

Keep innovating as you grow

By 500+ engineers, congratulations: you are no longer a startup. You are doing awesome. You got this far by becoming an amazing software delivery organization, so don't quit now.

Know that every other software organization that's bigger than you is trying to operate like you.

Know that every other software organization that's bigger than you is trying to operate like you, so don't get sucked into their world of slow legacy processes; keep being the model.

Specific implementations will change as you continue to grow, but don't lose sight of the approaches that got you here. Don't back away from your CI/CD culture. Yes, you need more systems and processes, but you'll adopt them with the same discerning eye you've always applied to any new tool or process: enabling velocity.

As you continue to scale, new stakeholders like auditors will make demands on you as an organization. Understand the outcomes they need, but don't assume you have to follow their recommendations on how to get there. They're not on the bleeding edge of software development. As a successful startup with all this growth under your belt, you are. Build tools that make it possible to still be an awesome software team. Don't fall off the train because you're worried about operating at your size.

Thanks for reading. If you can implement some of the mindset I've shared here, you will be among the highest—performing software delivery organizations in the world - and that's something to be massively proud of.